ASSIGNNMENT -1.5

HT-NO:2303A51260

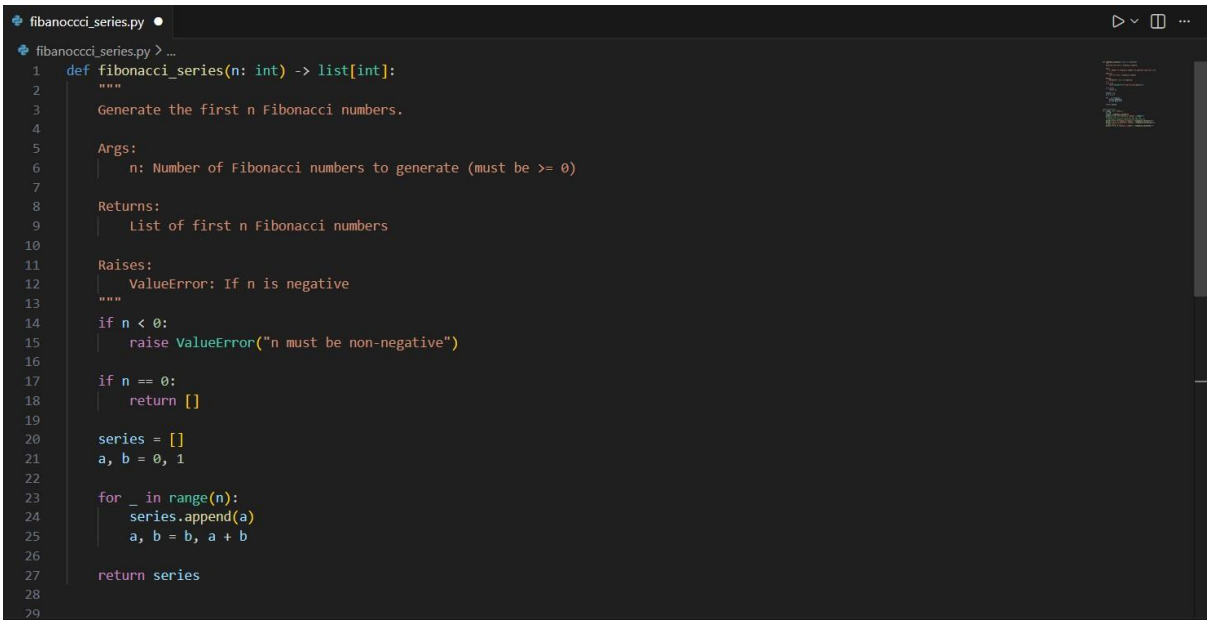
TASK-1

Task : Generate a Fibonacci series using github copilot

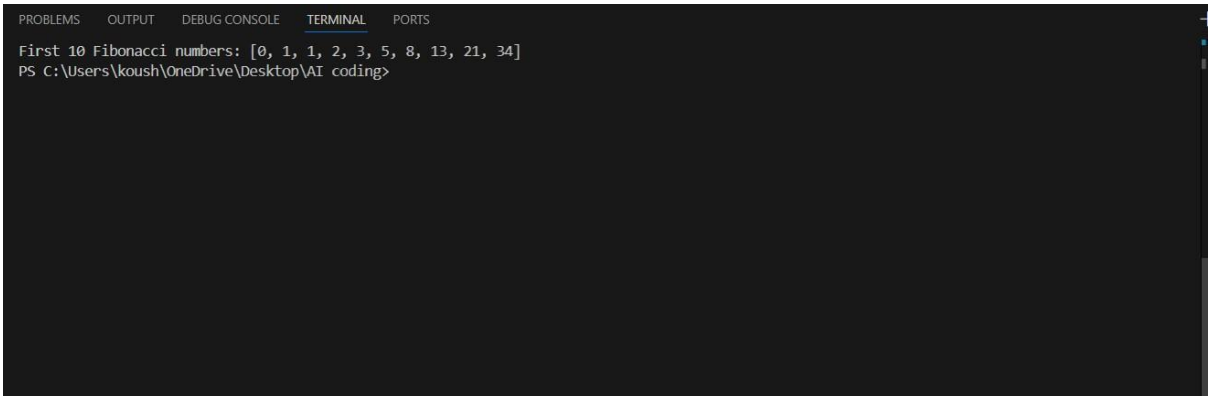
Prompt :

Generate a fibanocci series in a language of python.

Code:



Output:



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

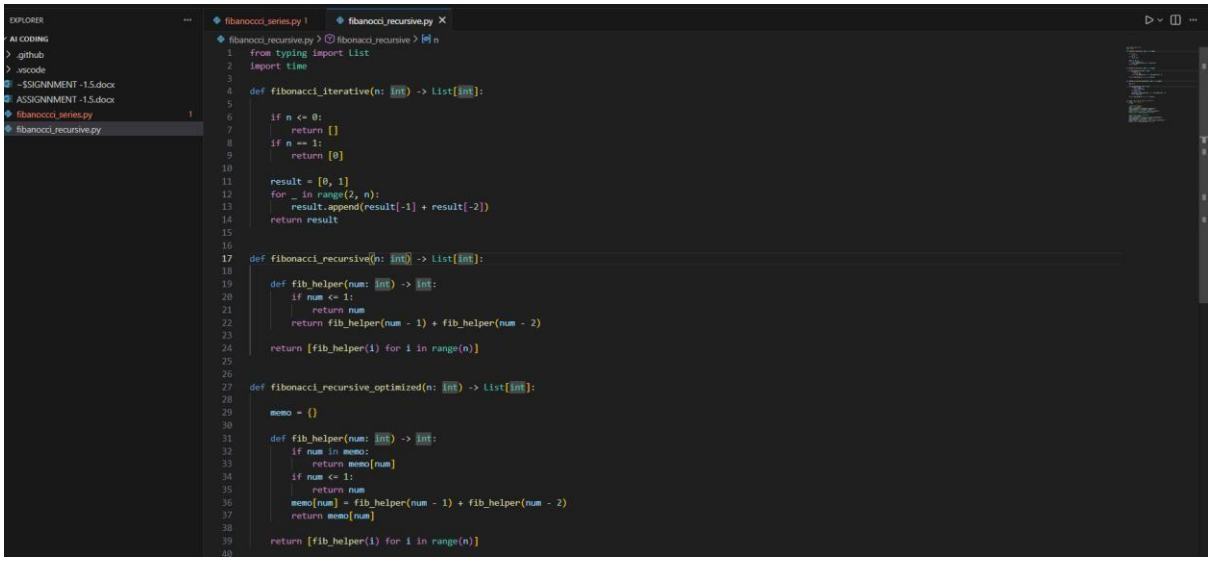
Algorithmic Approaches to String Reversal)

Prompt:

Generate the code of AI-Generated Iterative vs Recursive Fibonacci Approaches (Different

Algorithmic Approaches to String Reversal)

Code:



```python
from typing import List
import time

def fibonacci_iterative(n: int) -> List[int]:

    if n <= 0:
        return []
    if n == 1:
        return [0]

    result = [0, 1]
    for _ in range(2, n):
        result.append(result[-1] + result[-2])
    return result


def fibonacci_recursive(n: int) -> List[int]:

    def fib_helper(num: int) -> int:
        if num <= 1:
            return num
        return fib_helper(num - 1) + fib_helper(num - 2)

    return [fib_helper(i) for i in range(n)]


def fibonacci_recursive_optimized(n: int) -> List[int]:

    memo = {}

    def fib_helper(num: int) -> int:
        if num in memo:
            return memo[num]
        if num <= 1:
            return num
        memo[num] = fib_helper(num - 1) + fib_helper(num - 2)
        return memo[num]

    return [fib_helper(i) for i in range(n)]
```

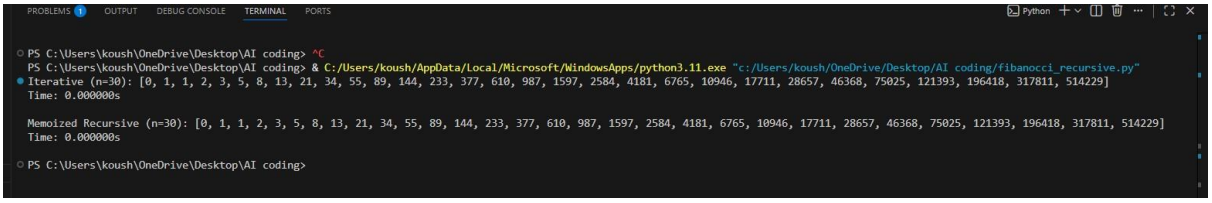
```python
    return [fib_helper(i) for i in range(n)]


# Example usage and performance comparison
if __name__ == "__main__":
    n = 30

    # Iterative approach
    start = time.time()
    result_iterative = fibonacci_iterative(n)
    time_iterative = time.time() - start
    print(f"Iterative (n={n}): {result_iterative}")
    print(f"Time: {time_iterative:.6f}s\n")

    # Optimized recursive approach
    start = time.time()
    result_optimized = fibonacci_recursive_optimized(n)
    time_optimized = time.time() - start
    print(f"Memoized Recursive (n={n}): {result_optimized}")
    print(f"Time: {time_optimized:.6f}s\n")
```

OUTPUT:



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:

```python
# Comparative Analysis - Procedural vs Modular Approach

def fibonacci_series(n: int) -> list[int]:
    """Generates the first n Fibonacci numbers using an iterative approach."""
    if n < 0:
        raise ValueError("n must be a non-negative integer.")
    elif n == 0:
        return []
    elif n == 1:
        return [0]

    fibonacci_numbers = [0, 1]
    for i in range(2, n):
        current, next_value = fibonacci_numbers[-2], fibonacci_numbers[-1]
        fibonacci_numbers.append(current + next_value)

    return fibonacci_numbers

# Procedural Approach
def procedural_fibonacci(n):
    """Generates Fibonacci numbers in a procedural manner."""
    if n < 0:
        raise ValueError("n must be a non-negative integer.")
    elif n == 0:
        print([])
        return
    elif n == 1:
        print([0])
        return

    a, b = 0, 1
    result = [a]
    for _ in range(1, n):
        result.append(b)
        a, b = b, a + b
    print(result)

# Example usage
n = 10
print("Modular Approach:", fibonacci_series(n))
print("Procedural Approach:")
procedural_fibonacci(n)
```

OUTPUT:

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                                    Python

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>  Open file in editor (ctrl + click)
                                              & 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>
```

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.

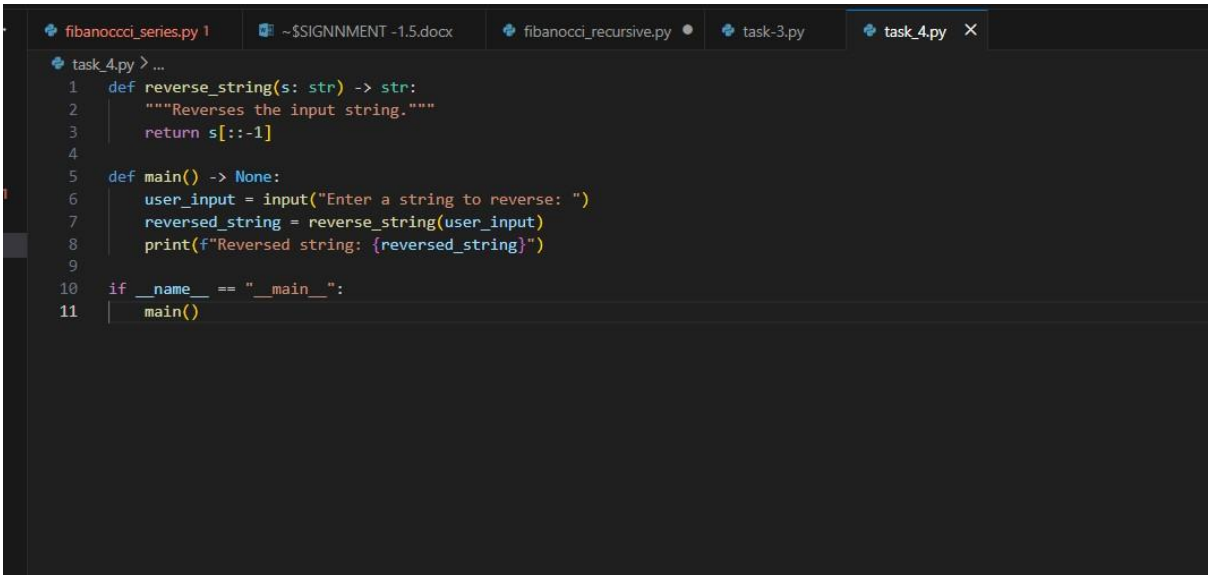
5. The procedural approach directly outputs results, which may be useful for quick checks or debugging.

TASK-4

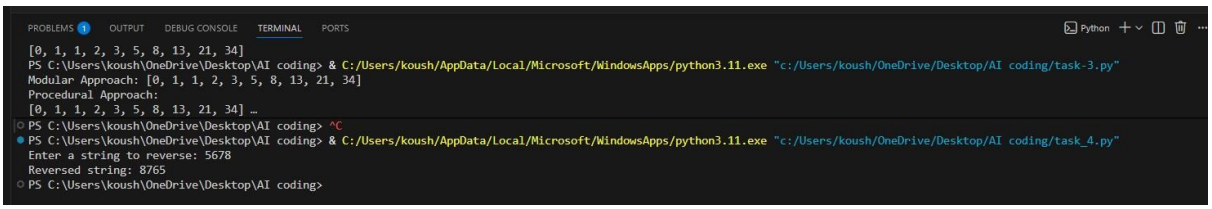
Prompt:

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

Code:



OUTPUT:



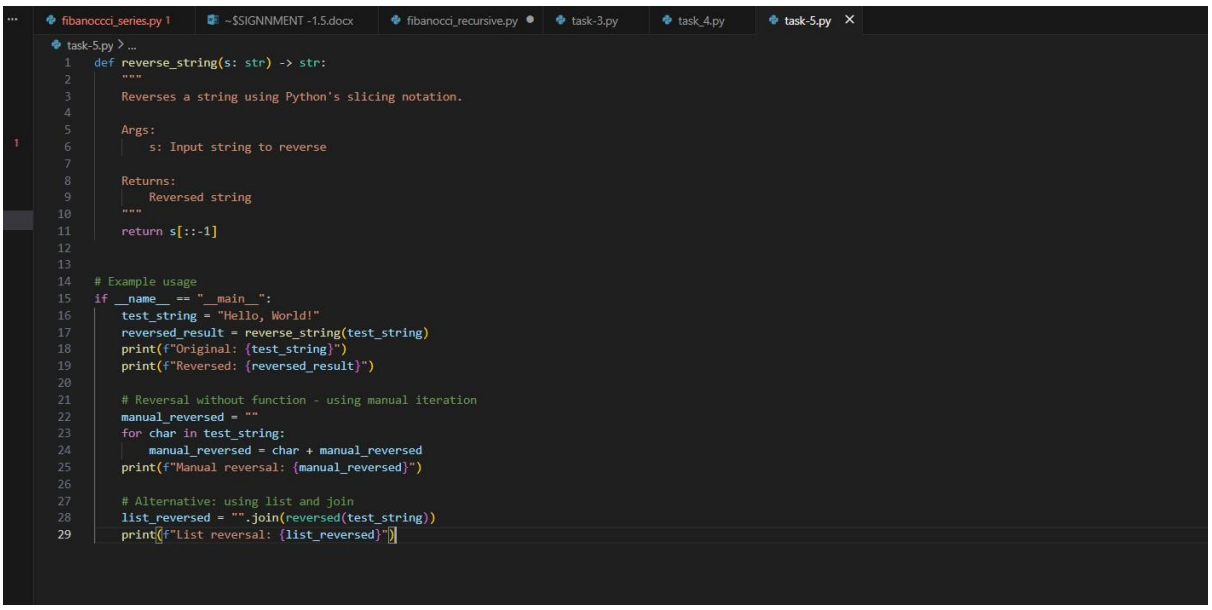
Observation:

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

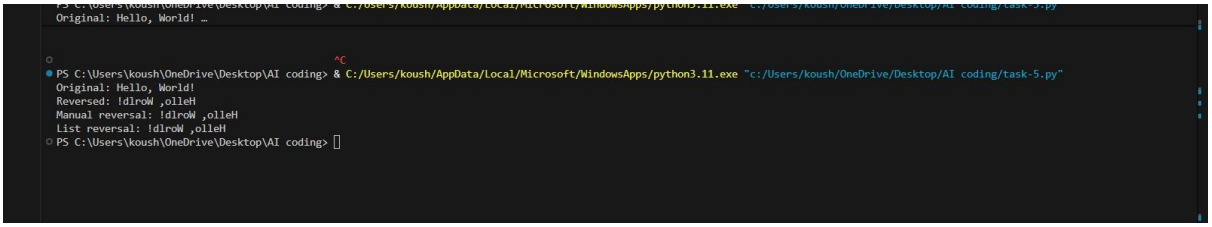
TASK-5

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

Code :



OutPut:



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