Assignment-1

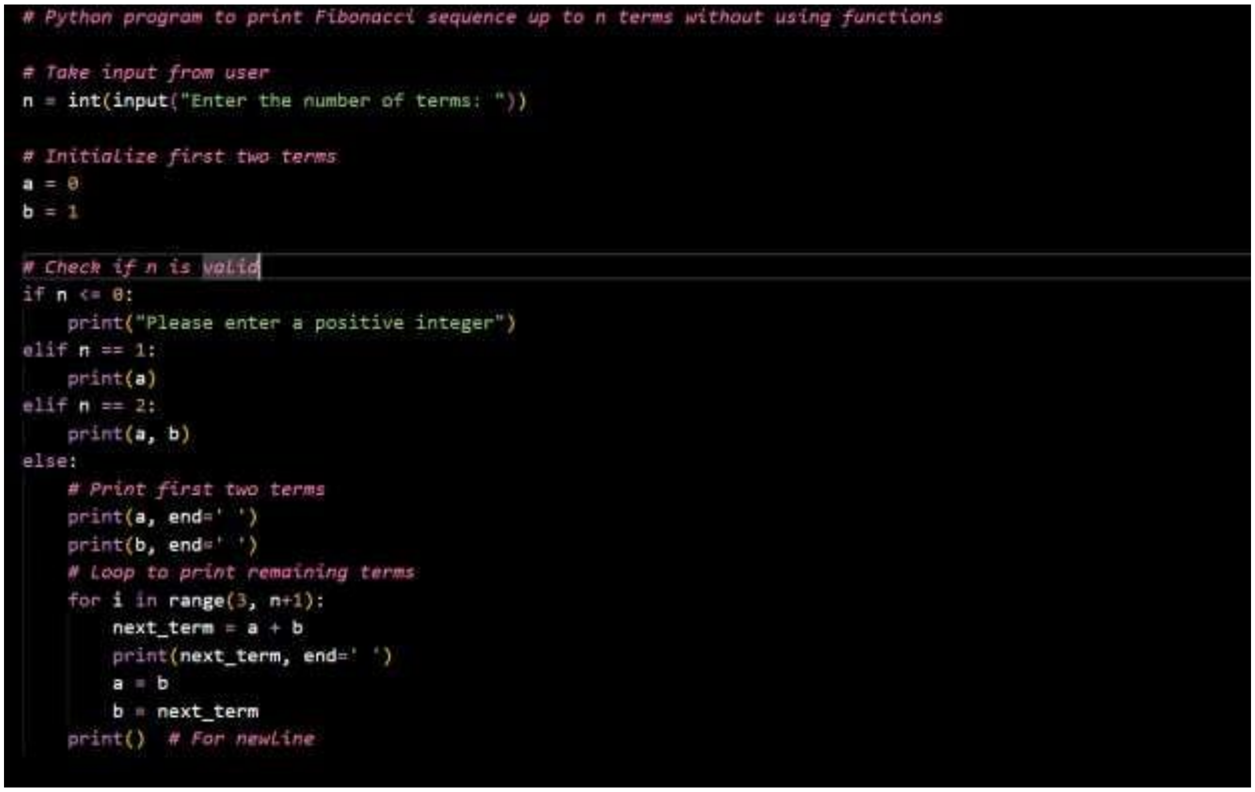
HT No :2303A51905

Batch No:28

Task1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)

Prompt: Write a Python program to print Fibonacci sequence up to n terms without using functionsWrite a Python program to print Fibonacci sequence up to n terms without using functions

Code:



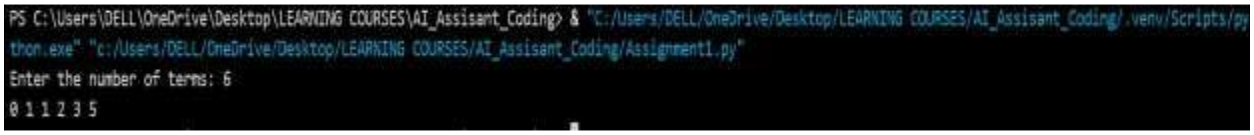
```python
# Python program to print Fibonacci sequence up to n terms without using functions

# Take input from user
n = int(input("Enter the number of terms: "))

# Initialize first two terms
a = 0
b = 1

# Check if n is valid
if n <= 0:
    print("Please enter a positive integer")
elif n == 1:
    print(a)
elif n == 2:
    print(a, b)
else:
    # Print first two terms
    print(a, end=' ')
    print(b, end=' ')
    # Loop to print remaining terms
    for i in range(3, n+1):
        next_term = a + b
        print(next_term, end=' ')
        a = b
        b = next_term
    print()  # For newline
```

Output:



```
PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "C:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assisant_Coding/.venv/Scripts/py
thon.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assisant_Coding/Assignment1.py"
Enter the number of terms: 6
0 1 1 2 3 5
```

Justification

1. Uses direct logic to clearly demonstrate Fibonacci sequence generation.

2. Avoids functions to keep the code simple and easy to understand.

3. Suitable for quick testing and learning basic looping concepts.

4. Helps beginners focus on algorithm flow without modular complexity.

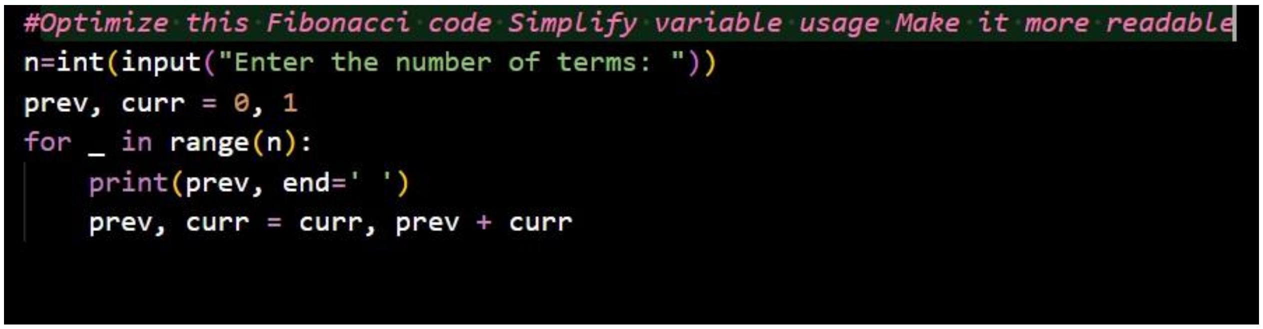
5. Ensures minimal overhead for a small, single-use program.

6. Ideal for rapid prototyping scenarios.

Task2 : AI Code Optimization & Cleanup (Improving Efficiency)

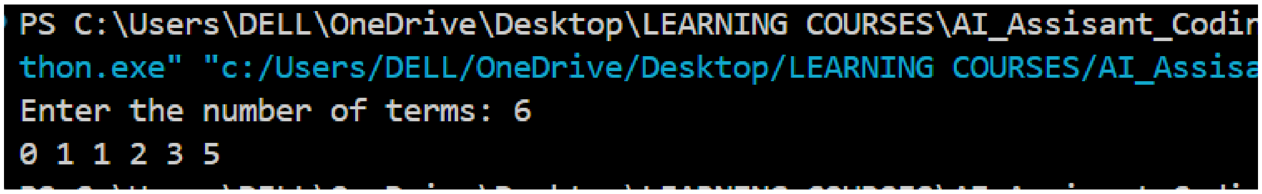
Prompt: Optimize the Fibonacci code Simplify variable usage Make it more readable

Code:



```
#Optimize this Fibonacci code Simplify variable usage Make it more readable
n=int(input("Enter the number of terms: "))
prev, curr = 0, 1
for _ in range(n):
    print(prev, end=' ')
    prev, curr = curr, prev + curr
```

Output:



```
PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Codin
thon.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assisa
Enter the number of terms: 6
0 1 1 2 3 5
```

Justification:

1. Eliminates unnecessary variables to reduce memory usage.

2. Simplifies loop logic for better readability and maintainability.

3. Produces the same output with cleaner and shorter code.

4. Improves performance by avoiding redundant computations.

5. Makes the code easier to review and modify.

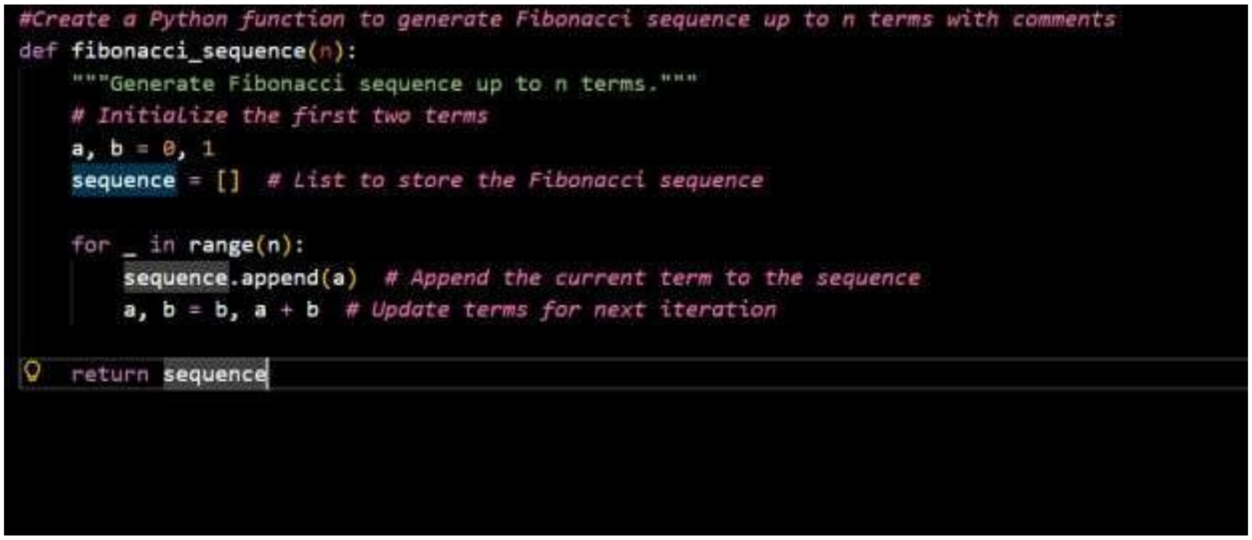
6. Demonstrates AI-assisted code refinement.

Task3: Modular Design Using AI Assistance (Fibonacci Using Functions)

Prompt: Create a Python function to generate Fibonacci sequence up to n terms

with comments

Code:

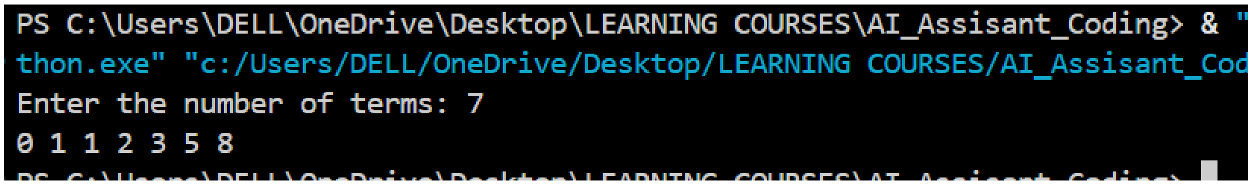


```python
#Create a Python function to generate Fibonacci sequence up to n terms with comments
def fibonacci_sequence(n):
    """Generate Fibonacci sequence up to n terms."""
    # Initialize the first two terms
    a, b = 0, 1
    sequence = []  # List to store the Fibonacci sequence

    for _ in range(n):
        sequence.append(a)  # Append the current term to the sequence
        a, b = b, a + b  # Update terms for next iteration

    return sequence
```

Output:



```
PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "
thon.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assisant_Cod
Enter the number of terms: 7
0 1 1 2 3 5 8
```

Justification:

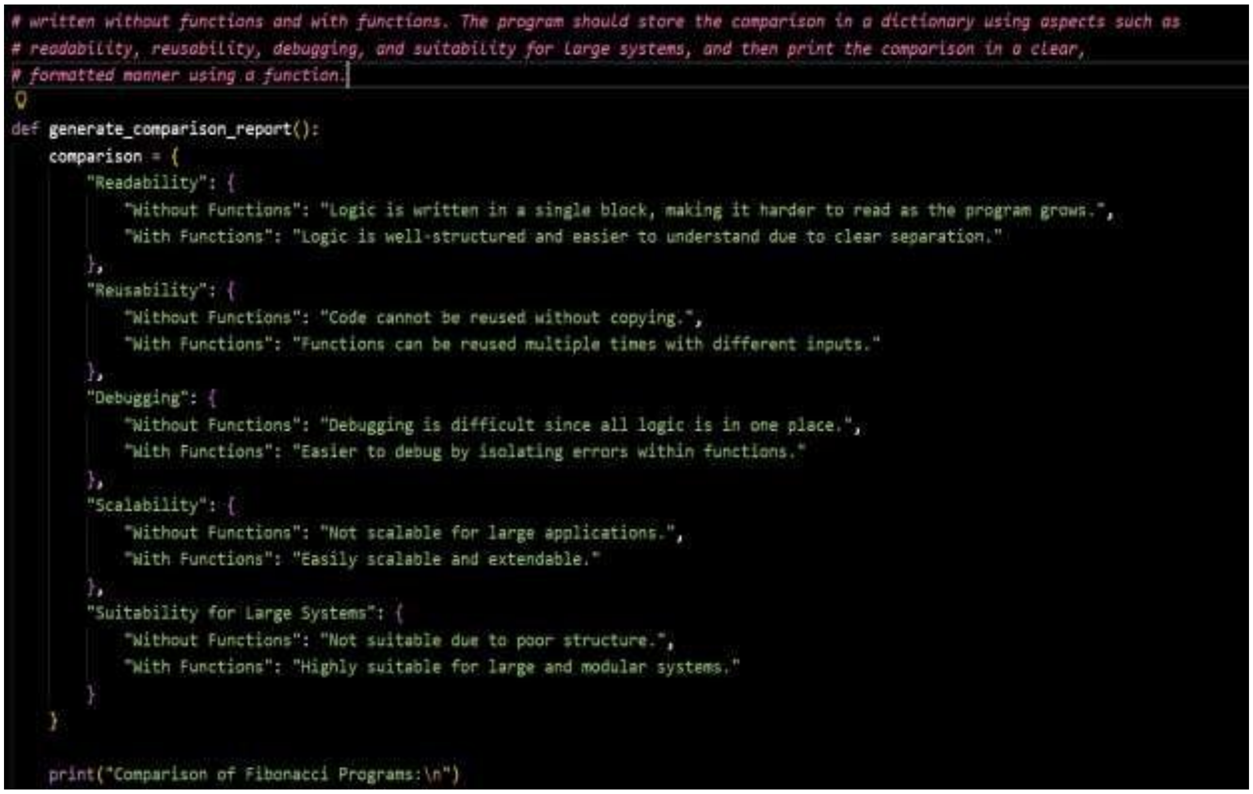
1. Encapsulates logic inside a function for reuse.
2. Improves code organization and readability.
3. Simplifies debugging by isolating Fibonacci logic.
4. Allows the function to be used in multiple modules.
5. Enhances maintainability for future changes.

6. Aligns with best practices in software development.

<span>Task4:</span> Comparative Analysis – Procedural vs Modular Fibonacci Code

Prompt: Generate two Python programs for the Fibonacci sequence, one written procedurally without using any user-defined functions and the other using a user-defined function. After generating the code, review and compare both implementations by explaining the differences in code readability, reusability, ease of debugging, scalability, and suitability for large applications.

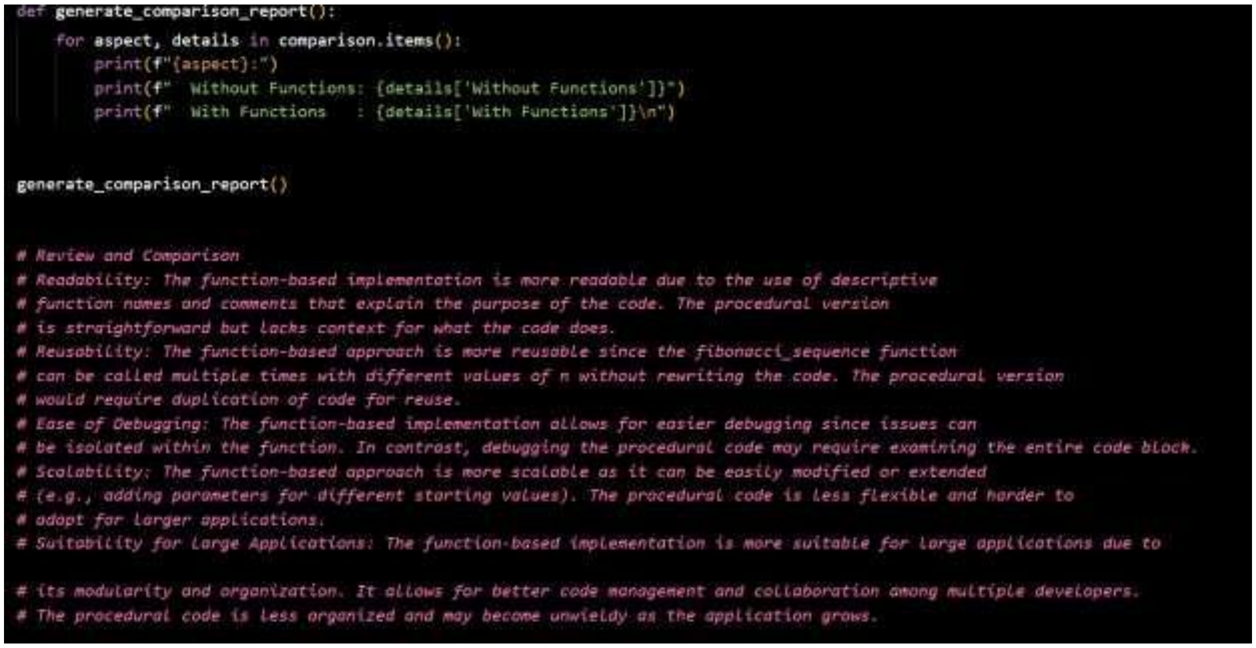
Code:



```
# written without functions and with functions. The program should store the comparison in a dictionary using aspects such as
# readability, reusability, debugging, and suitability for large systems, and then print the comparison in a clear,
# formatted manner using a function.

def generate_comparison_report():
    comparison = {
        "Readability": {
            "Without Functions": "Logic is written in a single block, making it harder to read as the program grows.",
            "With Functions": "Logic is well-structured and easier to understand due to clear separation."
        },
        "Reusability": {
            "Without Functions": "Code cannot be reused without copying.",
            "With Functions": "Functions can be reused multiple times with different inputs."
        },
        "Debugging": {
            "Without Functions": "Debugging is difficult since all logic is in one place.",
            "With Functions": "Easier to debug by isolating errors within functions."
        },
        "Scalability": {
            "Without Functions": "Not scalable for large applications.",
            "With Functions": "Easily scalable and extendable."
        },
        "Suitability for Large Systems": {
            "Without Functions": "Not suitable due to poor structure.",
            "With Functions": "Highly suitable for large and modular systems."
        }
    }

    print("Comparison of Fibonacci Programs:\n")
```



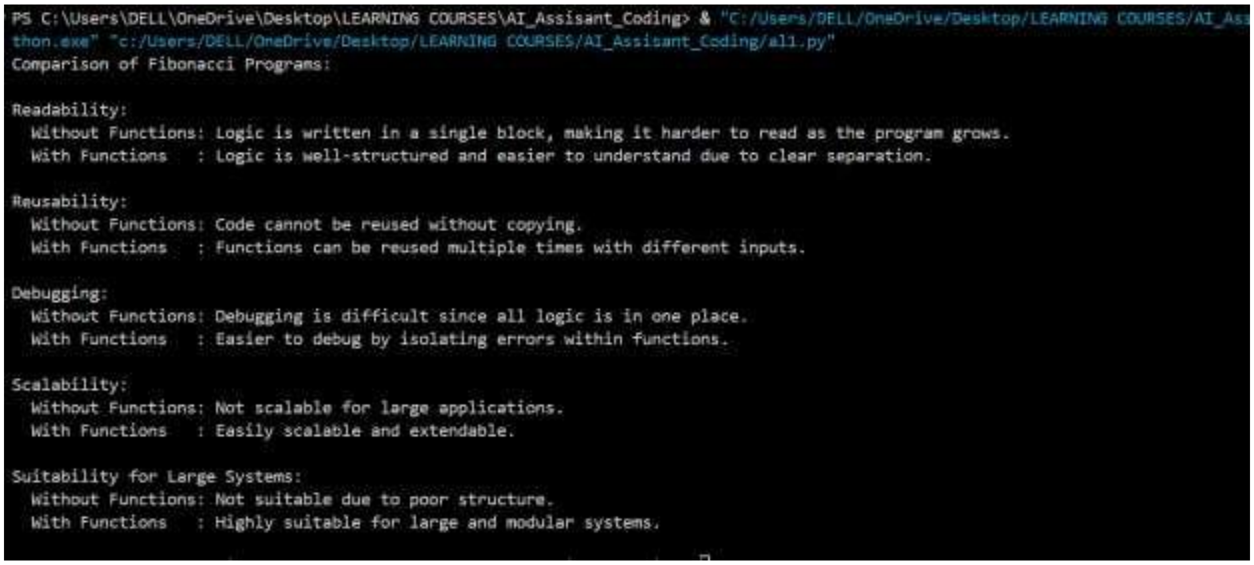
```
def generate_comparison_report():
    for aspect, details in comparison.items():
        print(f"{aspect}:")
        print(f"  Without Functions: {details['Without Functions']}")
        print(f"  With Functions    : {details['With Functions']}\n")


generate_comparison_report()


# Review and Comparison
# Readability: The function-based implementation is more readable due to the use of descriptive
# function names and comments that explain the purpose of the code. The procedural version
# is straightforward but lacks context for what the code does.
# Reusability: The function-based approach is more reusable since the fibonacci_sequence function
# can be called multiple times with different values of n without rewriting the code. The procedural version
# would require duplication of code for reuse.
# Ease of Debugging: The function-based implementation allows for easier debugging since issues can
# be isolated within the function. In contrast, debugging the procedural code may require examining the entire code block.
# Scalability: The function-based approach is more scalable as it can be easily modified or extended
# (e.g., adding parameters for different starting values). The procedural code is less flexible and harder to
# adopt for larger applications.
# Suitability for Large Applications: The function-based implementation is more suitable for large applications due to

# its modularity and organization. It allows for better code management and collaboration among multiple developers.
# The procedural code is less organized and may become unwieldy as the application grows.
```

Output:



```
PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "C:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_As
thon.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assisant_Coding/all.py"
Comparison of Fibonacci Programs:

Readability:
  Without Functions: Logic is written in a single block, making it harder to read as the program grows.
  With Functions    : Logic is well-structured and easier to understand due to clear separation.

Reusability:
  Without Functions: Code cannot be reused without copying.
  With Functions    : Functions can be reused multiple times with different inputs.

Debugging:
  Without Functions: Debugging is difficult since all logic is in one place.
  With Functions    : Easier to debug by isolating errors within functions.

Scalability:
  Without Functions: Not scalable for large applications.
  With Functions    : Easily scalable and extendable.

Suitability for Large Systems:
  Without Functions: Not suitable due to poor structure.
  With Functions    : Highly suitable for large and modular systems.
```

Justification:

1. Understanding Code Structure Differences

This task helps in understanding the structural difference between procedural code and modular code. By comparing Fibonacci implementations with and without functions, it becomes clear how code organization affects clarity and flow. Procedural code places all logic in one block, while modular code separates logic into meaningful units.

2. Importance of Readability and Maintainability
The comparison highlights that function-based code is easier to read and maintain. Descriptive function names and clear separation of logic make the program understandable even for new developers. Procedural code may become confusing as the program size increases.

3. Reusability of Code
This task justifies why modular programming is preferred in real applications. Function-based Fibonacci logic can be reused multiple times without rewriting code, whereas procedural code requires duplication, increasing the chance of errors.

4. Ease of Debugging and Testing
Debugging is simpler in modular code because errors can be isolated within a specific function. In procedural code, identifying issues is harder since the entire logic is written in one place. This comparison emphasizes the role of modularity in efficient debugging.

5. Scalability for Large Applications
The analysis shows that modular code is more scalable and adaptable for large systems. Additional features or modifications can be added easily without affecting the existing code. Procedural code is less flexible and difficult to extend.

Task5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)

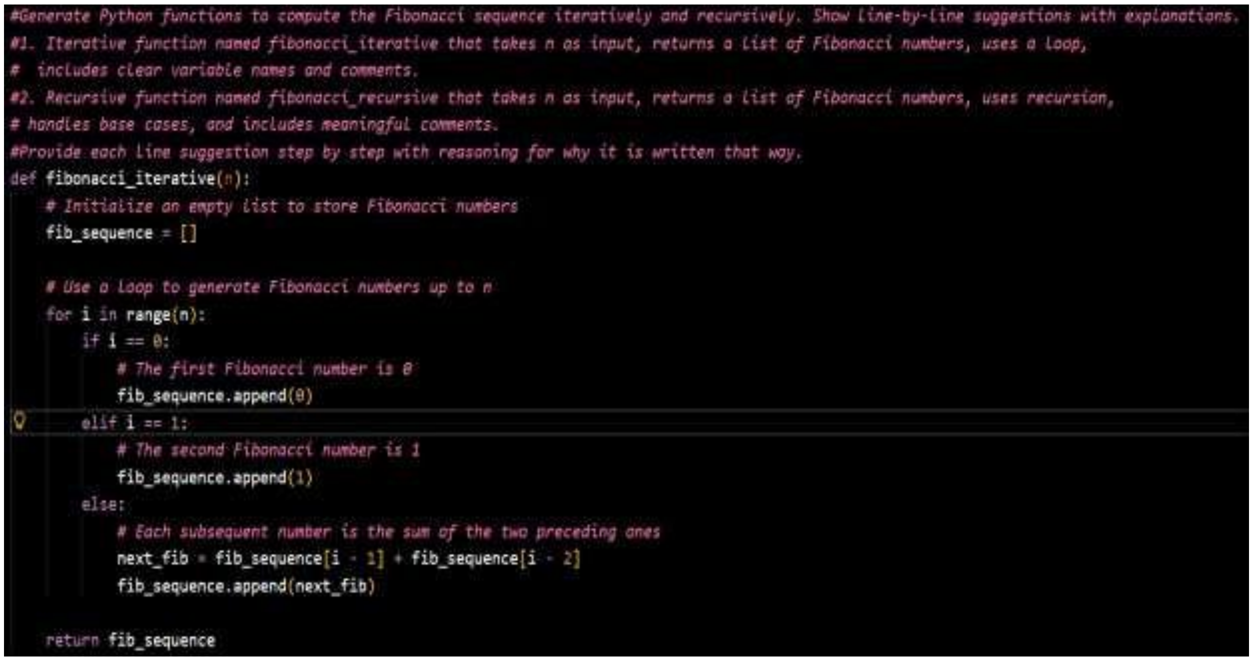
Prompt: Generate Python functions to compute the Fibonacci sequence iteratively and recursively. Show line-by-line suggestions with explanations. Requirements:

1. Iterative function named fibonacci_iterative that takes n as input, returns a list of Fibonacci numbers, uses a loop, includes clear variable names and comments.

2. Recursive function named fibonacci_recursive that takes n as input, returns a list of Fibonacci numbers, uses recursion, handles base cases, and includes meaningful comments.

Provide each line suggestion step by step with reasoning for why it is written that way.

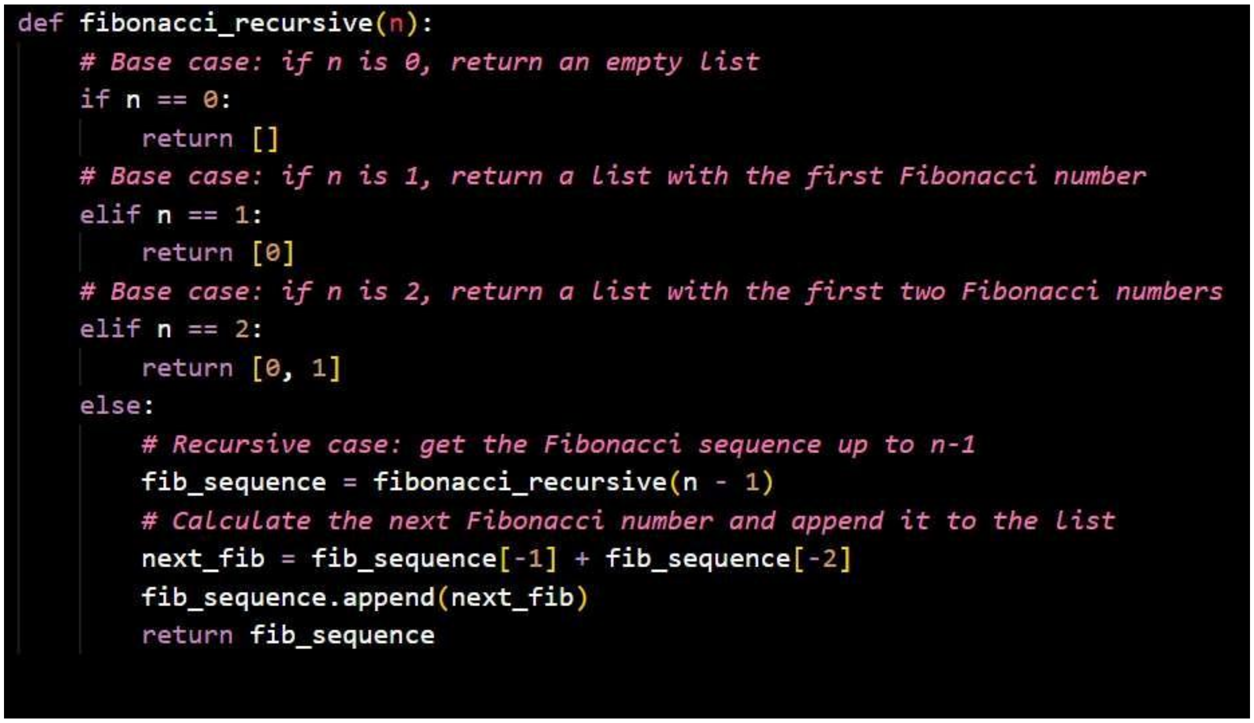
Code:



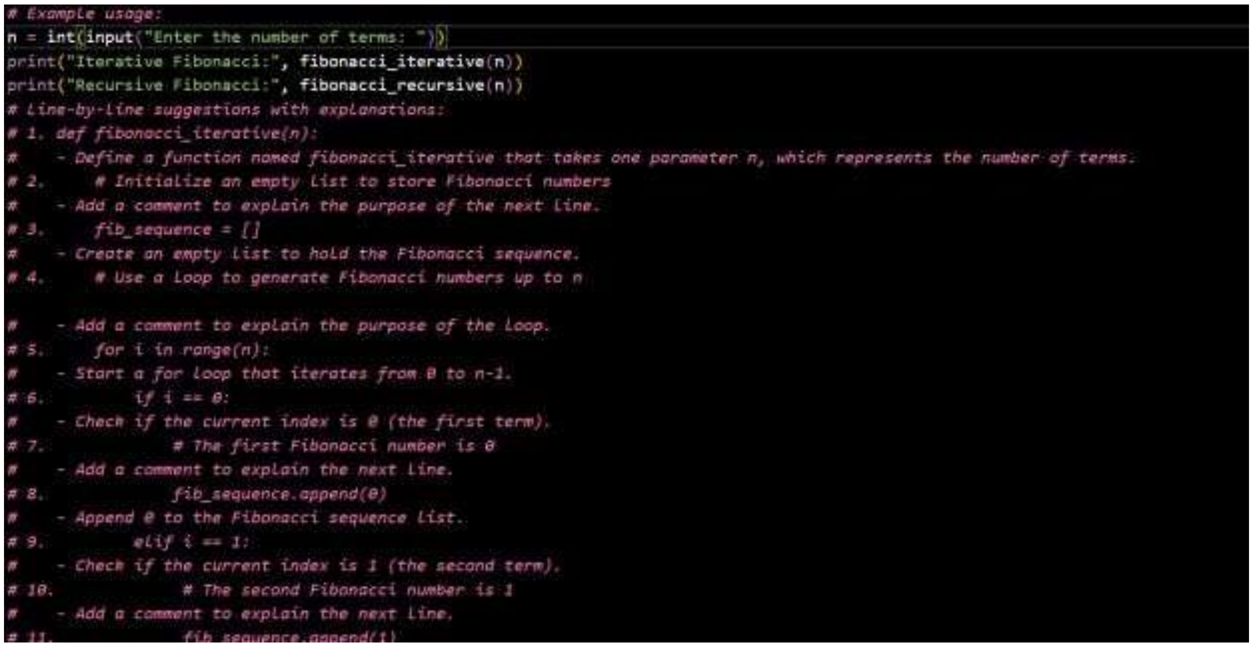
```python
#Generate Python functions to compute the Fibonacci sequence iteratively and recursively. Show line-by-line suggestions with explanations.
#1. Iterative function named fibonacci_iterative that takes n as input, returns a list of Fibonacci numbers, uses a loop,
#  includes clear variable names and comments.
#2. Recursive function named fibonacci_recursive that takes n as input, returns a list of Fibonacci numbers, uses recursion,
# handles base cases, and includes meaningful comments.
#Provide each line suggestion step by step with reasoning for why it is written that way.
def fibonacci_iterative(n):
    # Initialize an empty list to store Fibonacci numbers
    fib_sequence = []

    # Use a loop to generate Fibonacci numbers up to n
    for i in range(n):
        if i == 0:
            # The first Fibonacci number is 0
            fib_sequence.append(0)
        elif i == 1:
            # The second Fibonacci number is 1
            fib_sequence.append(1)
        else:
            # Each subsequent number is the sum of the two preceding ones
            next_fib = fib_sequence[i - 1] + fib_sequence[i - 2]
            fib_sequence.append(next_fib)

    return fib_sequence
```

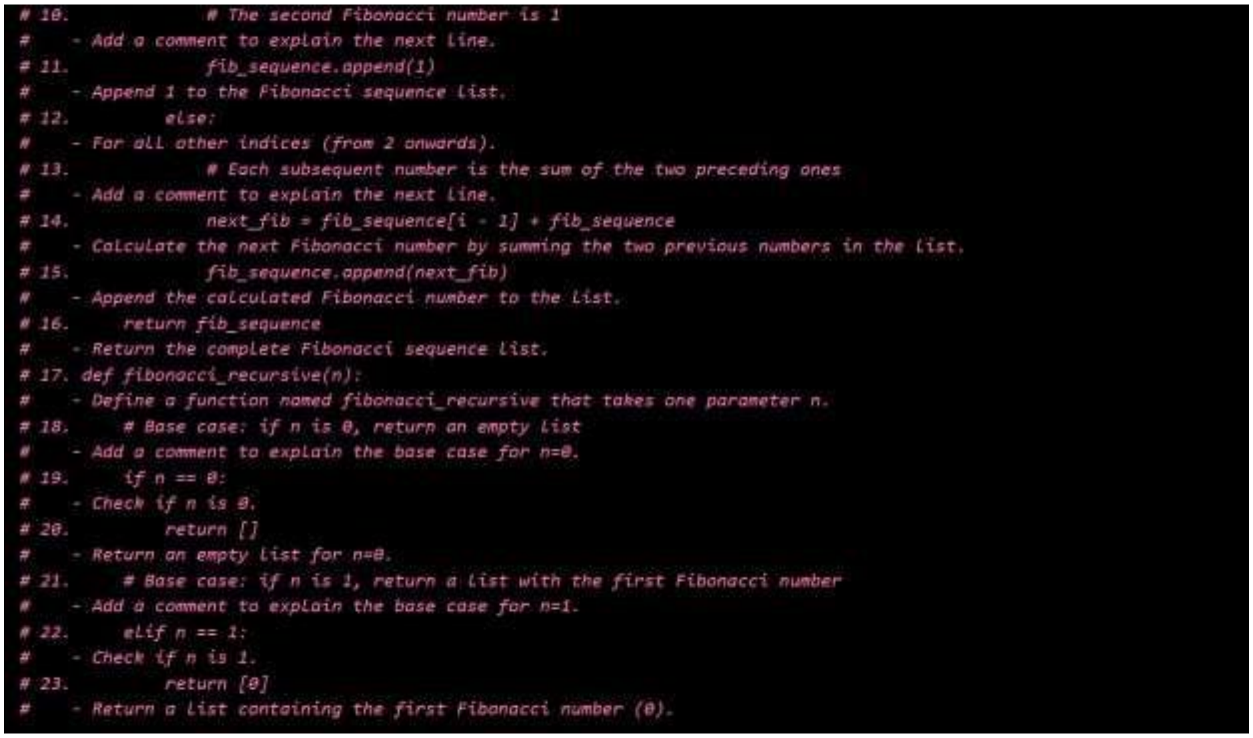


```python
def fibonacci_recursive(n):
    # Base case: if n is 0, return an empty list
    if n == 0:
        return []
    # Base case: if n is 1, return a list with the first Fibonacci number
    elif n == 1:
        return [0]
    # Base case: if n is 2, return a list with the first two Fibonacci numbers
    elif n == 2:
        return [0, 1]
    else:
        # Recursive case: get the Fibonacci sequence up to n-1
        fib_sequence = fibonacci_recursive(n - 1)
        # Calculate the next Fibonacci number and append it to the list
        next_fib = fib_sequence[-1] + fib_sequence[-2]
        fib_sequence.append(next_fib)
        return fib_sequence
```

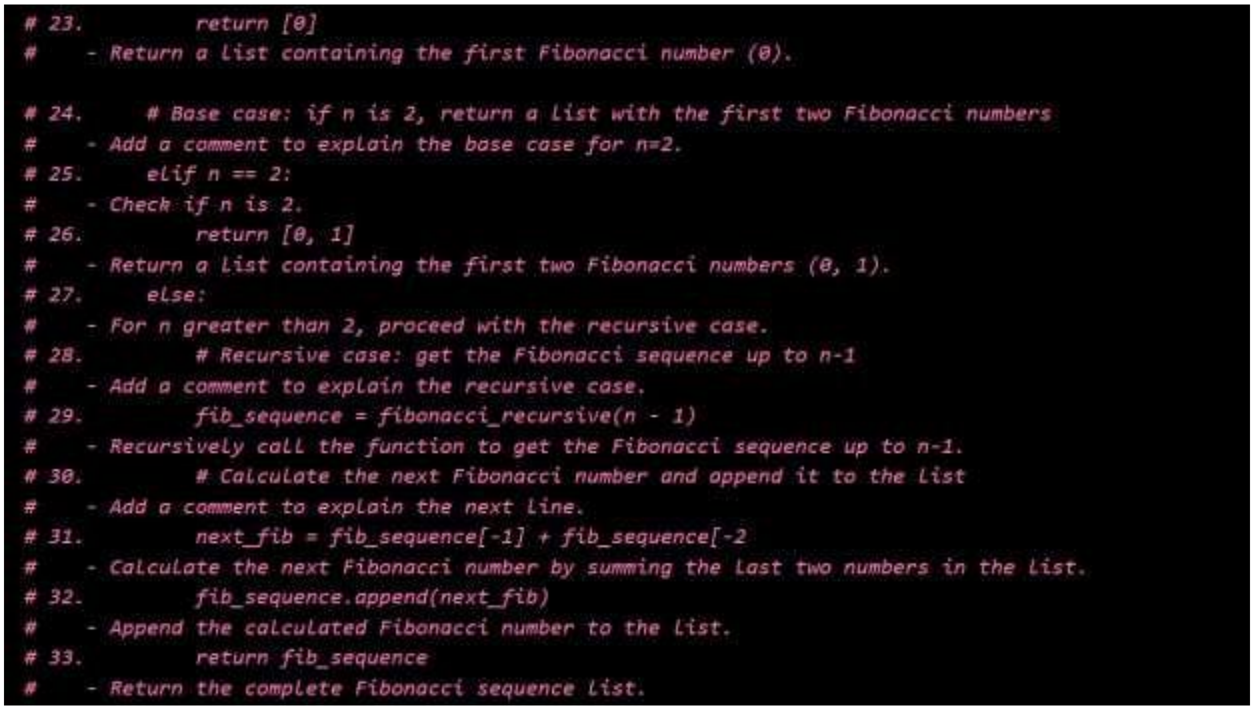


```
# Example usage:
n = int(input("Enter the number of terms: "))
print("Iterative Fibonacci:", fibonacci_iterative(n))
print("Recursive Fibonacci:", fibonacci_recursive(n))
# Line-by-line suggestions with explanations:
# 1. def fibonacci_iterative(n):
#     - Define a function named fibonacci_iterative that takes one parameter n, which represents the number of terms.
# 2.      # Initialize an empty list to store Fibonacci numbers
#     - Add a comment to explain the purpose of the next line.
# 3.      fib_sequence = []
#     - Create an empty list to hold the Fibonacci sequence.
# 4.      # Use a loop to generate Fibonacci numbers up to n

#     - Add a comment to explain the purpose of the loop.
# 5.      for i in range(n):
#     - Start a for loop that iterates from 0 to n-1.
# 6.          if i == 0:
#     - Check if the current index is 0 (the first term).
# 7.              # The first Fibonacci number is 0
#     - Add a comment to explain the next line.
# 8.              fib_sequence.append(0)
#     - Append 0 to the Fibonacci sequence list.
# 9.          elif i == 1:
#     - Check if the current index is 1 (the second term).
# 10.             # The second Fibonacci number is 1
#     - Add a comment to explain the next line.
# 11.             fib_sequence.append(1)
```



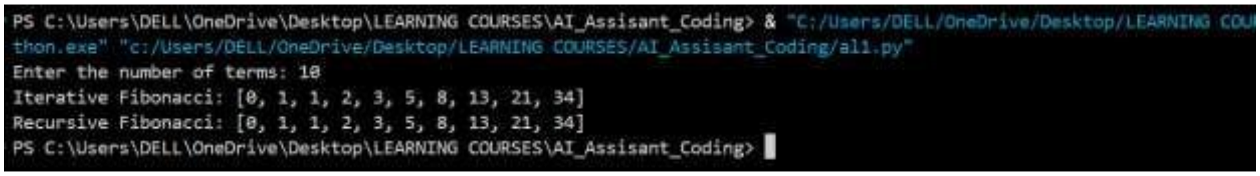
```
# 10.             # The second Fibonacci number is 1
#     - Add a comment to explain the next line.
# 11.             fib_sequence.append(1)
#     - Append 1 to the Fibonacci sequence list.
# 12.         else:
#     - For all other indices (from 2 onwards).
# 13.             # Each subsequent number is the sum of the two preceding ones
#     - Add a comment to explain the next line.
# 14.             next_fib = fib_sequence[i - 1] + fib_sequence
#     - Calculate the next Fibonacci number by summing the two previous numbers in the list.
# 15.             fib_sequence.append(next_fib)
#     - Append the calculated Fibonacci number to the list.
# 16.     return fib_sequence
#     - Return the complete Fibonacci sequence list.
# 17. def fibonacci_recursive(n):
#     - Define a function named fibonacci_recursive that takes one parameter n.
# 18.     # Base case: if n is 0, return an empty list
#     - Add a comment to explain the base case for n=0.
# 19.     if n == 0:
#     - Check if n is 0.
# 20.         return []
#     - Return an empty list for n=0.
# 21.     # Base case: if n is 1, return a list with the first Fibonacci number
#     - Add a comment to explain the base case for n=1.
# 22.     elif n == 1:
#     - Check if n is 1.
# 23.         return [0]
#     - Return a list containing the first Fibonacci number (0).
```



```
# 23.         return [0]
#     - Return a list containing the first Fibonacci number (0).

# 24.     # Base case: if n is 2, return a list with the first two Fibonacci numbers
#     - Add a comment to explain the base case for n=2.
# 25.     elif n == 2:
#     - Check if n is 2.
# 26.         return [0, 1]
#     - Return a list containing the first two Fibonacci numbers (0, 1).
# 27.     else:
#     - For n greater than 2, proceed with the recursive case.
# 28.         # Recursive case: get the Fibonacci sequence up to n-1
#     - Add a comment to explain the recursive case.
# 29.         fib_sequence = fibonacci_recursive(n - 1)
#     - Recursively call the function to get the Fibonacci sequence up to n-1.
# 30.         # Calculate the next Fibonacci number and append it to the list
#     - Add a comment to explain the next line.
# 31.         next_fib = fib_sequence[-1] + fib_sequence[-2
#     - Calculate the next Fibonacci number by summing the last two numbers in the list.
# 32.         fib_sequence.append(next_fib)
#     - Append the calculated Fibonacci number to the list.
# 33.         return fib_sequence
#     - Return the complete Fibonacci sequence list.
```

Output:



```
PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> & "C:/Users/DELL/OneDrive/Desktop/LEARNING COU
thon.exe" "c:/Users/DELL/OneDrive/Desktop/LEARNING COURSES/AI_Assisant_Coding/al1.py"
Enter the number of terms: 10
Iterative Fibonacci: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Recursive Fibonacci: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS C:\Users\DELL\OneDrive\Desktop\LEARNING COURSES\AI_Assisant_Coding> 
```

Justification:

1. The iterative function efficiently generates the Fibonacci sequence using a loop, minimizing memory overhead and avoiding recursion depth limits.

2. The recursive function demonstrates a clear, step-by-step approach that builds the sequence using base cases and recursive calls, which helps in understanding recursion.

3. Both functions return a list of Fibonacci numbers, making the outputs reusable in other parts of a program or for further computations.

4. Comments and meaningful variable names improve code readability, making it easier for others to follow the logic line by line.

5. The iterative approach is better suited for larger values of n due to lower time and space complexity, while the recursive approach is useful for small inputs and educational purposes.

6. Providing line-by-line explanations helps learners understand each operation, including initialization, base cases, loop logic, and recursive computation.