

Lab Assignment 1 – AI Assistant Coding

2303A52173

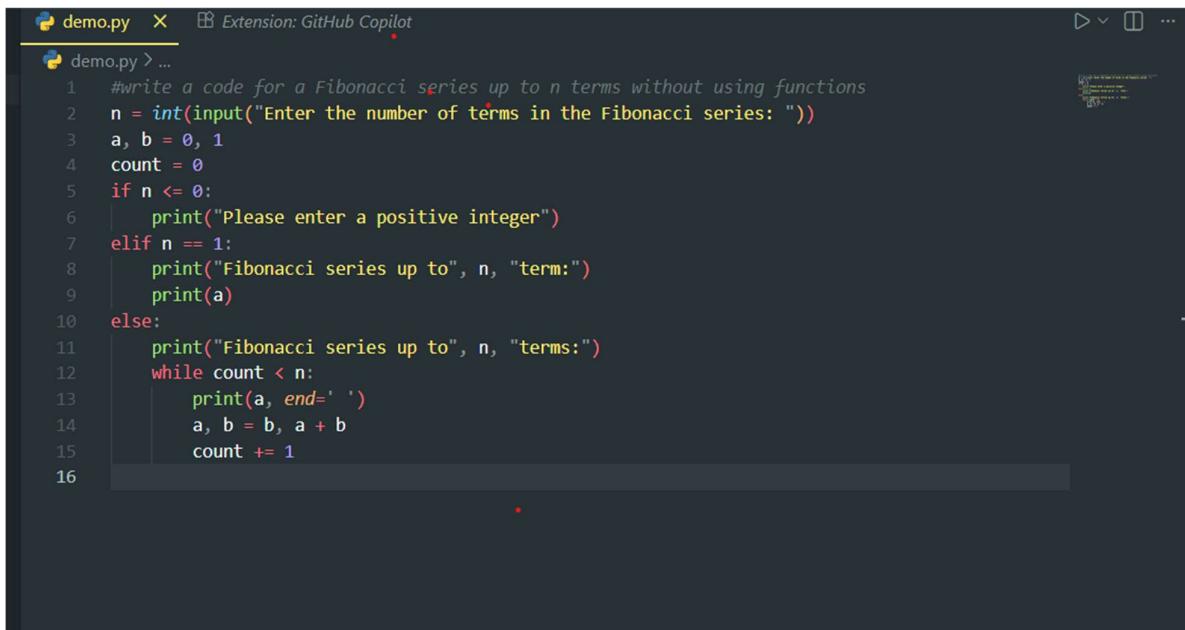
K SAI PRANEETH

Task 1: Fibonacci Without Functions (Procedural)

Prompt used:

write a code for a Fibonacci series up to n terms without using functions

Code:



```
demo.py X Extension: GitHub Copilot
demo.py > ...
1 #write a code for a Fibonacci series up to n terms without using functions
2 n = int(input("Enter the number of terms in the Fibonacci series: "))
3 a, b = 0, 1
4 count = 0
5 if n <= 0:
6     print("Please enter a positive integer")
7 elif n == 1:
8     print("Fibonacci series up to", n, "term:")
9     print(a)
10 else:
11     print("Fibonacci series up to", n, "terms:")
12     while count < n:
13         print(a, end=' ')
14         a, b = b, a + b
15         count += 1
16
```

Output:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS D:\Python> & "C:\Program Files\Python314\python.exe" d:/Python/demo.py
Enter the number of terms in the Fibonacci series: 10
Fibonacci series up to 10 terms:
0 1 1 2 3 5 8 13 21 34
PS D:\Python>
```

Justification:

The logic for calculating the Fibonacci numbers is directly included in the main program without using any user-defined functions, following the requirement of the task.

A simple loop generates the Fibonacci sequence in the most efficient way.

The code is easy to understand for beginners and suitable for quick prototyping.

Task 2: Code Optimization

Prompt used:

Optimize this Fibonacci code and make it more readable

Code:

```
#Optimize this Fibonacci code and make it more readable
n = int(input("Enter the number of terms in the Fibonacci series: "))
if n <= 0:
    print("Please enter a positive integer")
else:
    print("Fibonacci series up to", n, "terms:")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b
```

Output:



The screenshot shows a terminal window with the following content:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS D:\Python> & "C:\Program Files\Python314\python.exe" d:/Python/demo1
Enter the number of terms in the Fibonacci series: 10
Fibonacci series up to 10 terms:
0 1 1 2 3 5 8 13 21 34
PS D:\Python>
```

Justification:

Optimized code has removed those redundant variables that were not useful in the logic.

The corrected version does not have runtime errors due to uninitialized variables.

Code efficiency is maintained as it has a linear time complexity $O(n)$.

Task 3: Fibonacci Using Functions (Modular)

Prompt used:

write a code for a Fibonacci series up to n terms using functions

code:

```
def fibonacci_series(n):
    a, b = 0, 1
    series = []
    for _ in range(n):
        series.append(a)
        a, b = b, a + b
    return series
n = int(input("Enter the number of terms in the Fibonacci series: "))
if n <= 0:
    print("Please enter a positive integer")
else:
    print("Fibonacci series up to", n, "terms:")
    print(fibonacci_series(n))
```

Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS D:\Python> & "C:\Program Files\Python314\python.exe" d:/Python/demo1
Enter the number of terms in the Fibonacci series: 10
Fibonacci series up to 10 terms:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS D:\Python>
```

Justification:

The Fibonacci logic is coded inside the user-defined function.

It now returns a function that generates the Fibonacci series. This increases reusability.

The modular structure of this code will make debugging and testing simpler.

AI-assisted comments and structure improve understanding.

Task 4: Procedural vs Modular Comparison

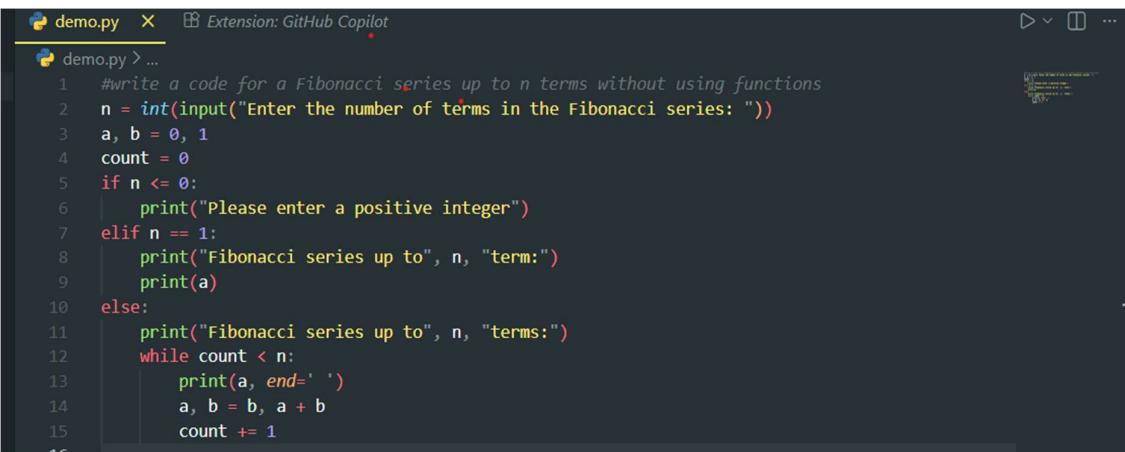
Prompt used:

write a code for a Fibonacci series up to n terms without using functions

write a code for a Fibonacci series up to n terms with using functions

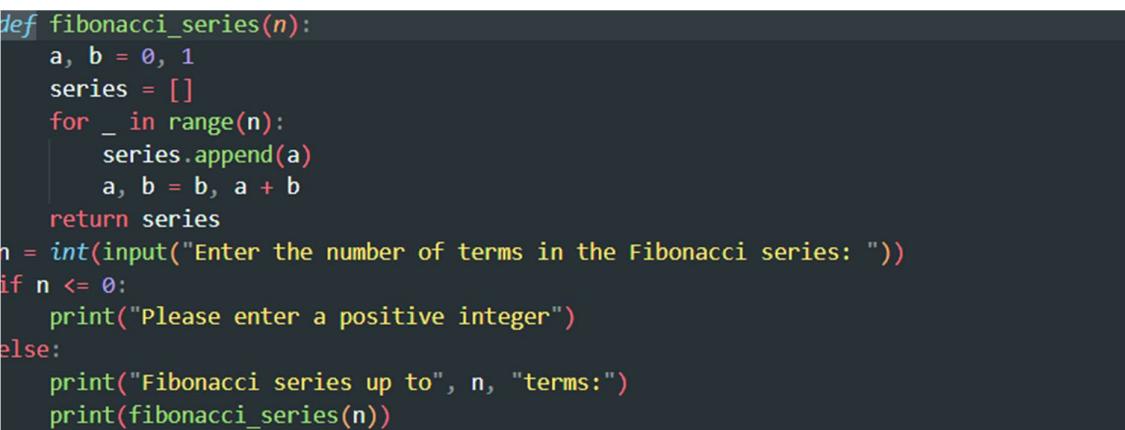
code:

Procedural:



```
demo.py  X  Extension: GitHub Copilot ...  
demo.py > ...  
1 #write a code for a Fibonacci series up to n terms without using functions  
2 n = int(input("Enter the number of terms in the Fibonacci series: "))  
3 a, b = 0, 1  
4 count = 0  
5 if n <= 0:  
6     print("Please enter a positive integer")  
7 elif n == 1:  
8     print("Fibonacci series up to", n, "term:")  
9     print(a)  
10 else:  
11     print("Fibonacci series up to", n, "terms:")  
12     while count < n:  
13         print(a, end=' ')  
14         a, b = b, a + b  
15         count += 1  
16
```

Modular:



```
def fibonacci_series(n):  
    a, b = 0, 1  
    series = []  
    for _ in range(n):  
        series.append(a)  
        a, b = b, a + b  
    return series  
n = int(input("Enter the number of terms in the Fibonacci series: "))  
if n <= 0:  
    print("Please enter a positive integer")  
else:  
    print("Fibonacci series up to", n, "terms:")  
    print(fibonacci_series(n))
```

Output:

Procedural:

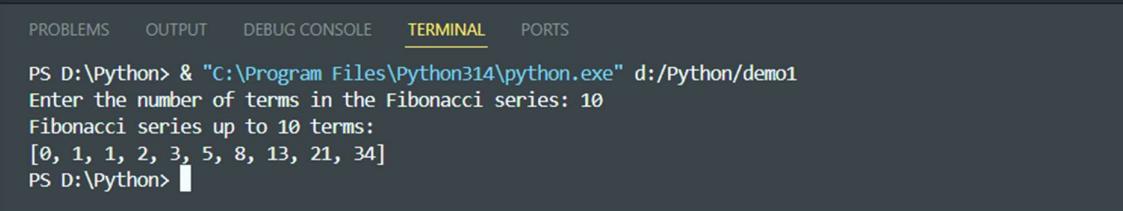


The screenshot shows a terminal window with the following output:

```
PS D:\Python> & "C:\Program Files\Python314\python.exe" d:/Python/demo.py
Enter the number of terms in the Fibonacci series: 10
Fibonacci series up to 10 terms:
0 1 1 2 3 5 8 13 21 34
PS D:\Python>
```

The terminal interface includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is selected), and PORTS. On the right side, there are icons for cmd and powershell.

Modular:



The screenshot shows a terminal window with the following output:

```
PS D:\Python> & "C:\Program Files\Python314\python.exe" d:/Python/demo1
Enter the number of terms in the Fibonacci series: 10
Fibonacci series up to 10 terms:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS D:\Python>
```

Justification:

It is easy to understand but is not reusable.

Modular code makes the code easier to follow because the code's logic is separated into functions.

Modular programs provide easier debugging.

Modular design is more apt for bigger applications.

Task 5: Iterative vs Recursive Fibonacci

Prompt used:

Generate iterative and recursive Fibonacci implementations

Code:

```
demo.py > ...
42     def fibonacci_iterative(n):
43         a, b = 0, 1
44         series = []
45         for _ in range(n):
46             series.append(a)
47             a, b = b, a + b
48         return series
49 n = int(input("Enter number of terms: "))
50
51 print("Iterative Fibonacci Series:")
52 print(fibonacci_iterative(n))
53 def fibonacci_recursive(n):
54     if n <= 0:
55         return []
56     elif n == 1:
57         return [0]
58     elif n == 2:
59         return [0, 1]
60     else:
61         series = fibonacci_recursive(n - 1)
62         series.append(series[-1] + series[-2])
63     return series
64 n = int(input("Enter number of terms: "))
65
66 print("Iterative Fibonacci Series:")
67 print(fibonacci_recursive(n))
```

Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS +
PS D:\Python> & "C:\Program Files\Python314\python.exe" d:/Python/demo1
Enter number of terms: 10
Iterative Fibonacci Series:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Enter number of terms: 10
Iterative Fibonacci Series:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
PS D:\Python> []
```

Justification:

The iterative approach uses loops and is memory efficient

In the recursive solution, a divide-and-conquer paradigm is followed.

For larger input values, the Iterative implementation is better.

A recursive approach requires more processing time because of its nature to call itself many times