

Y.Pranith Sai

2303A54072 B-48

## Lab 1

### Assignment Number 1.3

#### TASK:1

PROMPT: Write a Python program that asks the user for a number and prints the Fibonacci sequence up to that number, without using any functions

```
1 import sys
2
3 try:
4     n = int(input("Enter a non-negative integer: "))
5 except ValueError:
6     print("Invalid input. Please enter an integer.")
7     sys.exit(1)
8
9 if n < 0:
10     print("Please enter a non-negative integer.")
11     sys.exit(1)
12
13 # Generate and print Fibonacci numbers up to n (inclusive)
14 a = 0
15 b = 1
16
17 first_printed = False
18 if n >= 0:
19     print(a, end=" ")
20     first_printed = True
21
22 if n >= 1:
23     if first_printed:
24         print(" ", end=" ")
25     print(b, end=" ")
26     a, b = b, a + b
27
28 while True:
```

PS C:\Users\manis\OneDrive\Desktop\AI-Assisted-coding> & c:\Users\manis\AppData\Local\Microsoft\WindowsApps\python3.10.exe c:/Users/manis/OneDrive/Desktop/AI-Assisted-Coding/Assignment1.py  
Enter a non-negative integer: 4  
0 1 2 3  
PS C:\Users\manis\OneDrive\Desktop\AI-Assisted-Coding>

CHAT

WRITE A PYTHON PROGRAM THAT ASKS ...

- ✓ Optimizing tool selection...
- ✓ Planned, updated, created, executed, applied updates
- I'll add a short TODO plan, then open and update [Assignment1.py](#).
- ✓ Managed todo items and implemented Fibonacci sequence
- Now I'll write the requested program into [Assignment1.py](#).
- ✓ Called `apply_patch` with explanation parameter
- ✓ Edited [Assignment1.py](#) +38,-1
- ✓ Running the script verification
- Now I'll run [Assignment1.py](#) with sample input 10 to check output.

Run `push` command?

```
echo 10 | python "c:/Users/manis/OneDrive/Desktop/AI-Assisted-Coding/Assignment1.py"
```

[Allow](#) [Skip](#)

Create Fibonacci script (1/3)

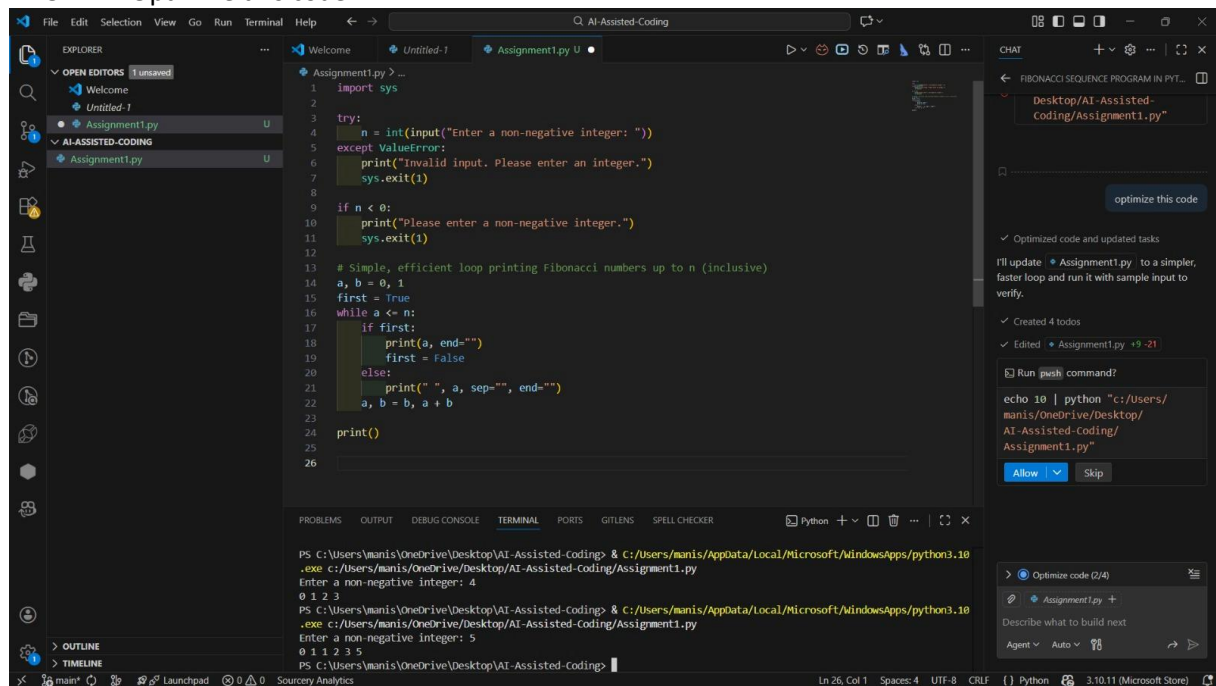
- [Assignment1.py](#) +

Describe what to build next

Agent [Auto](#) [10](#) [➔](#)

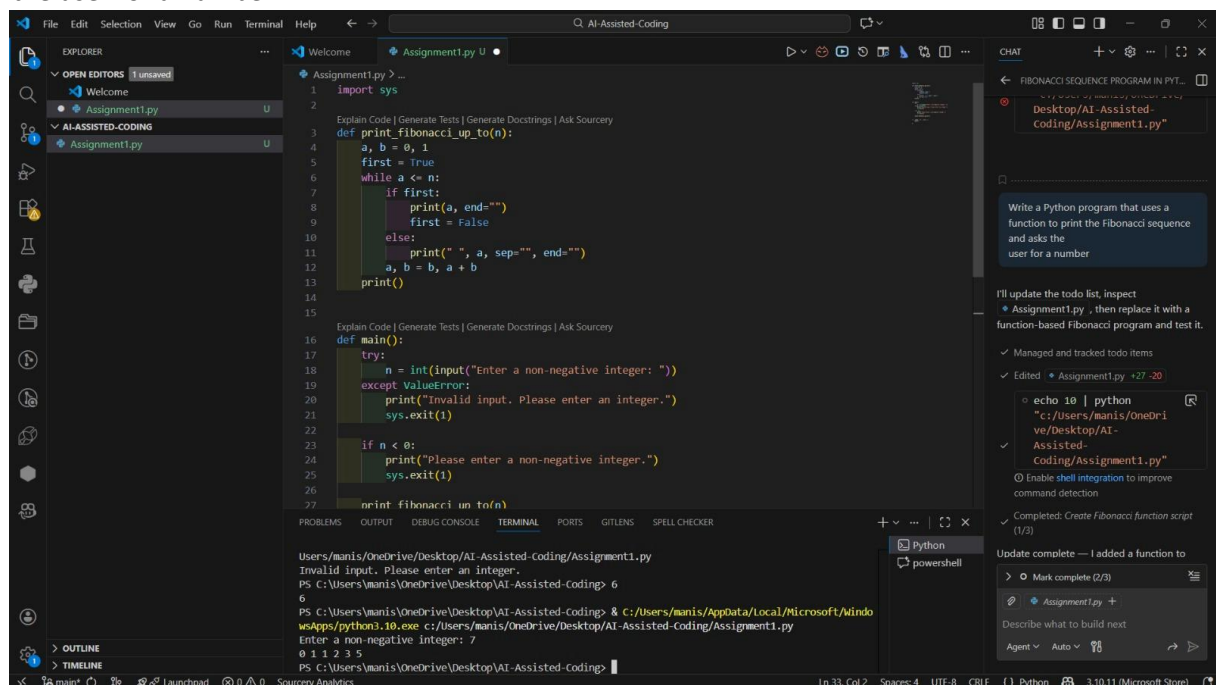
## TASK:2

PROMPT: Optimize this code



## TASK:3

PROMPT: Generate a Python program that uses a function to print the Fibonacci sequence and asks the user for a number.



### TASK:4

PROMPT: Print Fibonacci code with functions and without functions. Analyze code clarity, reusability, debugging ease, and suitability for large systems.

The screenshot displays a Windows 10 desktop environment with the Visual Studio Code (VS Code) editor open. The editor window shows a Python script named `task4.py` with the following content:

```

1  """
2  Comparative Analysis: Procedural vs Modular Fibonacci Implementation
3  """
4  =====
5  """
6
7  # COMPARISON TABLE
8  """
9  =====
10
11 comparison_data = {
12     "Criteria": [
13         "Code Clarity",
14         "Reusability",
15         "Debugging Ease",
16         "Maintainability",
17         "Scalability",
18         "Testing",
19         "Documentation",
20         "Performance Overhead"
21     ],
22     "Procedural (without Functions)": [
23         "Medium - Sequential logic, harder to follow",
24         "Low - Cannot reuse code easily",
25         "Medium - Errors mixed with main logic",
26         "Low - Changes require modifying entire block",
27         "Poor - Difficult to extend functionality",
28         "Hard - Cannot isolate test cases",
29         "Implicit - Logic embedded in code",
30         "None - Direct execution"
31     ],
32     "Modular (with Functions)": [
33         "High - Clear function names explain intent",
34         "High - Functions can be called multiple times",
35     ]
36 }

```

The Explorer sidebar on the left shows a file structure with a folder named `AIAC` and several files: `first.py`, `task1.py`, `task2.py`, `task3.py`, and `task4.py`. The bottom status bar indicates the current line and column as `Ln 122, Col 27`, the file encoding as `UTF-8`, and the date and time as `3:13:17 PM 07-01-2006`.

The screenshot displays the Visual Studio Code editor with a Python file named `task4.py` open. The file contains a function `fibonacci` and an analysis report.

```

31 def fibonacci(n):
32     """Modular (With Functions)": [
33         "High - Clear function names explain intent",
34         "High - Functions can be called multiple times",
35         "High - Isolated functions easier to debug",
36         "High - Update single function vs entire code",
37         "Excellent - Easy to add new features",
38         "Easy - Test individual functions separately",
39         "Explicit - Docstrings explain behavior",
40         "Minimal - Function call overhead"
41     ]
42 }
43
44 # =====
45 # ANALYSIS REPORT
46 # =====
47
48 analysis_report = """
49 EXECUTIVE SUMMARY
50 -----
51 The modular approach with functions is superior for professional development,
52 while procedural code may be acceptable for simple one-time scripts.
53
54 DETAILED ANALYSIS
55 -----
56
57 1. CODE CLARITY ***** (Modular wins)
58    - Procedural: Logic scattered throughout
59    - Modular: Self-documenting function names (fibonacci_recursive, fibonacci_iterative)
60
61 2. REUSABILITY ***** (Modular wins)
62    - Procedural: Copy-paste entire code block
63    - Modular: Import and call function anywhere
64
65 """

```

The interface includes a sidebar on the left with views for Explorer, Outline, and Timeline. The bottom status bar shows the file is encoded in UTF-8, uses the Python language, and has 122 columns and 27 lines. The system tray at the bottom shows the date and time as 07-01-2026, 21:22.

The screenshot shows a VS Code editor with a Python file named `task4.py` open. The file contains a detailed analysis of modular vs. procedural code. The script is structured as follows:

```

67
68 4. MAINTAINABILITY ***** (Modular Wins)
69   - Procedural: bug fix requires modifying one monolithic block
70   - Modular: Fix single function in isolation
71
72 5. SCALABILITY ***** (Modular Wins)
73   - Procedural: Adding modularization requires rewriting
74   - Modular: Create new function, keep old ones for comparison
75
76 6. TESTING ***** (Modular Wins)
77   - Procedural: Cannot write unit tests
78   - Modular: Easy pytest/unittest coverage
79
80 RECOMMENDATION
81 -----
82 Use MODULAR approach for:
83   ✓ Production code
84   ✓ Team projects
85   ✓ Code that will be maintained
86   ✓ Any system with >100 lines
87
88 Use PROCEDURAL approach for:
89   ✓ Quick scripts
90   ✓ One-time analysis
91   ✓ Learning/exploration
92   ✓ Interactive notebooks (Jupyter)
93
94 CONCLUSION
95 -----
96 Modular code requires slightly more initial effort but pays dividends
97 in maintainability, testability, and scalability. Best practice:
98 Always prefer functions unless writing disposable code.
99 ---

```

The interface includes a file explorer on the left showing the project structure, a terminal at the bottom, and a status bar at the very bottom indicating the current file is `Ln 122, Col 27` in `task4.py`.

The screenshot shows a Visual Studio Code editor with a file explorer on the left containing files task1.py through task5.py. The main editor window displays task4.py, which contains a script for a comparative analysis of Fibonacci implementations. The script prints a table header and rows comparing 'Procedural' and 'Modular' approaches across various criteria. The terminal at the bottom shows the output of the script, which is a table with three columns: Criteria, Procedural, and Modular. The table lists criteria such as Code Clarity, Reusability, Debugging Ease, Maintainability, Scalability, Testing, Documentation, and Performance Overhead, comparing the two approaches.

```
105 if __name__ == "__main__":
106     print("-" * 80)
107     print("COMPARATIVE ANALYSIS: FIBONACCI IMPLEMENTATIONS")
108     print("-" * 80)
109     print()
110
111     # Print table header
112     print(f"{'Criteria':<25} | {'Procedural':<30} | {'Modular':<30}")
113     print("-" * 88)
114
115     # Print table rows
116     for i in range(len(comparison_data["Criteria"])):
117         print(f"{'Criteria':<25} | "
118               f"{'Procedural (Without Functions)':<30} | "
119               f"{'Modular (With Functions)':<30}")
120
121     print()
122     print(analysis_report)
```

Criteria	Procedural	Modular
Code Clarity	Medium - Sequential logic, harder to follow	High - Clear function names explain intent
Reusability	Low - Cannot reuse code easily	High - Functions can be called multiple times
Debugging Ease	Medium - Errors mixed with main logic	High - Isolated functions easier to debug
Maintainability	Low - Changes require modifying entire block	High - Update single function vs entire code
Scalability	Poor - Difficult to extend functionality	Excellent - Easy to add new features
Testing	Hard - Cannot isolate test cases	Easy - Test individual functions separately
Documentation	Implicit - Logic embedded in code	Explicit - Docstrings explain behavior
Performance Overhead	None - Direct execution	Minimal - Function call overhead

## TASK:5

PROMPT: Generate an iterative Fibonacci implementation, recursive Fibonacci implementation compare their time complexity, performance for large n

The screenshot shows a Visual Studio Code editor with a file explorer on the left containing files task1.py through task5.py. The main editor window displays task5.py, which contains three Python functions for calculating Fibonacci numbers: an iterative version, a recursive version, and a memoized recursive version. The terminal at the bottom shows the output of the script, which is a table with three columns: Criteria, Procedural, and Modular. The table lists criteria such as Code Clarity, Reusability, Debugging Ease, Maintainability, Scalability, Testing, Documentation, and Performance Overhead, comparing the two approaches.

```
1 import time
2 # ITERATIVE FIBONACCI
3 def fibonacci_iterative(n: int) -> int:
4     if n <= 0:
5         return 0
6     elif n == 1:
7         return 1
8     a, b = 0, 1
9     for _ in range(2, n + 1):
10         a, b = b, a + b
11     return b
12
13 # RECURSIVE FIBONACCI
14 def fibonacci_recursive(n: int) -> int:
15     if n <= 0:
16         return 0
17     elif n == 1:
18         return 1
19     return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
20
21 # OPTIMIZED RECURSIVE (with memoization)
22 def fibonacci_memoized(n: int, memo: dict = None) -> int:
23     if memo is None:
24         memo = {}
25     if n in memo:
26         return memo[n]
```

Criteria	Procedural	Modular
Code Clarity	Medium - Sequential logic, harder to follow	High - Clear function names explain intent
Reusability	Low - Cannot reuse code easily	High - Functions can be called multiple times
Debugging Ease	Medium - Errors mixed with main logic	High - Isolated functions easier to debug
Maintainability	Low - Changes require modifying entire block	High - Update single function vs entire code
Scalability	Poor - Difficult to extend functionality	Excellent - Easy to add new features
Testing	Hard - Cannot isolate test cases	Easy - Test individual functions separately
Documentation	Implicit - Logic embedded in code	Explicit - Docstrings explain behavior
Performance Overhead	None - Direct execution	Minimal - Function call overhead

VS Code interface showing a Python file named `task5.py` with a Fibonacci memoization implementation. The code includes a `fibonacci_memoized` function and a `main` function that compares iterative and recursive methods for `n=35`.

```
20 def fibonacci_memoized(n: int, memo: dict = None) -> int:
21     return memo[n]
22 # COMPARISON & TESTING
23 if __name__ == "__main__":
24     n = 35
25     # Iterative Test
26     start = time.time()
27     result_iter = fibonacci_iterative(n)
28     time_iter = time.time() - start
29     print(f"Iterative (n={n}): {result_iter} | Time: {time_iter:.6f}s")
30     # Recursive Test (safe limit)
31     if n <= 30:
32         start = time.time()
33         result_rec = fibonacci_recursive(n)
34         time_rec = time.time() - start
35         print(f"Recursive (n={n}): {result_rec} | Time: {time_rec:.6f}s")
36     else:
37         print(f"Recursive: Skipped (too slow for n={n})")
38     # Memoized Test
39     start = time.time()
40     result_memo = fibonacci_memoized(n)
41     time_memo = time.time() - start
42     print(f"Memoized (n={n}): {result_memo} | Time: {time_memo:.6f}s")
```

The bottom panel displays a comparison table between Procedural and Modular programming criteria.

Criteria	Procedural	Modular
Code Clarity	Medium - Sequential logic, harder to follow	High - Clear function names explain intent
Reusability	Low - Cannot reuse code easily	High - Functions can be called multiple times
Debugging Ease	Medium - Errors mixed with main logic	High - Isolated functions easier to debug
Maintainability	Low - Changes require modifying entire block	High - Update single function vs entire code
Scalability	Poor - Difficult to extend functionality	Excellent - Easy to add new features
Testing	Hard - Cannot isolate test cases	Easy - Test individual functions separately
Documentation	Implicit - Logic embedded in code	Explicit - Docstrings explain behavior
Performance Overhead	None - Direct execution	Minimal - Function call overhead

The status bar at the bottom shows the file is at Line 46, Column 58, with 4 spaces, UTF-8 encoding, and CRLF line endings. The system tray at the very bottom indicates a temperature of 20°C and the date 09-01-2026.