

G.Tejomayi

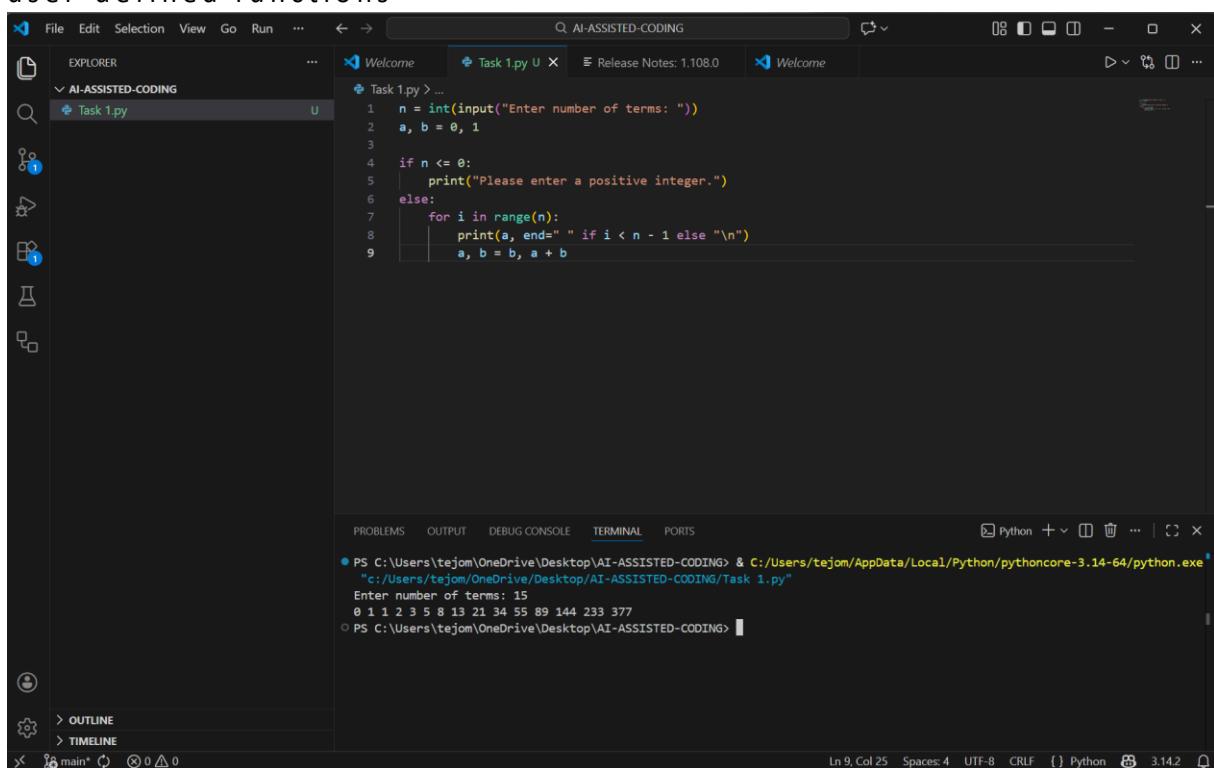
2303A54064 B-47B

Lab 1

Assignment Number 1.3

TASK:1

PROMPT: Write a Python program that takes user input n and prints the Fibonacci sequence up to n terms using main code only without any user-defined functions



The screenshot shows a Microsoft Visual Studio Code (VS Code) interface. The left sidebar has a tree view with 'EXPLORER' expanded, showing 'AI-ASSISTED-CODING' and 'Task 1.py'. The main editor area contains the following Python code:

```
n = int(input("Enter number of terms: "))
a, b = 0, 1
if n <= 0:
    print("Please enter a positive integer.")
else:
    for i in range(n):
        print(a, end=" " if i < n - 1 else "\n")
        a, b = b, a + b
```

The terminal at the bottom shows the output of running the script:

```
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING> & C:/Users/tejom/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/tejom/OneDrive/Desktop/AI-ASSISTED-CODING/Task 1.py"
Enter number of terms: 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

TASK:2

PROMPT: Optimize this Fibonacci code by removing redundant variables, simplifying loop logic, avoiding unnecessary computations, and improving readability while keeping it function-free

The screenshot shows the VS Code interface with the 'AI-ASSISTED-CODING' extension active. The Explorer sidebar shows files 'Task 1.py', 'task-2', and 'task-2'. The 'task-2' editor tab contains the following Python code:

```
# - Removing redundant variables
# - Simplifying loop logic
# - Avoiding unnecessary computations
# - Improving readability
```

The terminal below shows the output of running the code:

```
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING> & C:/Users/tejom/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/tejom/OneDrive/Desktop/AI-ASSISTED-CODING/Task 1.py"
Enter number of terms: 15
0 1 2 3 5 8 13 21 34 55 89 144 233 377
```

The status bar at the bottom indicates the code is 3.14.2.

TASK:3

PROMPT: Write a Python program that uses a user-defined function to generate and return the Fibonacci sequence up to n terms with clear and meaningful comments

The screenshot shows the VS Code interface with the 'AI-ASSISTED-CODING' extension active. The Explorer sidebar shows files 'Task 1.py', 'task-2.py', and 'task-3.py'. The 'task-3.py' editor tab contains the following Python code:

```
# Function to generate Fibonacci sequence up to n terms
def generate_fibonacci(n):
    """
    Generate Fibonacci sequence up to n terms.

    Args:
        n: Number of Fibonacci terms to generate

    Returns:
        List containing the Fibonacci sequence
    """
    # Initialize list to store Fibonacci numbers
    fib_sequence = []

    # Handle edge case: if n is 0 or negative
    if n <= 0:
        return fib_sequence

    # Initialize first two Fibonacci numbers
    a, b = 0, 1

    # Generate n Fibonacci numbers
    for i in range(n):
```

The terminal below shows the output of running the code:

```
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING> & C:/Users/tejom/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/tejom/OneDrive/Desktop/AI-ASSISTED-CODING/task-2.py"
Enter the number of terms: 26
0 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025
PS C:\Users\tejom\OneDrive\Desktop\AI-ASSISTED-CODING> & C:/Users/tejom/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/tejom/OneDrive/Desktop/AI-ASSISTED-CODING/task 3.py"
Enter the number of Fibonacci terms: 21
Fibonacci sequence (21 terms): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

The status bar at the bottom indicates the code is 3.14.2.

TASK:4

PROMPT: Create a comparative analysis between procedural Fibonacci code without functions and modular Fibonacci code with functions focusing on code clarity, reusability, debugging ease, and suitability for large systems

```

File Edit Selection View Go Run Terminal Help ⌘ Aiac
EXPLORER ... task1.py task2.py task3.py task4.py
task4.py > ...
1 """
2 Comparative Analysis: Procedural vs Modular Fibonacci Implementation
3 -----
4 """
5
6 # -----
7 # COMPARISON TABLE
8 # -----
9
10 comparison_data = {
11     "criteria": [
12         "Code Clarity",
13         "Reusability",
14         "Debugging Ease",
15         "Maintainability",
16         "Scalability",
17         "Testing",
18         "Documentation",
19         "Performance Overhead"
20     ],
21     "Procedural (Without Functions)": [
22         "Medium - Sequential logic, harder to follow",
23         "Low - Cannot reuse code easily",
24         "Medium - Embedded with main logic",
25         "Low - Changes require modifying entire block",
26         "Poor - Difficult to extend functionality",
27         "Hard - Cannot isolate test cases",
28         "Implicit - logic embedded in code",
29         "None - Direct execution"
30     ],
31     "Modular (With Functions)": [
32         "High - Clear function names explain intent",
33         "High - Functions can be called multiple times",
34     ]
35 }
36
37 # -----
38 # ANALYSIS REPORT
39 # -----
40
41 analysis_report = """
42 EXECUTIVE SUMMARY
43 -----
44 The modular approach with functions is superior for professional development,
45 while procedural code may be acceptable for simple one-time scripts.
46
47 DETAILED ANALYSIS
48 -----
49
50 1. CODE CLARITY ***** (Modular Wins)
51     - Procedural: Logic scattered throughout
52     - Modular: Self-documenting function names (fibonacci_recursive, fibonacci_iterative)
53
54 2. REUSABILITY ***** (Modular Wins)
55     - Procedural: Copy-paste entire code block
56     - Modular: Import and call function anywhere
57
58 3. MAINTAINABILITY ***** (Modular Wins)
59     - Procedural: Bug fix requires modifying one monolithic block
60     - Modular: Fix single function in isolation
61
62 4. SCALABILITY ***** (Modular Wins)
63     - Procedural: Adding memoization requires rewriting
64     - Modular: Create new function, keep old ones for comparison
65
66 5. TESTING ***** (Modular Wins)
67     - Procedural: Cannot write unit tests
68     - Modular: Easy pytest/unittest coverage
69
70 6. CONCLUSION
71 -----
72 Modular code requires slightly more initial effort but pays dividends
73 in maintainability, testability, and scalability. Best practice:
74 Always prefer functions unless writing disposable code.
75 """
76
77 # -----
78 # RECOMMENDATION
79 # -----
80
81 Use MODULAR approach for:
82     ✓ Production code
83     ✓ Team projects
84     ✓ Code that will be maintained
85     ✓ Any system with >1000 lines
86
87 Use PROCEDURAL approach for:
88     ✓ Quick scripts
89     ✓ One-time analysis
90     ✓ Learning/exploration
91     ✓ Interactive notebooks (Jupyter)
92
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 """

```

```

File Edit Selection View Go Run Terminal Help ⌘ Aiac
EXPLORER ... task1.py task2.py task3.py task4.py
task4.py > ...
1 """
2 Comparative Analysis: Procedural vs Modular Fibonacci Implementation
3 -----
4 """
5
6 # -----
7 # COMPARISON TABLE
8 # -----
9
10 comparison_data = {
11     "criteria": [
12         "Code Clarity",
13         "Reusability",
14         "Debugging Ease",
15         "Maintainability",
16         "Scalability",
17         "Testing",
18         "Documentation",
19         "Performance Overhead"
20     ],
21     "Procedural (Without Functions)": [
22         "Medium - Sequential logic, harder to follow",
23         "Low - Cannot reuse code easily",
24         "Medium - Embedded with main logic",
25         "Low - Changes require modifying entire block",
26         "Poor - Difficult to extend functionality",
27         "Hard - Cannot isolate test cases",
28         "Implicit - logic embedded in code",
29         "None - Direct execution"
30     ],
31     "Modular (With Functions)": [
32         "High - Clear function names explain intent",
33         "High - Functions can be called multiple times",
34     ]
35 }
36
37 # -----
38 # ANALYSIS REPORT
39 # -----
40
41 analysis_report = """
42 EXECUTIVE SUMMARY
43 -----
44 The modular approach with functions is superior for professional development,
45 while procedural code may be acceptable for simple one-time scripts.
46
47 DETAILED ANALYSIS
48 -----
49
50 1. CODE CLARITY ***** (Modular Wins)
51     - Procedural: Logic scattered throughout
52     - Modular: Self-documenting function names (fibonacci_recursive, fibonacci_iterative)
53
54 2. REUSABILITY ***** (Modular Wins)
55     - Procedural: Copy-paste entire code block
56     - Modular: Import and call function anywhere
57
58 3. MAINTAINABILITY ***** (Modular Wins)
59     - Procedural: Bug fix requires modifying one monolithic block
60     - Modular: Fix single function in isolation
61
62 4. SCALABILITY ***** (Modular Wins)
63     - Procedural: Adding memoization requires rewriting
64     - Modular: Create new function, keep old ones for comparison
65
66 5. TESTING ***** (Modular Wins)
67     - Procedural: Cannot write unit tests
68     - Modular: Easy pytest/unittest coverage
69
70 6. CONCLUSION
71 -----
72 Modular code requires slightly more initial effort but pays dividends
73 in maintainability, testability, and scalability. Best practice:
74 Always prefer functions unless writing disposable code.
75 """
76
77 # -----
78 # RECOMMENDATION
79 # -----
80
81 Use MODULAR approach for:
82     ✓ Production code
83     ✓ Team projects
84     ✓ Code that will be maintained
85     ✓ Any system with >1000 lines
86
87 Use PROCEDURAL approach for:
88     ✓ Quick scripts
89     ✓ One-time analysis
90     ✓ Learning/exploration
91     ✓ Interactive notebooks (Jupyter)
92
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 """

```

```

File Edit Selection View Go Run Terminal Help ⌘ Aiac
EXPLORER ... task1.py task2.py task3.py task4.py
task4.py > ...
1 """
2 Comparative Analysis: Procedural vs Modular Fibonacci Implementation
3 -----
4 """
5
6 # -----
7 # COMPARISON TABLE
8 # -----
9
10 comparison_data = {
11     "criteria": [
12         "Code Clarity",
13         "Reusability",
14         "Debugging Ease",
15         "Maintainability",
16         "Scalability",
17         "Testing",
18         "Documentation",
19         "Performance Overhead"
20     ],
21     "Procedural (Without Functions)": [
22         "Medium - Sequential logic, harder to follow",
23         "Low - Cannot reuse code easily",
24         "Medium - Embedded with main logic",
25         "Low - Changes require modifying entire block",
26         "Poor - Difficult to extend functionality",
27         "Hard - Cannot isolate test cases",
28         "Implicit - logic embedded in code",
29         "None - Direct execution"
30     ],
31     "Modular (With Functions)": [
32         "High - Clear function names explain intent",
33         "High - Functions can be called multiple times",
34     ]
35 }
36
37 # -----
38 # ANALYSIS REPORT
39 # -----
40
41 analysis_report = """
42 EXECUTIVE SUMMARY
43 -----
44 The modular approach with functions is superior for professional development,
45 while procedural code may be acceptable for simple one-time scripts.
46
47 DETAILED ANALYSIS
48 -----
49
50 1. CODE CLARITY ***** (Modular Wins)
51     - Procedural: Logic scattered throughout
52     - Modular: Self-documenting function names (fibonacci_recursive, fibonacci_iterative)
53
54 2. REUSABILITY ***** (Modular Wins)
55     - Procedural: Copy-paste entire code block
56     - Modular: Import and call function anywhere
57
58 3. MAINTAINABILITY ***** (Modular Wins)
59     - Procedural: Bug fix requires modifying one monolithic block
60     - Modular: Fix single function in isolation
61
62 4. SCALABILITY ***** (Modular Wins)
63     - Procedural: Adding memoization requires rewriting
64     - Modular: Create new function, keep old ones for comparison
65
66 5. TESTING ***** (Modular Wins)
67     - Procedural: Cannot write unit tests
68     - Modular: Easy pytest/unittest coverage
69
70 6. CONCLUSION
71 -----
72 Modular code requires slightly more initial effort but pays dividends
73 in maintainability, testability, and scalability. Best practice:
74 Always prefer functions unless writing disposable code.
75 """
76
77 # -----
78 # RECOMMENDATION
79 # -----
80
81 Use MODULAR approach for:
82     ✓ Production code
83     ✓ Team projects
84     ✓ Code that will be maintained
85     ✓ Any system with >1000 lines
86
87 Use PROCEDURAL approach for:
88     ✓ Quick scripts
89     ✓ One-time analysis
90     ✓ Learning/exploration
91     ✓ Interactive notebooks (Jupyter)
92
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 screenshot shows a Microsoft Visual Studio Code (VS Code) interface. The top navigation bar includes File, Edm, Selection, View, Go, Run, Terminal, Help, and a search bar labeled AIAC. The left sidebar has sections for Explorer, AIAC, and Task Manager. The main editor area displays Python code for generating a comparison table of Fibonacci implementations. Below the editor are tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. A terminal window titled 'COMPARATIVE ANALYSIS: FIBONACCI IMPLEMENTATIONS' shows the output of the script. To the right of the terminal is a table comparing different implementation criteria. The bottom status bar shows file statistics (Ln 122, Col 27), workspace details (Spaces: 3, UTF-8, CRLF), and a Go Live button. A bottom navigation bar includes icons for Home, Search, File Explorer, and Task Manager.

```
task1.py task2.py task3.py task4.py task5.py
task4.py ...
105 if __name__ == "__main__":
106     print("=" * 88)
107     print("COMPARATIVE ANALYSIS: FIBONACCI IMPLEMENTATIONS")
108     print("=" * 88)
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"{comparison_data['Criteria'][i]:<25} | "
118               f"{comparison_data['Procedural (Without Functions)'][i]:<30} | "
119               f"{comparison_data['Modular (With Functions)'][i]:<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 both iterative and recursive Fibonacci implementations in Python with comments, explain their execution flow, and compare their time and space complexity and performance for large n

The screenshot shows a code editor interface with the following details:

- File Explorer:** Shows files task1.py, task2.py, task3.py, task4.py, and task5.py.
- Code Editor:** Displays five versions of a Fibonacci function:
 - task5.py (Iterative):** A for loop from 2 to n, updating variables a and b.
 - task5.py (Recursive):** A recursive call to fibonacci_recursive(n-1) + fibonacci_recursive(n-2).
 - task5.py (Optimized Recursive):** A recursive call with memoization using a dictionary.
 - task4.py (Memoized):** A recursive call with memoization using a dictionary.
 - task3.py (Procedural):** A procedural implementation with nested loops.
- Performance Table:** Compares Procedural and Modular approaches across various criteria.

Criteria	Procedural	Modular
Code Clarity	Medium - Sequential logic, harder to follow High - Clear function names explain intent	Low - Cannot reuse code easily High - Functions can be called multiple times
Reusability	Medium - Errors mixed with main logic High - Isolated functions easier to debug	Low - Changes require modifying entire block High - Update single function vs entire code
Debugging Ease	Poor - Difficult to extend functionality Excellent - Easy to add new features	Hard - Cannot isolate test cases Easy - Test individual functions separately
Maintainability	Implicit - Logic embedded in code Explicit - Docstrings explain behavior	Implicit - Logic embedded in code Explicit - Docstrings explain behavior
Scalability	None - Direct execution	Minimal - Function call overhead
Testing	None - Direct execution	Minimal - Function call overhead
Documentation	None - Direct execution	Minimal - Function call overhead
Performance Overhead	None - Direct execution	Minimal - Function call overhead
- Bottom Bar:** Includes icons for file operations, search, terminal, and ports, along with system status (CPU 20%, mostly cloudy), network, and system info (Windows 10, ENG, 3.13.7, 22:01).

The screenshot shows the Microsoft Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder named "AIAC" containing files: first.py, task1.py, task2.py, task3.py, task4.py, and task5.py.
- Code Editor:** The active file is task5.py, which contains Python code for calculating Fibonacci numbers using memoization, iterative methods, and recursive methods. It includes timing logic to compare execution times between these approaches.
- Terminal:** The terminal tab is visible at the bottom of the editor area.
- Output:** The output tab shows a table comparing different software development 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	

- Activity Bar:** Includes icons for Outline, Timeline, and Taskbar.
- Bottom Status Bar:** Displays weather (20°C, Mostly cloudy), system icons (Windows, battery, signal), and the date/time (09-01-2026).