

Y.Pranith Sai

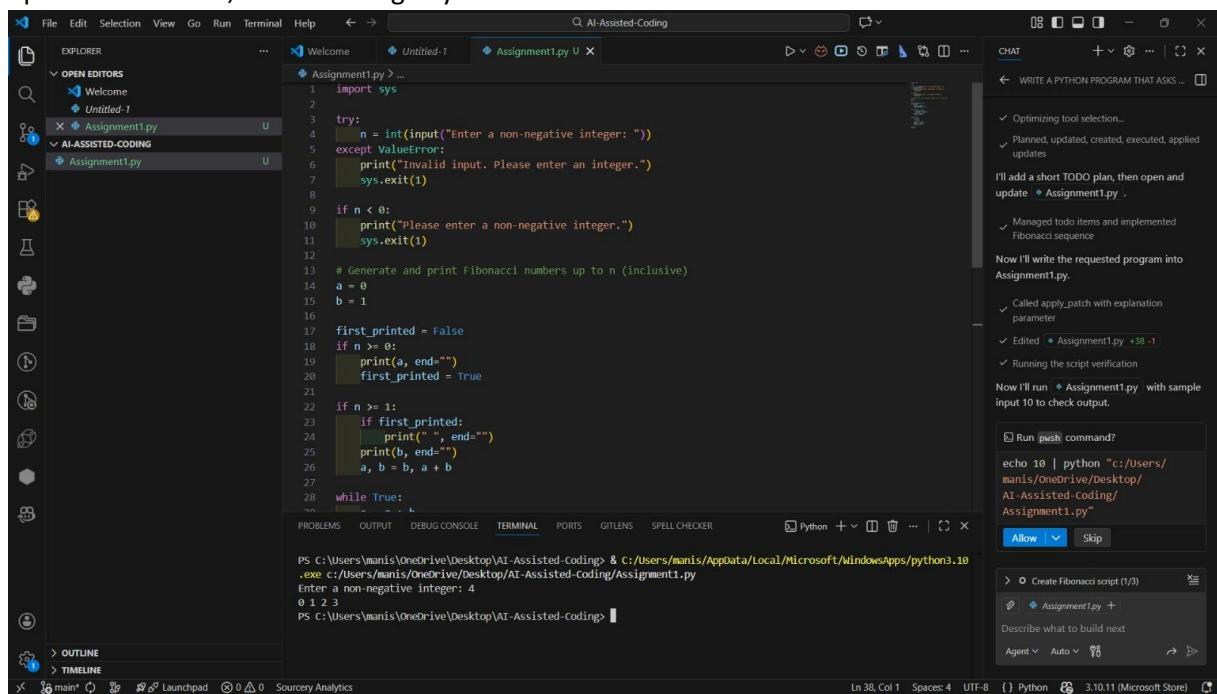
2303A54072 B-48

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



The screenshot shows the Microsoft Visual Studio Code interface. The left sidebar has 'OPEN EDITORS' expanded, showing 'Assignment1.py' and 'Untitled-1'. The main editor area contains the following Python code:

```
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:
29     -
```

The terminal below the code shows the command run and the output:

```
PS C:\Users\manis\OneDrive\Desktop\AI-Assisted-Coding> & c:/Users/manis/AppData/Local/Microsoft/Windows/apps/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>
```

The right sidebar shows AI-assisted coding suggestions and a todo list.

TASK:2

PROMPT: Optimize this code

The screenshot shows the Microsoft Visual Studio Code interface with the "AI-Assisted Coding" extension active. The main editor window contains the following Python code:

```
Assignment1.py > ...
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 # Simple, efficient loop printing Fibonacci numbers up to n (inclusive)
14 a, b = 0, 1
15 first = True
16 while a <= n:
17     if first:
18         print(a, end="")
19         first = False
20     else:
21         print(" ", a, sep="", end="")
22     a, b = b, a + b
23
24 print()
```

The right-hand sidebar displays the AI's response:

- CHAT: FIBONACCI SEQUENCE PROGRAM IN PY...
- Optimize this code
- Optimized code and updated tasks:
 - I'll update `Assignment1.py` to a simpler, faster loop and run it with sample input to verify.
- Created 4 todos
- Edited `Assignment1.py` +9 -21
- Run push command?
 - echo 10 | python "c:/Users/manis/OneDrive/Desktop/AI-Assisted-Coding/Assignment1.py"
- Allow Skip
- Optimize code (2/4)
- Describe what to build next
- Agent Auto

TASK:3

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

The screenshot shows the Microsoft Visual Studio Code interface with the "AI-Assisted Coding" extension active. The main editor window contains the following Python code:

```
Assignment1.py > ...
1 import sys
2
3 Explain Code | Generate Tests | Generate Docstrings | Ask Sourcey
4 def print_fibonacci_up_to(n):
5     a, b = 0, 1
6     first = True
7     while a <= n:
8         if first:
9             print(a, end="")
10            first = False
11        else:
12            print(" ", a, sep="", end="")
13        a, b = b, a + b
14
15 Explain Code | Generate Tests | Generate Docstrings | Ask Sourcey
16 def main():
17     try:
18         n = int(input("Enter a non-negative integer: "))
19     except ValueError:
20         print("Invalid input. Please enter an integer.")
21         sys.exit(1)
22
23     if n < 0:
24         print("Please enter a non-negative integer.")
25         sys.exit(1)
26
27     print(fibonacci_up_to(n))
```

The right-hand sidebar displays the AI's response:

- CHAT: FIBONACCI SEQUENCE PROGRAM IN PY...
- Write a Python program that uses a function to print the Fibonacci sequence and asks the user for a number
- I'll update the todo list, inspect `Assignment1.py`, then replace it with a function-based Fibonacci program and test it.
- Managed and tracked todo items
- Edited `Assignment1.py` +27 -20
- Enable shell integration to improve command detection
- Completed: Create Fibonacci function script (1/3)
- Update complete — I added a function to
- Mark complete (2/3)
- Describe what to build next
- Agent Auto

TASK:4

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

```

File Edit Selection View Go Run Terminal Help ⏎ → Q AIAC
EXPLORER ... task1.py task2.py task3.py task4.py
AIAC
first.py
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 - Errors mixed 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 # ANALYSTS REPORT
39 # -----
40
41 }
42
43 # -----
44 # ANALYSTS REPORT
45 # -----
46
47 analysis_report = """
48 EXECUTIVE SUMMARY
49 -----
50 The modular approach with functions is superior for professional development,
51 while procedural code may be acceptable for simple one-time scripts.
52
53 DETAILED ANALYSIS
54 -----
55
56 1. CODE CLARITY ***** (Modular Wins)
57     - Procedural: Logic scattered throughout
58     - Modular: Self-documenting function names (fibonacci_recursive, fibonacci_iterative)
59
60 2. REUSABILITY ***** (Modular Wins)
61     - Procedural: Copy/paste entire code block
62     - Modular: Import and call function anywhere
63
64 # -----
65 # ANALYSTS REPORT
66 # -----
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 memoization 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

```

```

File Edit Selection View Go Run Terminal Help ⏎ → Q AIAC
EXPLORER ... task1.py task2.py task3.py task4.py
AIAC
first.py
task1.py
task2.py
task3.py
task4.py
task4.py > ...
31     "Modular (With Functions)": [
32         "High - clear function names explain intent",
33         "High - Functions can be called multiple times",
34         "High - Isolated functions easier to debug",
35         "High - Update single function vs entire code",
36         "Excellent - Easy to add and remove",
37         "Easy - Test individual functions separately",
38         "Explicit - Docstrings explain behavior",
39         "Minimal - Function call overhead"
40     ]
41 }
42
43 # -----
44 # ANALYSTS REPORT
45 # -----
46
47 analysis_report = """
48 EXECUTIVE SUMMARY
49 -----
50 The modular approach with functions is superior for professional development,
51 while procedural code may be acceptable for simple one-time scripts.
52
53 DETAILED ANALYSIS
54 -----
55
56 1. CODE CLARITY ***** (Modular Wins)
57     - Procedural: Logic scattered throughout
58     - Modular: Self-documenting function names (fibonacci_recursive, fibonacci_iterative)
59
60 2. REUSABILITY ***** (Modular Wins)
61     - Procedural: Copy/paste entire code block
62     - Modular: Import and call function anywhere
63
64 # -----
65 # ANALYSTS REPORT
66 # -----
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 memoization 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

```

```

File Edit Selection View Go Run Terminal Help ⏎ → Q AIAC
EXPLORER ... task1.py task2.py task3.py task4.py
AIAC
first.py
task1.py
task2.py
task3.py
task4.py
task4.py > ...
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 memoization 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 screenshot shows the VS Code interface with the file `task4.py` open. The code performs a comparative analysis of Fibonacci implementations across different criteria. A terminal window below displays the results of the analysis.

```

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"{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)

```

COMPARATIVE ANALYSIS: FIBONACCI IMPLEMENTATIONS

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 the VS Code interface with the file `task5.py` open. The code defines iterative, recursive, and optimized recursive (with memoization) implementations of the Fibonacci sequence. A terminal window below displays the results of the analysis.

```

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 # RECURSIVE FIBONACCI
13 def fibonacci_recursive(n: int) -> int:
14     if n <= 0:
15         return 0
16     elif n == 1:
17         return 1
18     return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
19 # OPTIMIZED RECURSIVE (with memoization)
20 def fibonacci_memoized(n: int, memo: dict = None) -> int:
21     if memo is None:
22         memo = {}
23     if n in memo:
24         return memo[n]
25     memo[n] = ...

```

COMPARATIVE ANALYSIS: FIBONACCI IMPLEMENTATIONS

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	

```
task5.py > ...
20     def fibonacci_memoized(n: int, memo: dict = None) -> int:
30         return memo[n]
31     # COMPARISON & TESTING
32     if __name__ == "__main__":
33         n = 35
34         # Iterative Test
35         start = time.time()
36         result_iter = fibonacci_iterative(n)
37         time_iter = time.time() - start
38         print(f"Iterative (n={n}): {result_iter} | Time: {time_iter:.6f}s")
39         # Recursive Test (safe limit)
40         if n <= 30:
41             start = time.time()
42             result_rec = fibonacci_recursive(n)
43             time_rec = time.time() - start
44             print(f"Recursive (n={n}): {result_rec} | Time: {time_rec:.6f}s")
45         else:
46             print(f"Recursive: Skipped (too slow for n={n})")
47     # Memoized Test
48     start = time.time()
49     result_memo = fibonacci_memoized(n)
50     time_memo = time.time() - start
51     print(f"Memoized (n={n}): {result_memo} | Time: {time_memo:.6f}s")
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

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	

Python Python Python powershell powershell Python powershell

LN 46, Col 58 Spaces: 4 UTF-8 CRLF Python ENG IN 22:17 09-01-2026