Y.Pranith Sai

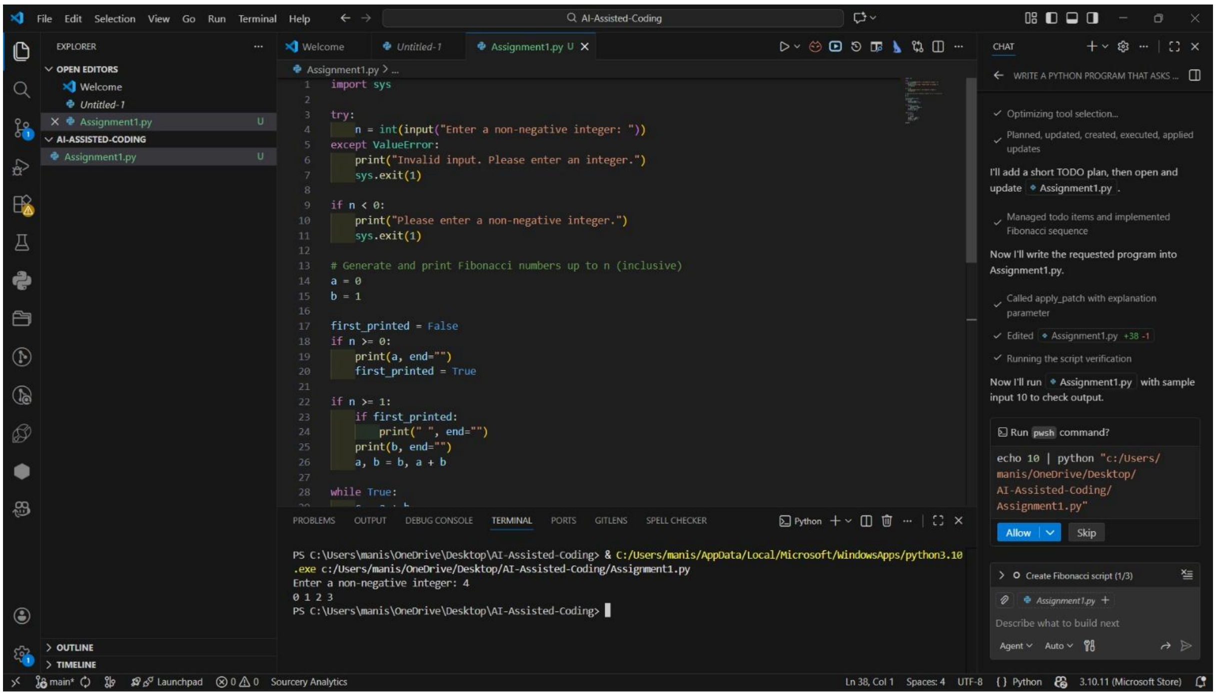
2303A54072 B-48

Lab 1

Assignment Number 1.3

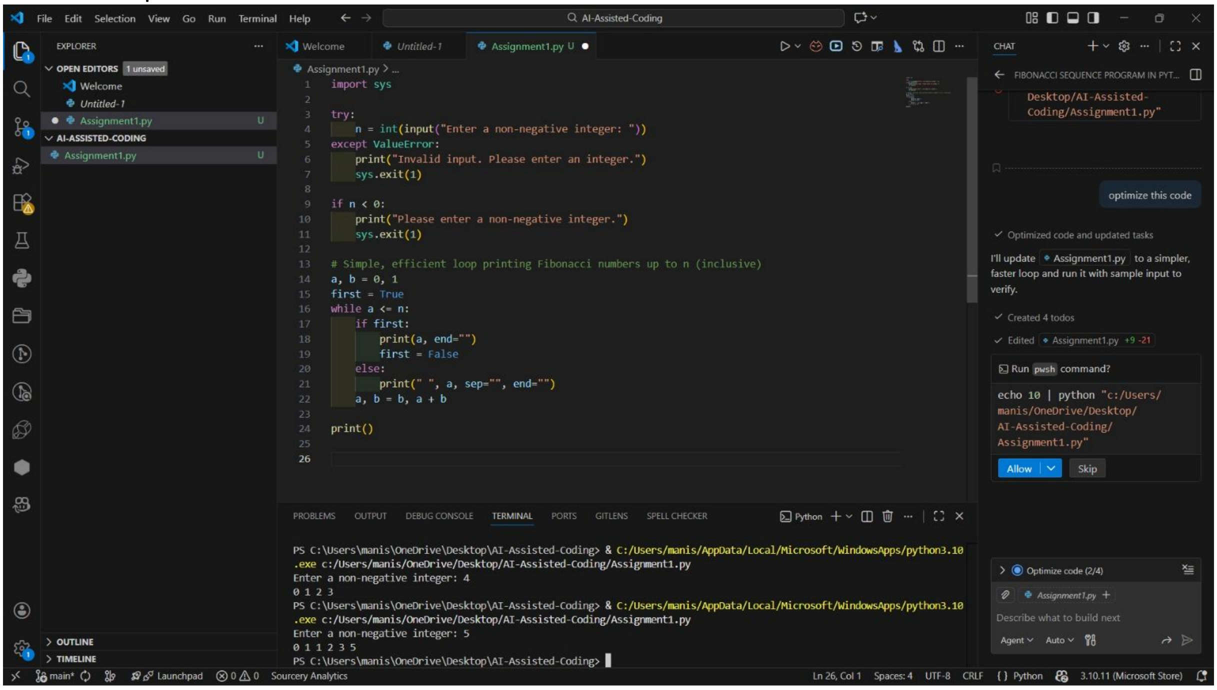
TASK:1

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



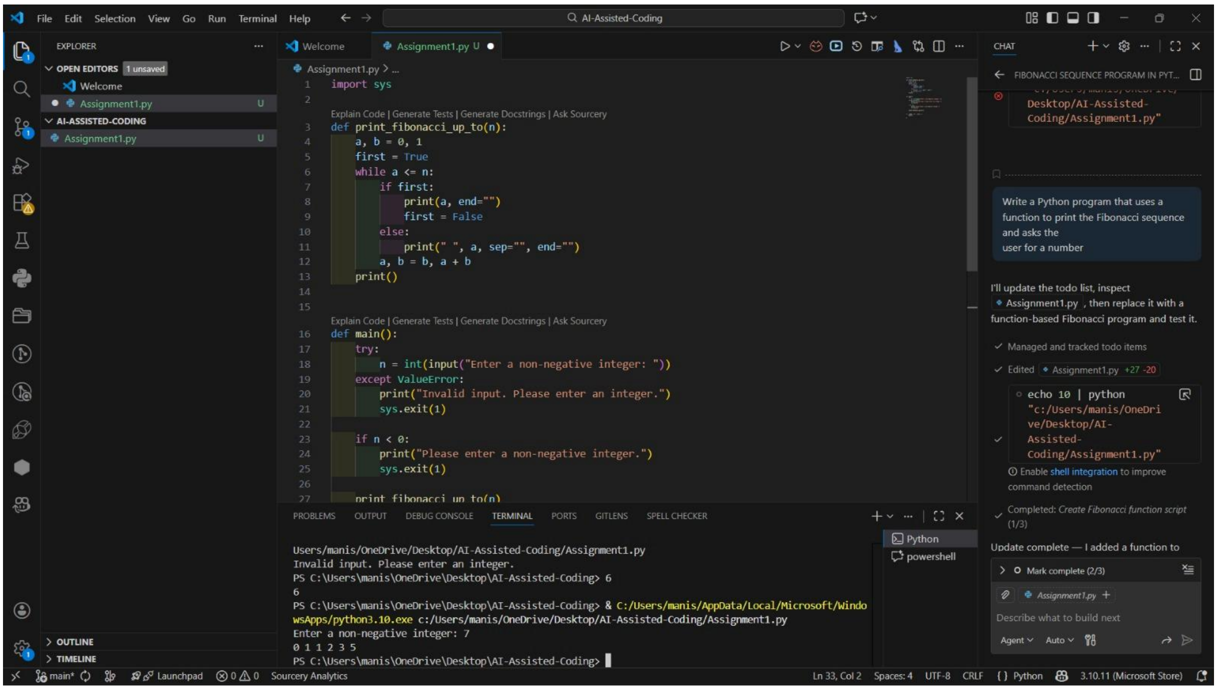
TASK:2

PROMPT: Optimize this code



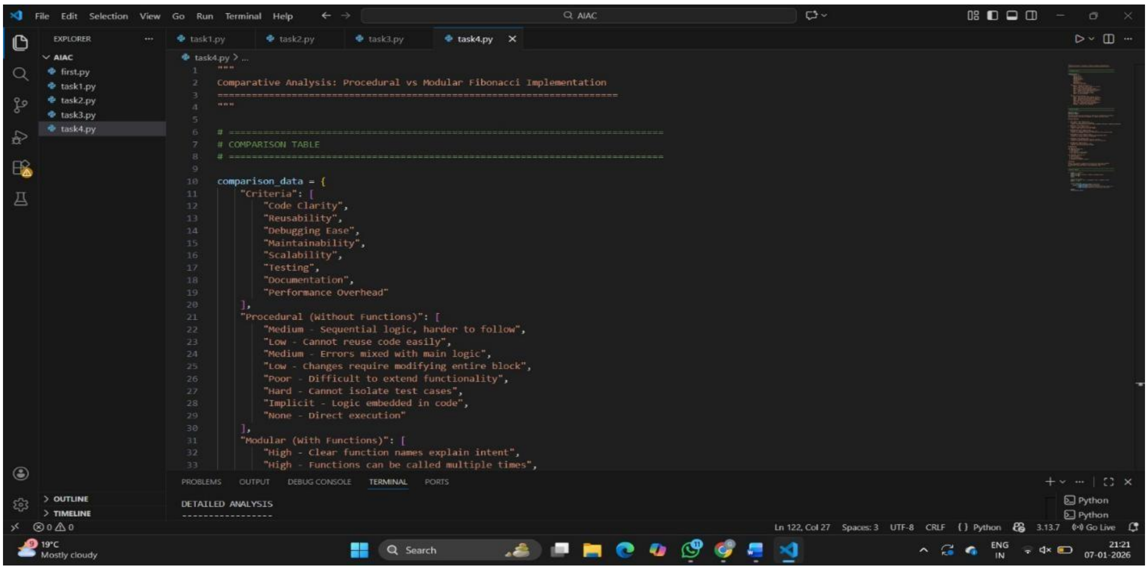

TASK:3

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



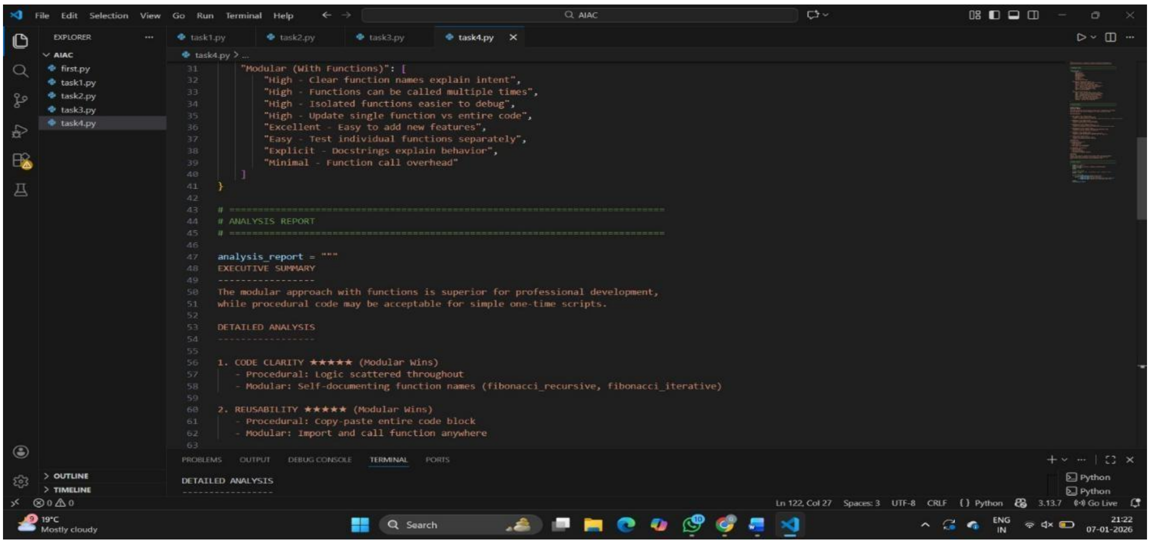
TASK:4

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



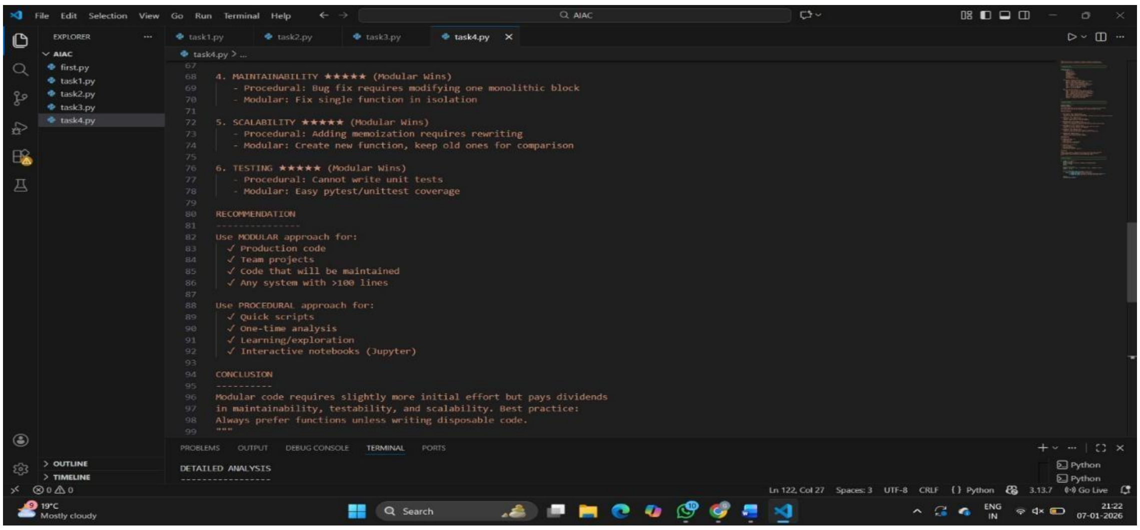
```python
"""
Comparative Analysis: Procedural vs Modular Fibonacci Implementation
======================================================================
"""

# ==============================================================================
# COMPARISON TABLE
# ==============================================================================

comparison_data = {
    "Criteria": [
        "Code Clarity",
        "Reusability",
        "Debugging Ease",
        "Maintainability",
        "Scalability",
        "Testing",
        "Documentation",
        "Performance Overhead"
    ],
    "Procedural (Without Functions)": [
        "Medium - Sequential logic, harder to follow",
        "Low - Cannot reuse code easily",
        "Medium - Errors mixed with main logic",
        "Low - Changes require modifying entire block",
        "Poor - Difficult to extend functionality",
        "Hard - Cannot isolate test cases",
        "Implicit - Logic embedded in code",
        "None - Direct execution"
    ],
    "Modular (With Functions)": [
        "High - Clear function names explain intent",
        "High - Functions can be called multiple times",
```

```
DETAILED ANALYSIS
-----------------
```



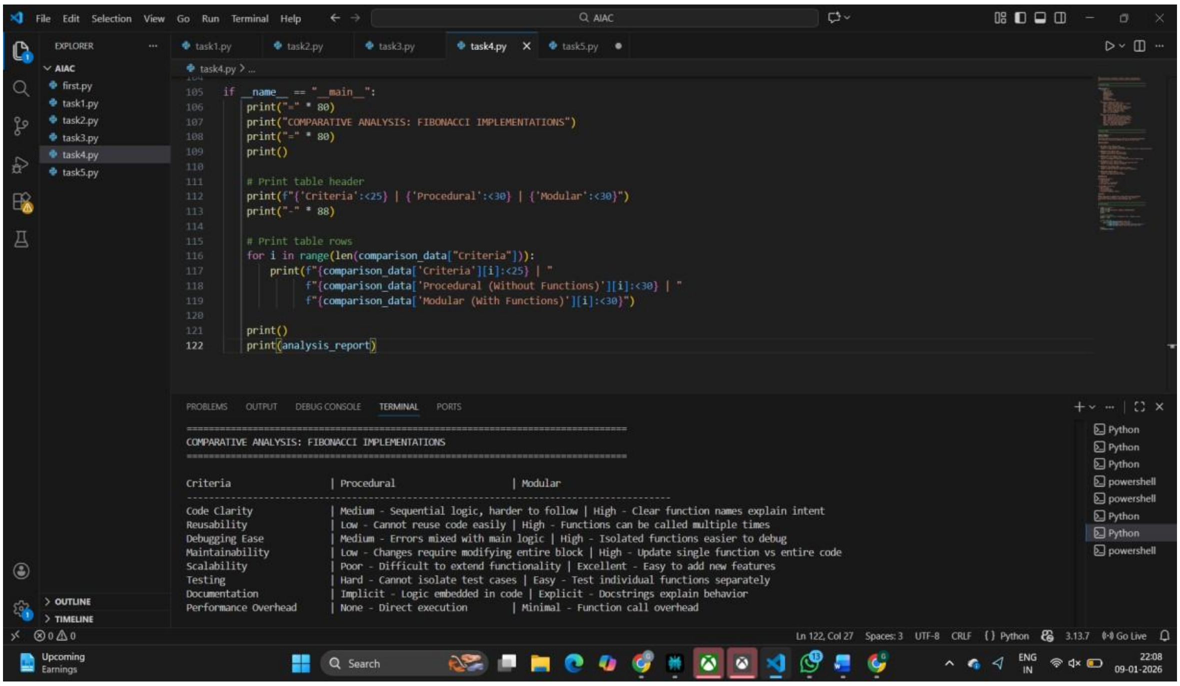
```python
    "Modular (With Functions)": [
        "High - Clear function names explain intent",
        "High - Functions can be called multiple times",
        "High - Isolated functions easier to debug",
        "High - Update single function vs entire code",
        "Excellent - Easy to add new features",
        "Easy - Test individual functions separately",
        "Explicit - Docstrings explain behavior",
        "Minimal - Function call overhead"
    ]
}

# ==============================================================================
# ANALYSIS REPORT
# ==============================================================================

analysis_report = """
EXECUTIVE SUMMARY
-----------------
The modular approach with functions is superior for professional development,
while procedural code may be acceptable for simple one-time scripts.

DETAILED ANALYSIS
-----------------

1. CODE CLARITY ★★★★★ (Modular Wins)
   - Procedural: Logic scattered throughout
   - Modular: Self-documenting function names (fibonacci_recursive, fibonacci_iterative)

2. REUSABILITY ★★★★★ (Modular Wins)
   - Procedural: Copy-paste entire code block
   - Modular: Import and call function anywhere
```

```
DETAILED ANALYSIS
-----------------
```



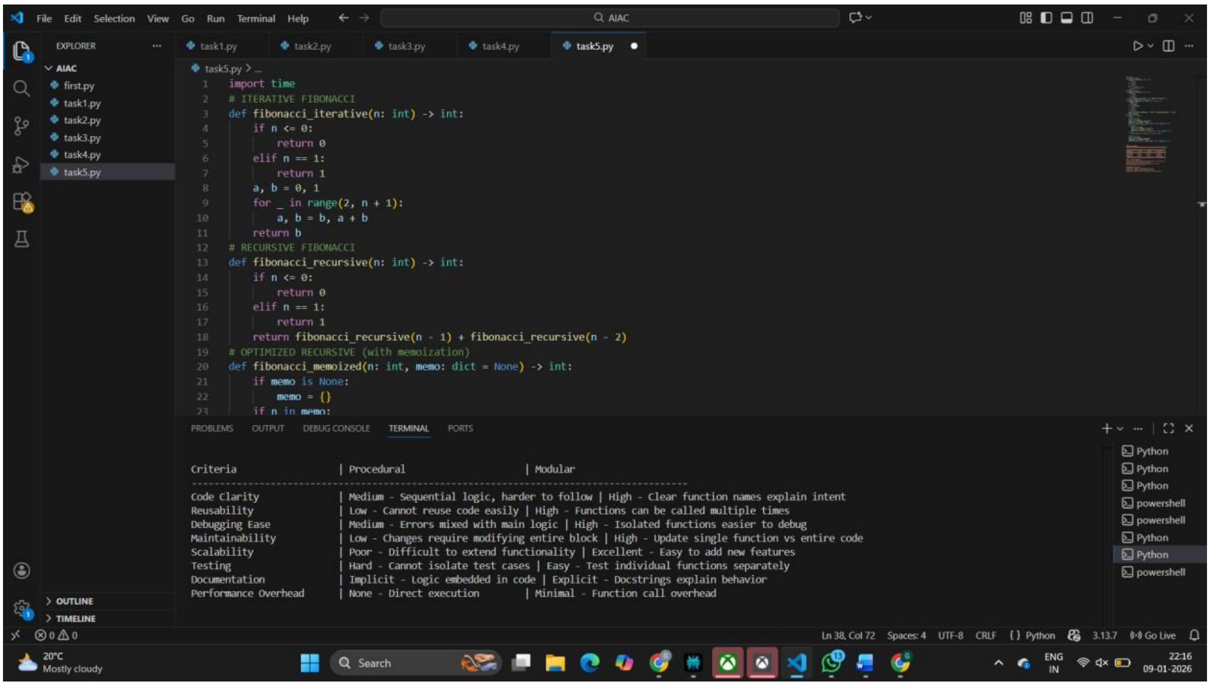
```python
4. MAINTAINABILITY ★★★★★ (Modular Wins)
   - Procedural: Bug fix requires modifying one monolithic block
   - Modular: Fix single function in isolation

5. SCALABILITY ★★★★★ (Modular Wins)
   - Procedural: Adding memoization requires rewriting
   - Modular: Create new function, keep old ones for comparison

6. TESTING ★★★★★ (Modular Wins)
   - Procedural: Cannot write unit tests
   - Modular: Easy pytest/unittest coverage

RECOMMENDATION
--------------
Use MODULAR approach for:
  ✓ Production code
  ✓ Team projects
  ✓ Code that will be maintained
  ✓ Any system with >100 lines

Use PROCEDURAL approach for:
  ✓ Quick scripts
  ✓ One-time analysis
  ✓ Learning/exploration
  ✓ Interactive notebooks (Jupyter)

CONCLUSION
----------
Modular code requires slightly more initial effort but pays dividends
in maintainability, testability, and scalability. Best practice:
Always prefer functions unless writing disposable code.
"""
```

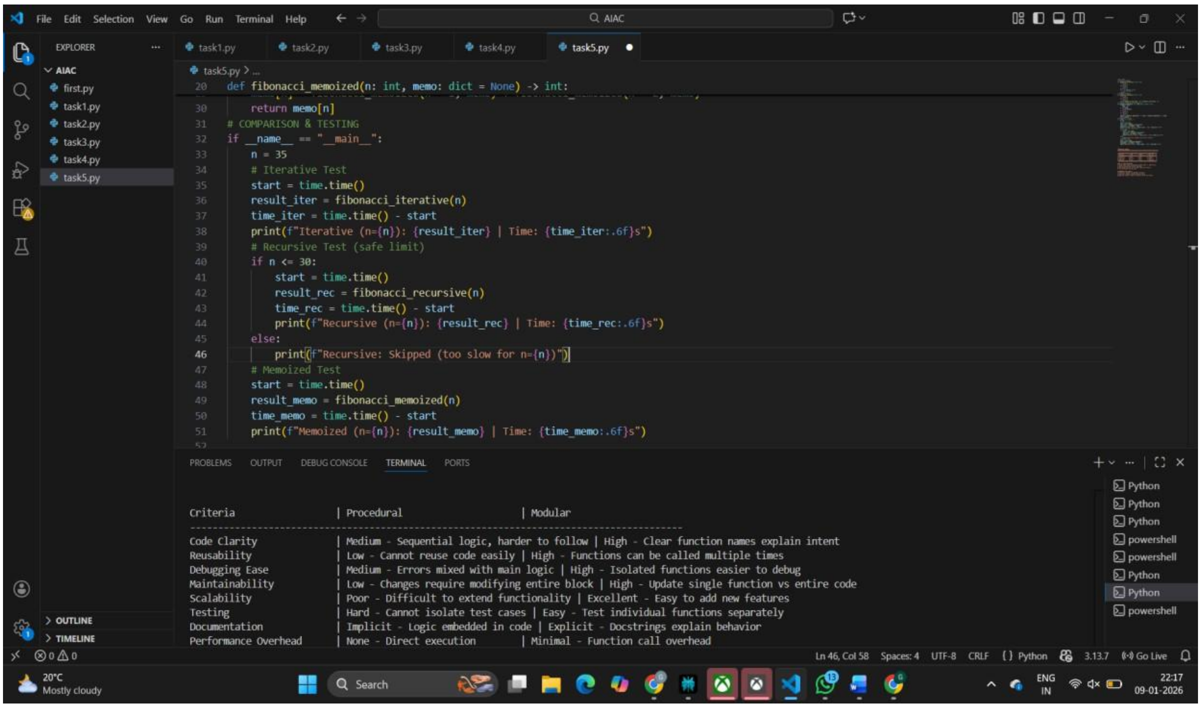
```
DETAILED ANALYSIS
-----------------
```




TASK:5

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





task1.py
task2.py
task3.py
task4.py
task5.py
def fibonacci_memoized(n: int, memo: dict = None) -> int:
    return memo[n]
# COMPARISON & TESTING
if __name__ == "__main__":
    n = 35
    # Iterative Test
    start = time.time()
    result_iter = fibonacci_iterative(n)
    time_iter = time.time() - start
    print(f"Iterative (n={n}): {result_iter} | Time: {time_iter:.6f}s")
    # Recursive Test (safe limit)
    if n <= 30:
        start = time.time()
        result_rec = fibonacci_recursive(n)
        time_rec = time.time() - start
        print(f"Recursive (n={n}): {result_rec} | Time: {time_rec:.6f}s")
    else:
        print(f"Recursive: Skipped (too slow for n={n})")
    # Memoized Test
    start = time.time()
    result_memo = fibonacci_memoized(n)
    time_memo = time.time() - start
    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
Ln 46, Col 58