ASSIGNMENT-9.1

G. Rishika
2303A52197
Batch: 35

Lab Experiment: Documentation Generation -Automatic documentation and code comments

Problem 1:

Consider the following Python function:

def find_max(numbers):

return max(numbers)

Task:

• Write documentation for the function in all three formats:

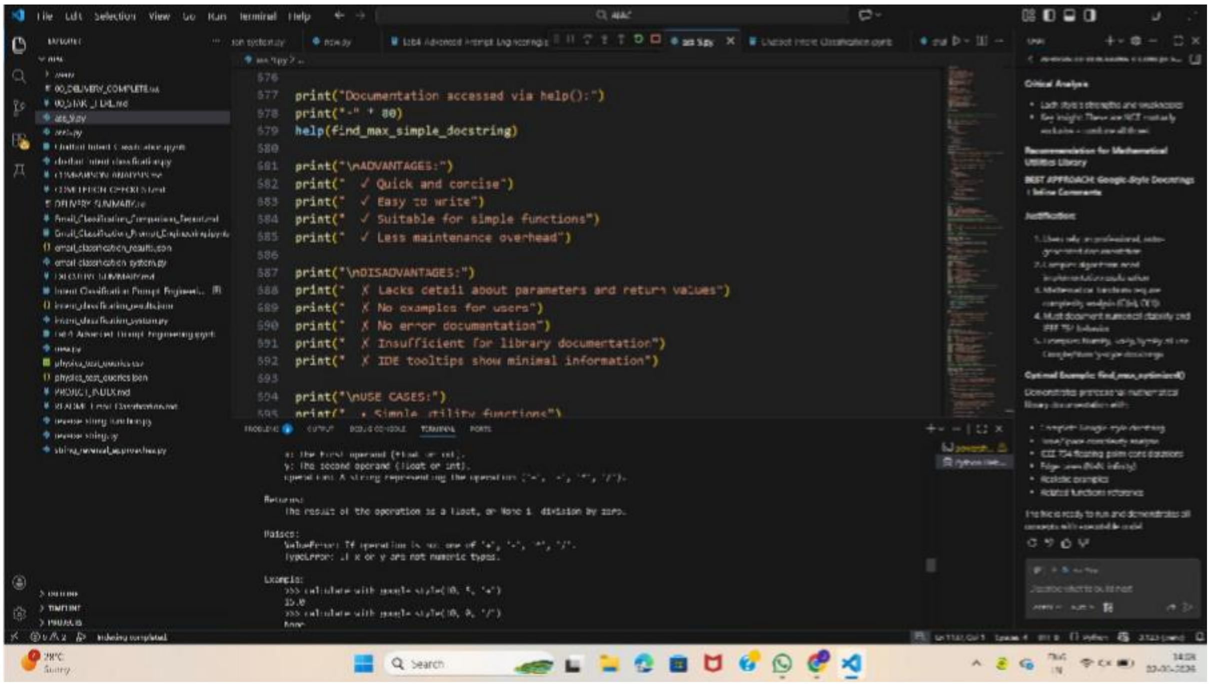
(a) Docstring

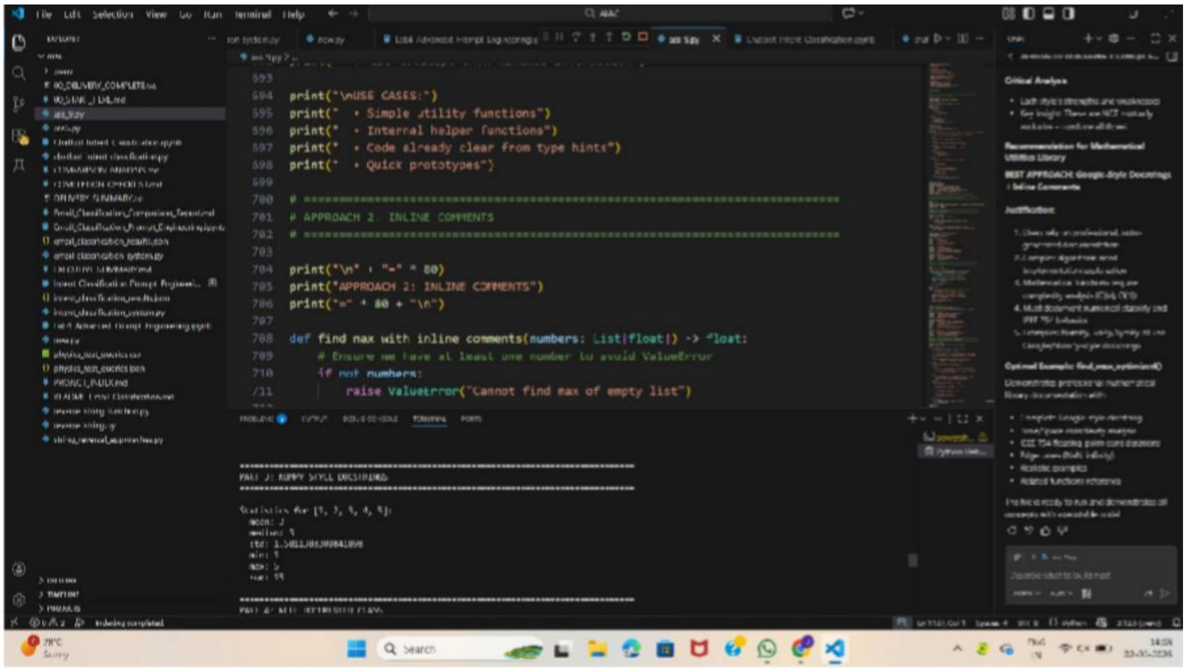
(b) Inline comments

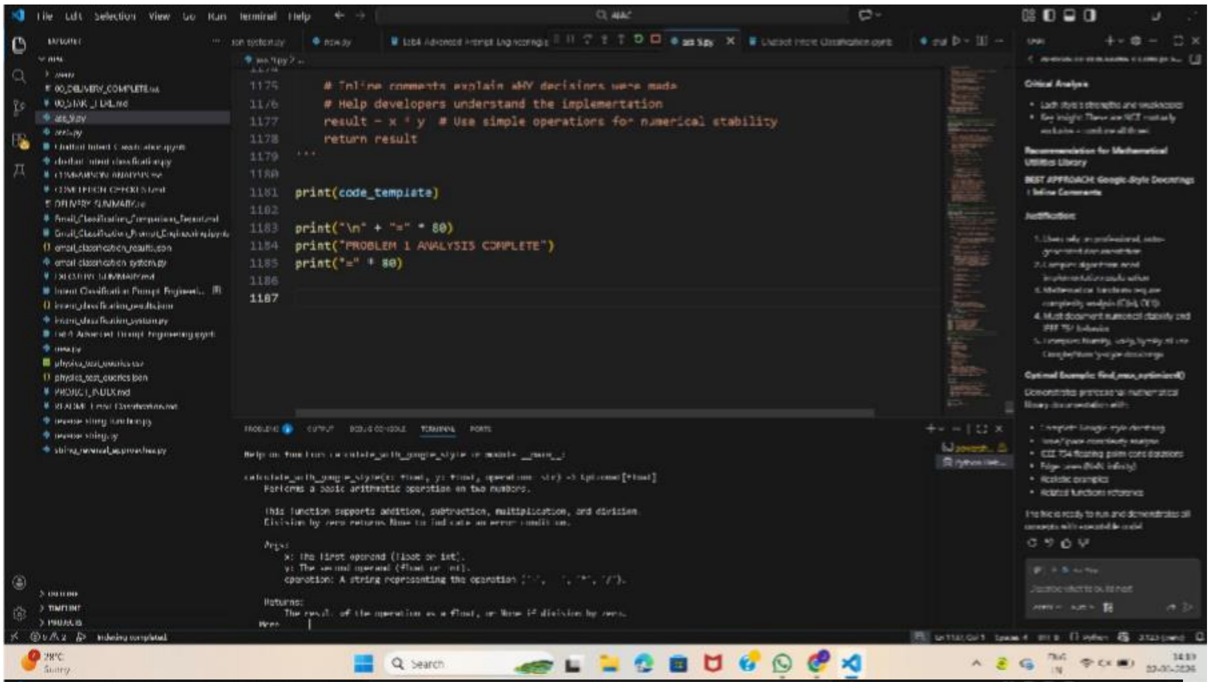
(c) Google-style documentation

• Critically compare the three approaches. Discuss the advantages, disadvantages, and suitable use cases of each style.

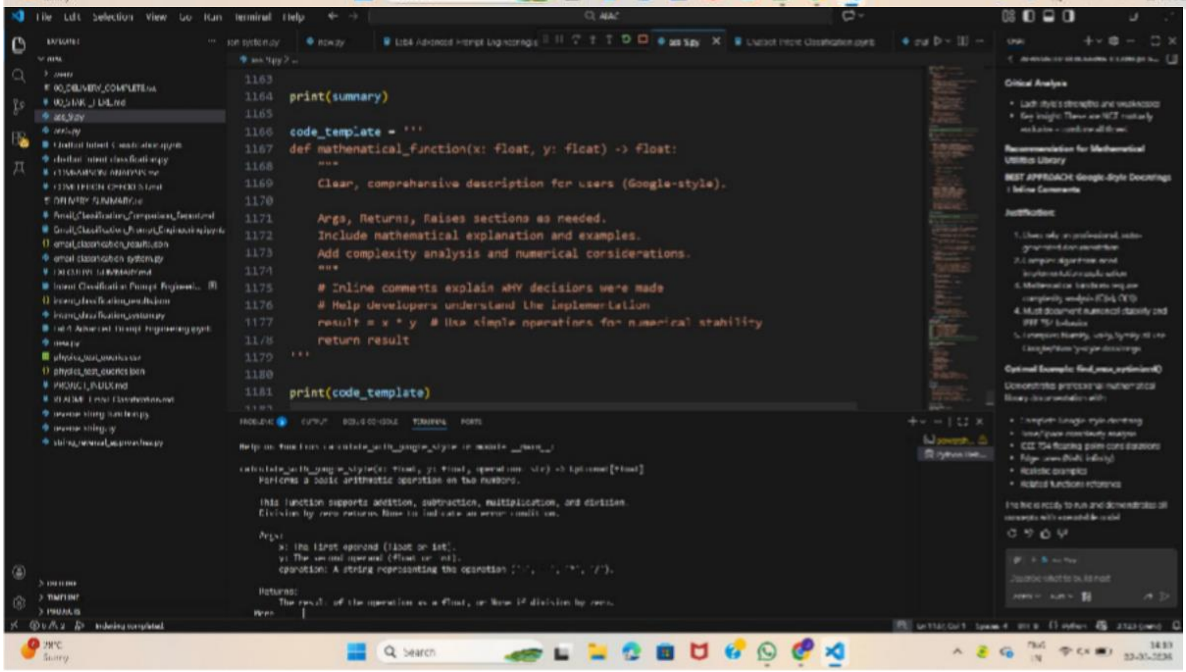
• Recommend which documentation style is most effective for a mathematical utilities library and justify your answer.


print("Documentation accessed via help():")
print("-" * 80)
help(find_max_simple_docstring)
print("\nADVANTAGES:")
print(" ✓ Quick and concise")
print(" ✓ Easy to write")
print(" ✓ Suitable for simple functions")
print(" ✓ Less maintenance overhead")
print("\nDISADVANTAGES:")
print(" ✗ Lacks detail about parameters and return values")
print(" ✗ No examples for users")
print(" ✗ No error documentation")
print(" ✗ Insufficient for library documentation")
print(" ✗ IDE tooltips show minimal information")
print("\nUSE CASES:")
Returns:
Raises:
Example:

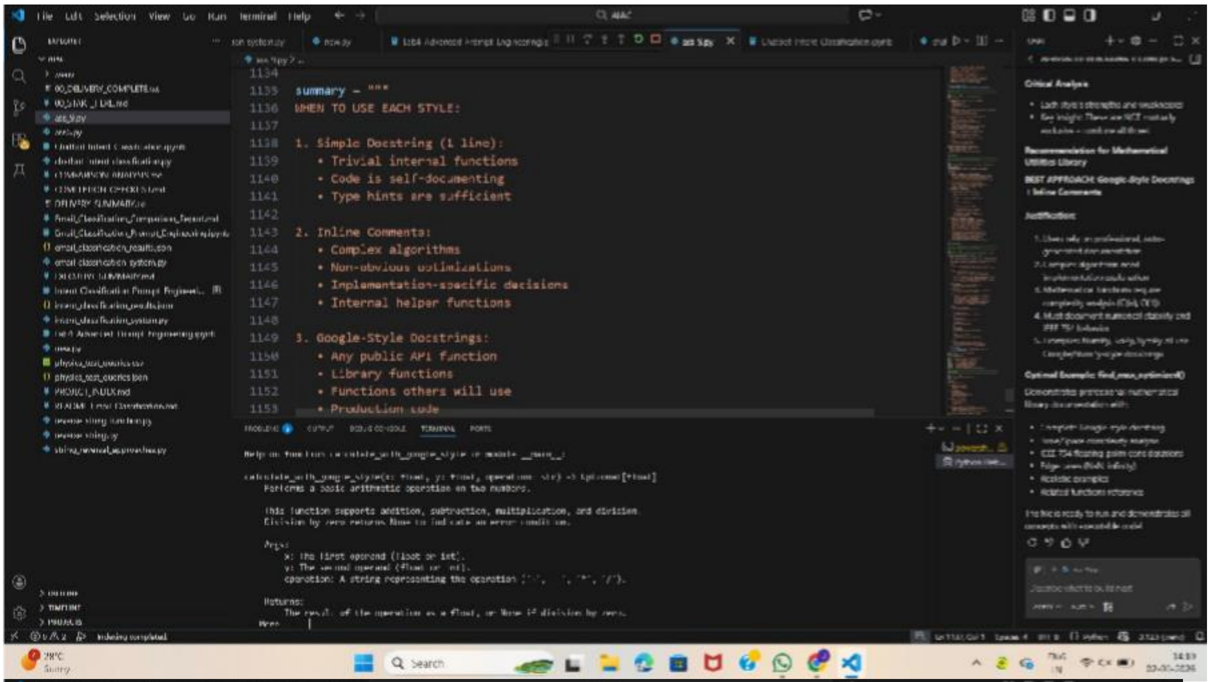

print("\nUSE CASES:")
print(" • Simple utility functions")
print(" • Internal helper functions")
print(" • Code already clear from type hints")
print(" • Quick prototypes")
# APPROACH 2: INLINE COMMENTS
print("\n" + "=" * 80)
print("APPROACH 2: INLINE COMMENTS")
print("=" * 80 + "\n")
def find_max_with_inline_comments(numbers: List[float]) -> float:
# Ensure we have at least one number to avoid ValueError
if not numbers:
raise ValueError("Cannot find max of empty list")

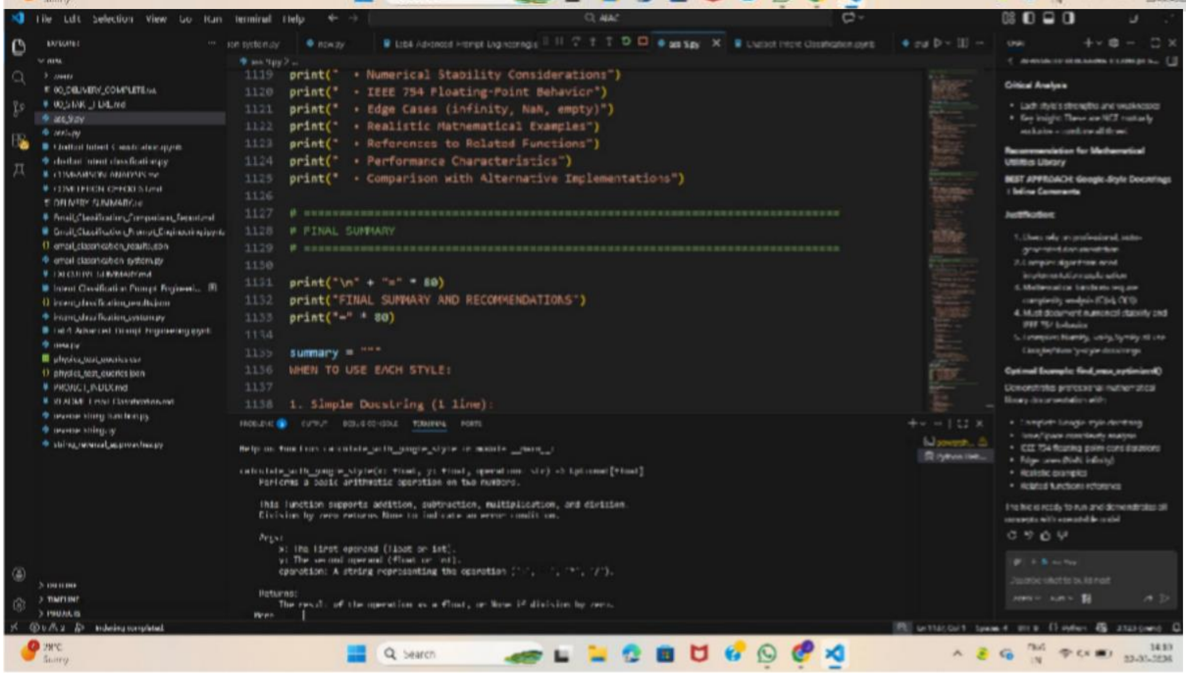


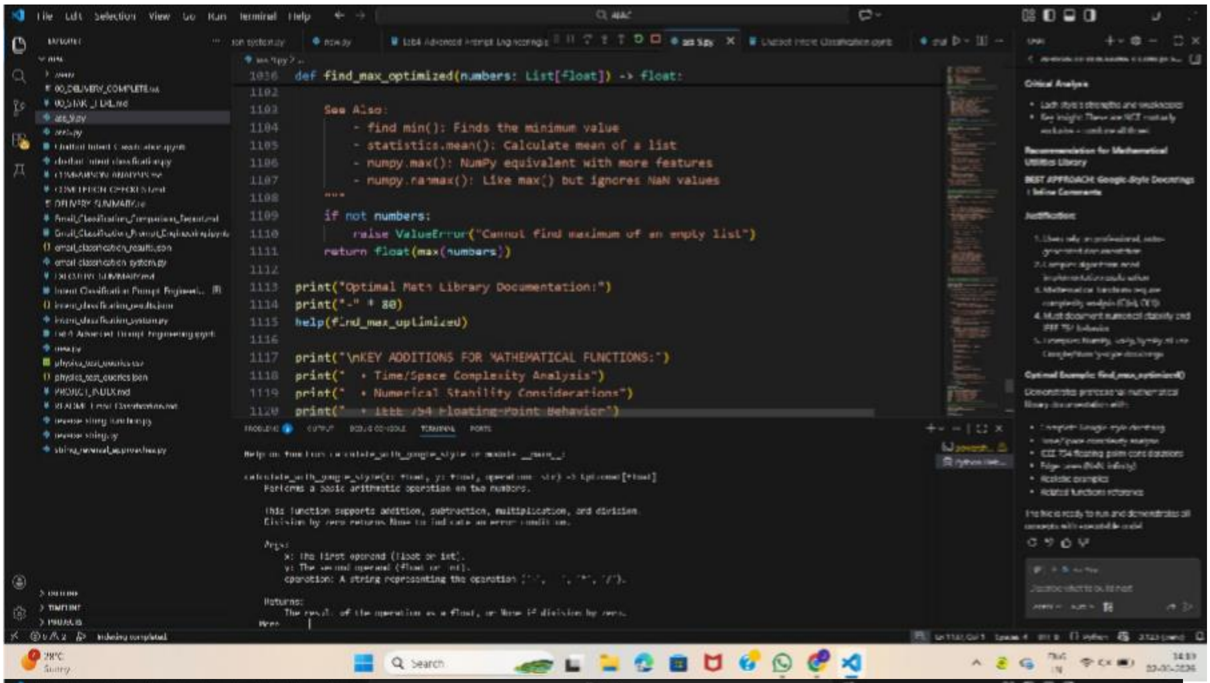
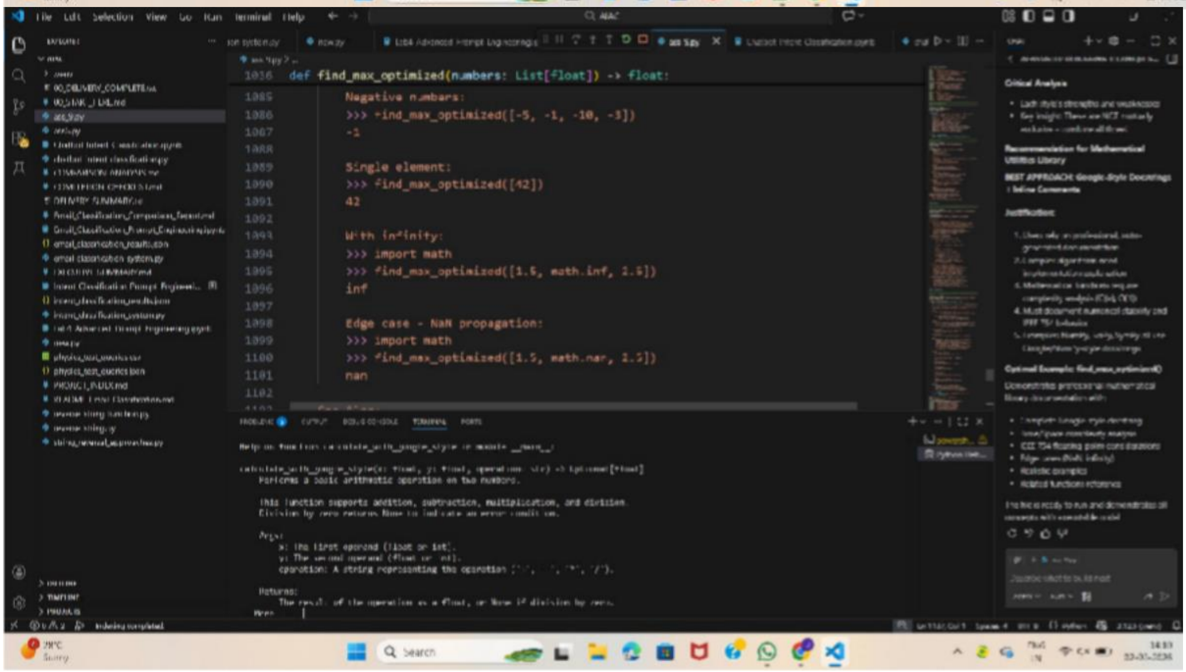
# Inline comments explain WHY decisions were made
# Help developers understand the implementation
result = x * y # Use simple operations for numerical stability
return result
'''
print(code_template)
print("\n" + "=" * 80)
print("PROBLEM 1 ANALYSIS COMPLETE")
print("=" * 80)
Performs a basic arithmetic operation on two numbers.
This function supports addition, subtraction, multiplication, and division.
Args:
x: The first operand (float or int).
Returns:

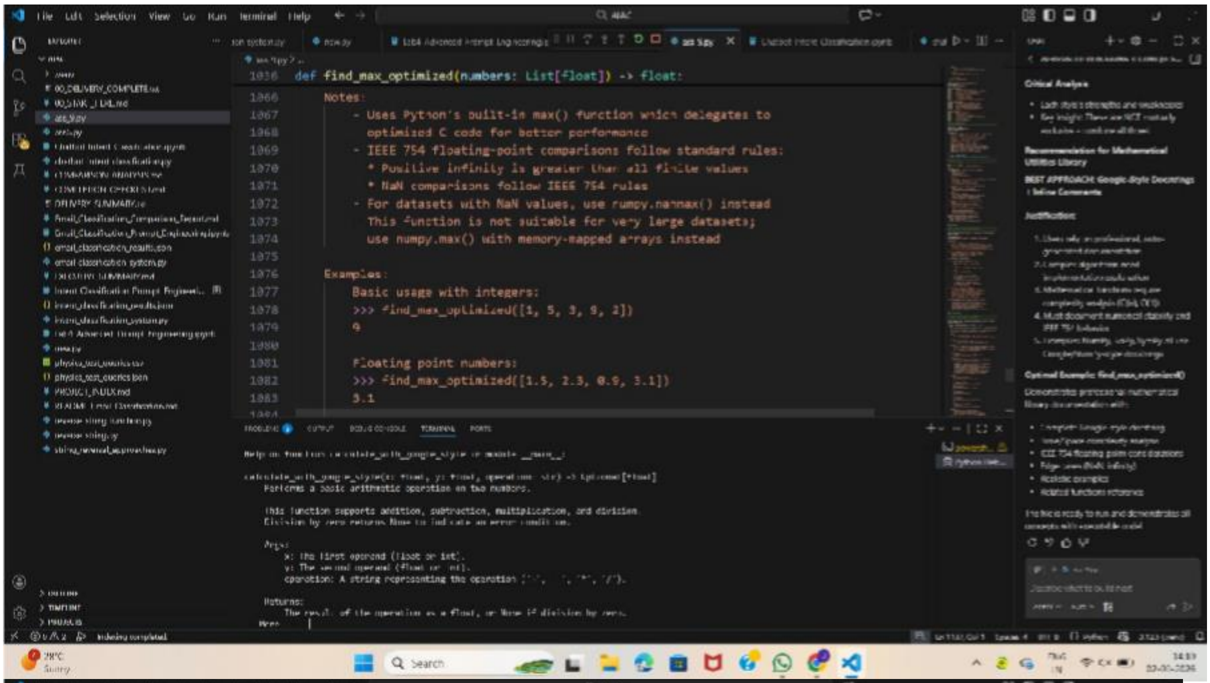
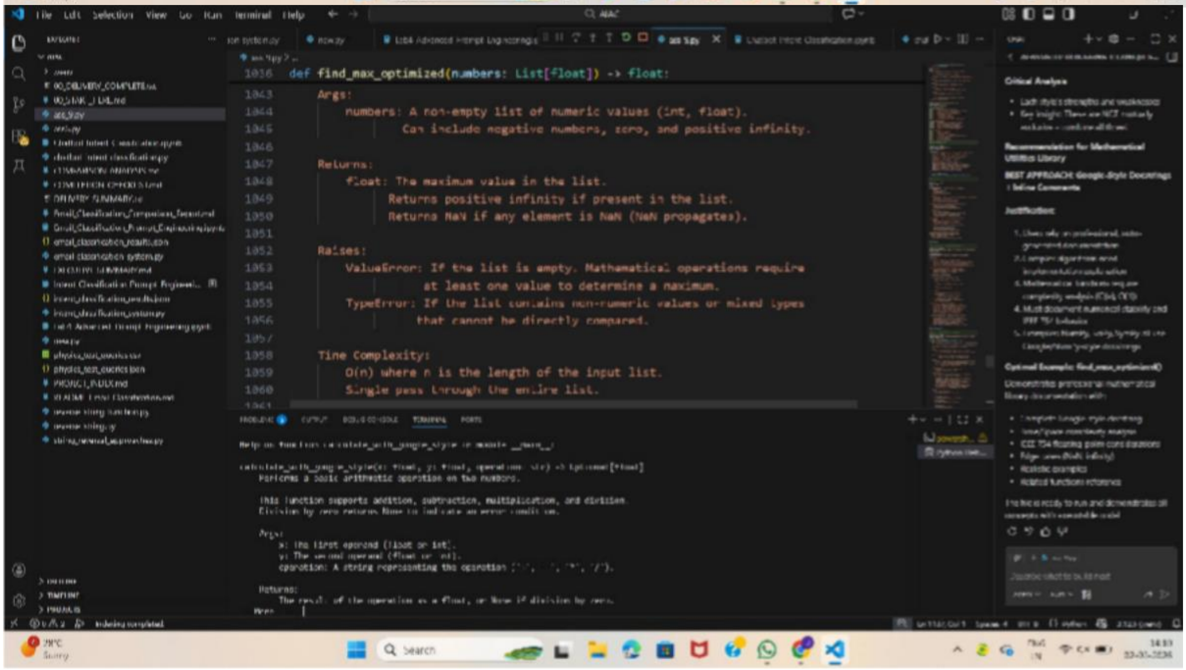


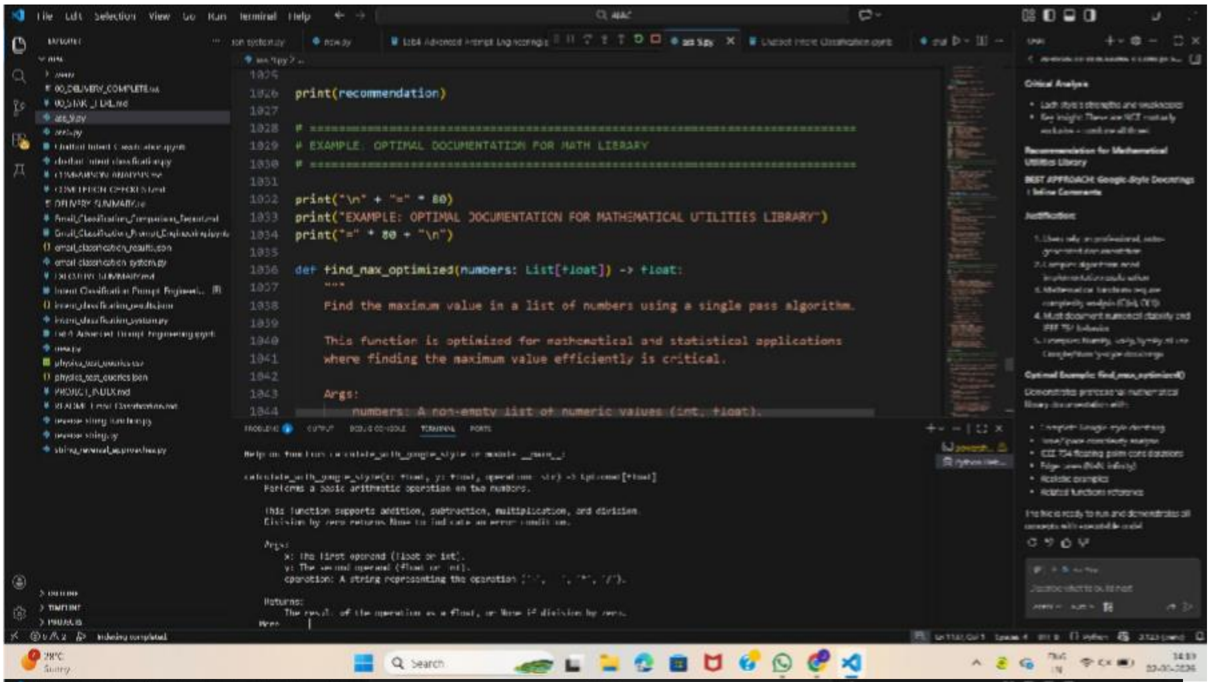
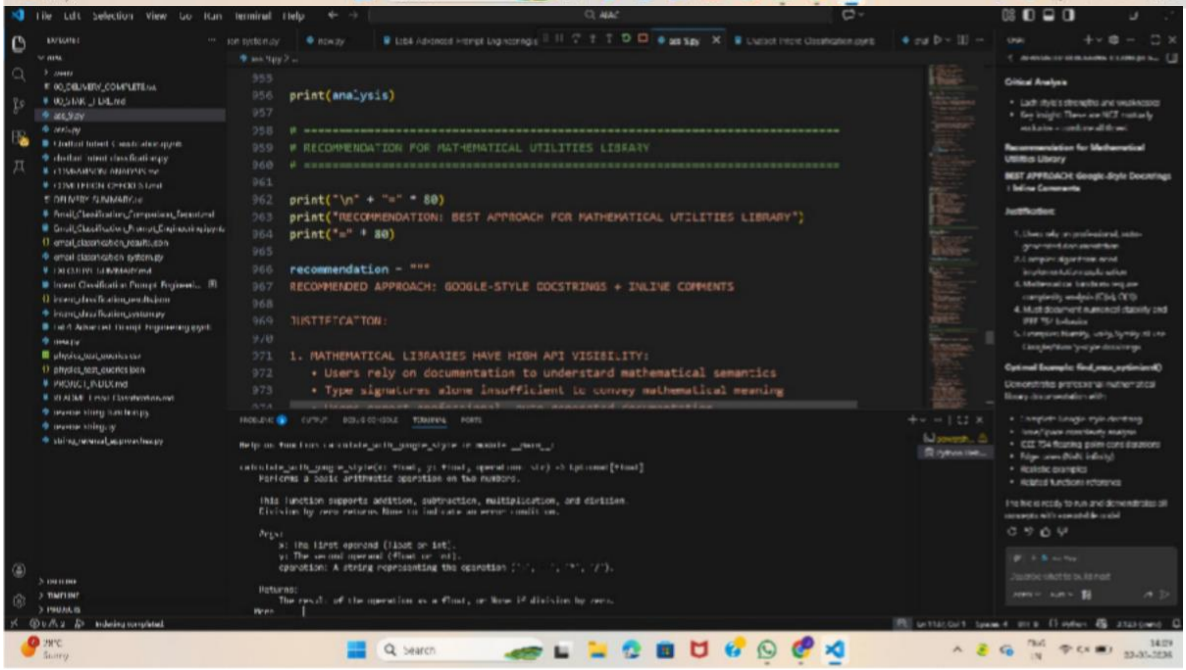
print(summary)
code_template = '''
def mathematical_function(x: float, y: float) -> float:
"""
Clear, comprehensive description for users (Google-style).
Args, Returns, Raises sections as needed.
Include mathematical explanation and examples.
Add complexity analysis and numerical considerations.
"""
# Inline comments explain WHY decisions were made
# Help developers understand the implementation
result = x * y # Use simple operations for numerical stability
return result
'''
print(code_template)
Performs a basic arithmetic operation on two numbers.
This function supports addition, subtraction, multiplication, and division.
Args:
x: The first operand (float or int).
Returns:

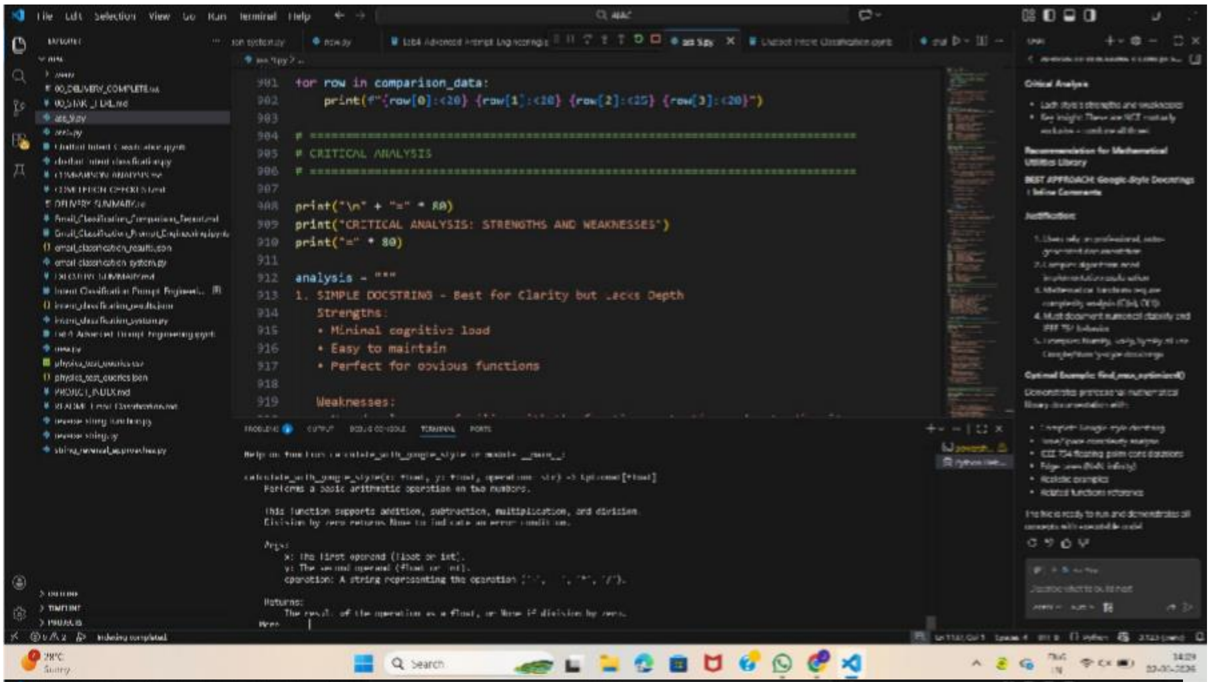

summary = """
WHEN TO USE EACH STYLE:
1. Simple Docstring (1 line):
• Trivial internal functions
• Code is self-documenting
• Type hints are sufficient
2. Inline Comments:
• Complex algorithms
• Non-obvious optimizations
• Implementation-specific decisions
• Internal helper functions
3. Google-Style Docstrings:
• Any public API function
• Library functions
• Functions others will use
• Production code
Performs a basic arithmetic operation on two numbers.
This function supports addition, subtraction, multiplication, and division.
Args:
x: The first operand (float or int).
Returns:
Critical Analysis
Recommendation for Mathematical Utilities Library
BEST APPROACH: Google-Style Docstrings + Inline Comments
Justification:

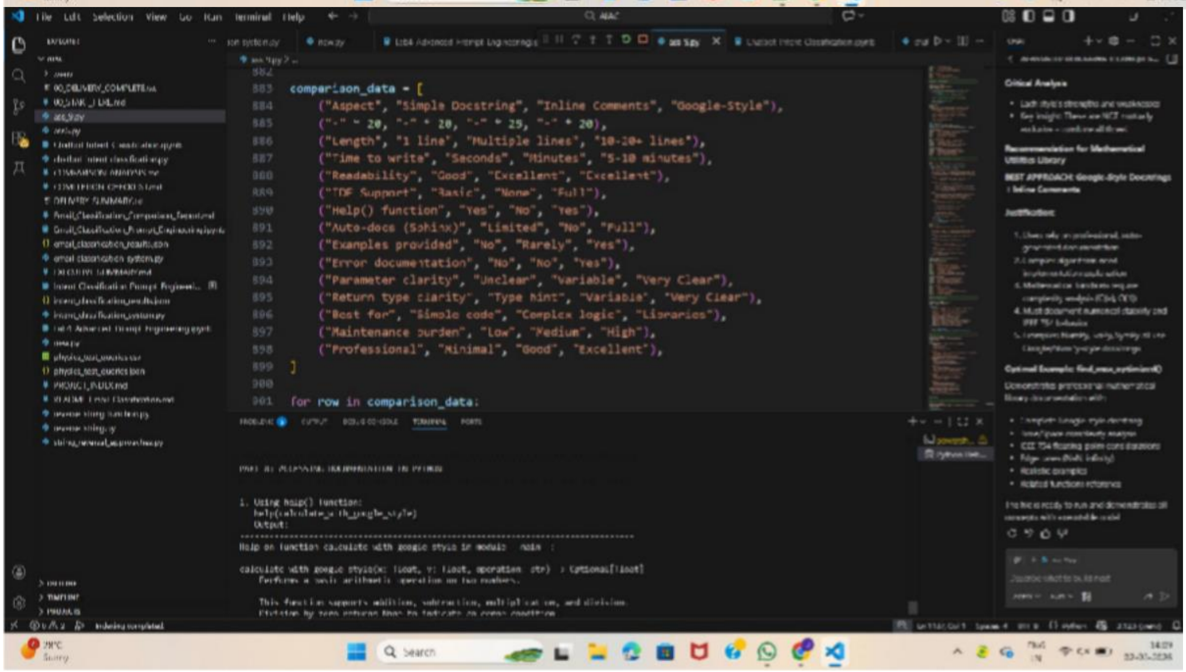

print(" • Numerical Stability Considerations")
print(" • IEEE 754 Floating-Point Behavior")
print(" • Edge Cases (infinity, NaN, empty)")
print(" • Realistic Mathematical Examples")
print(" • References to Related Functions")
print(" • Performance Characteristics")
print(" • Comparison with Alternative Implementations")
# FINAL SUMMARY
print("\n" + "=" * 80)
print("FINAL SUMMARY AND RECOMMENDATIONS")
print("=" * 80)
summary = """
WHEN TO USE EACH STYLE:
1. Simple Docstring (1 line):
Performs a basic arithmetic operation on two numbers.
This function supports addition, subtraction, multiplication, and division.
Args:
Returns:
Critical Analysis
Recommendation for Mathematical Utilities Library
BEST APPROACH: Google-Style Docstrings + Inline Comments
Justification:

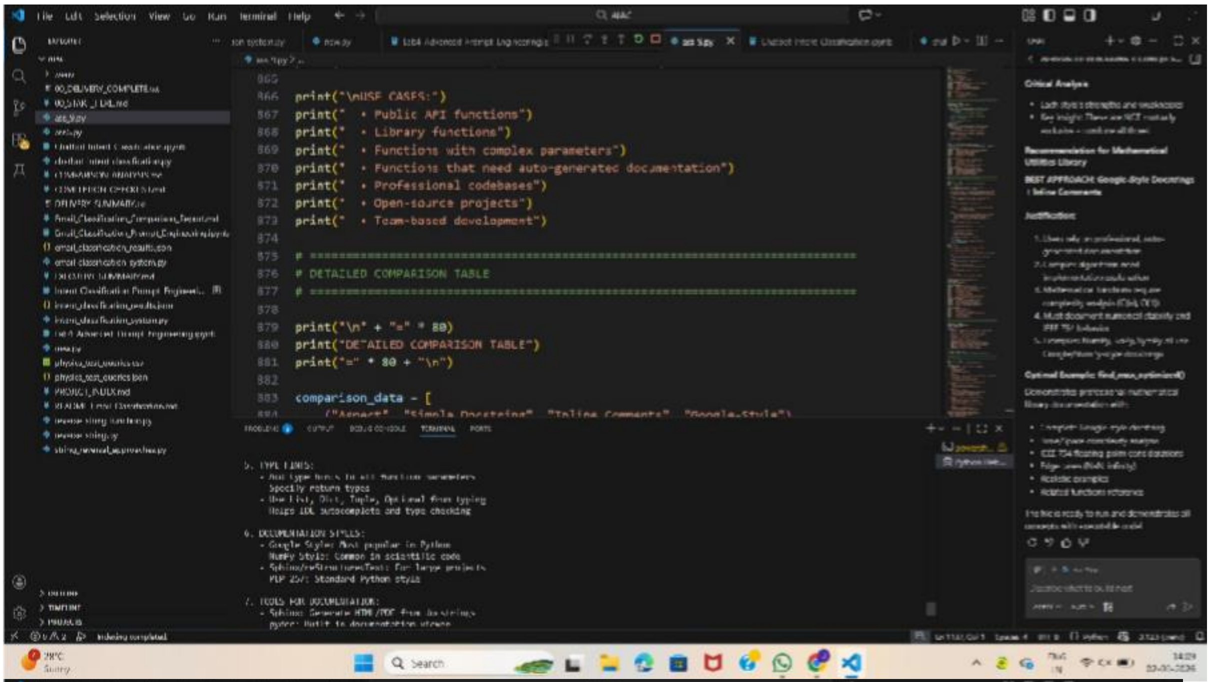

def find_max_optimized(numbers: List[float]) -> float:
See Also:
- find_min(): Finds the minimum value
- statistics.mean(): Calculate mean of a list
- numpy.max(): NumPy equivalent with more features
- numpy.nanmax(): Like max() but ignores NaN values
if not numbers:
raise ValueError("Cannot find maximum of an empty list")
return float(max(numbers))
print("Optimal Math Library Documentation:")
print("-" * 80)
help(find_max_optimized)
print("\nKEY ADDITIONS FOR MATHEMATICAL FUNCTIONS:")
print(" • Time/Space Complexity Analysis")
print(" • Numerical Stability Considerations")
Critical Analysis
Recommendation for Mathematical Utilities Library
BEST APPROACH: Google-Style Docstrings + Inline Comments
Justification:

def find_max_optimized(numbers: List[float]) -> float:
Negative numbers:
>>> find_max_optimized([-5, -1, -10, -3])
-1
Single element:
>>> find_max_optimized([42])
42
With infinity:
>>> import math
>>> find_max_optimized([1.5, math.inf, 2.5])
inf
Edge case - NaN propagation:
>>> import math
>>> find_max_optimized([1.5, math.nan, 2.5])
nan
Critical Analysis
Recommendation for Mathematical Utilities Library
BEST APPROACH: Google-Style Docstrings + Inline Comments
Justification:

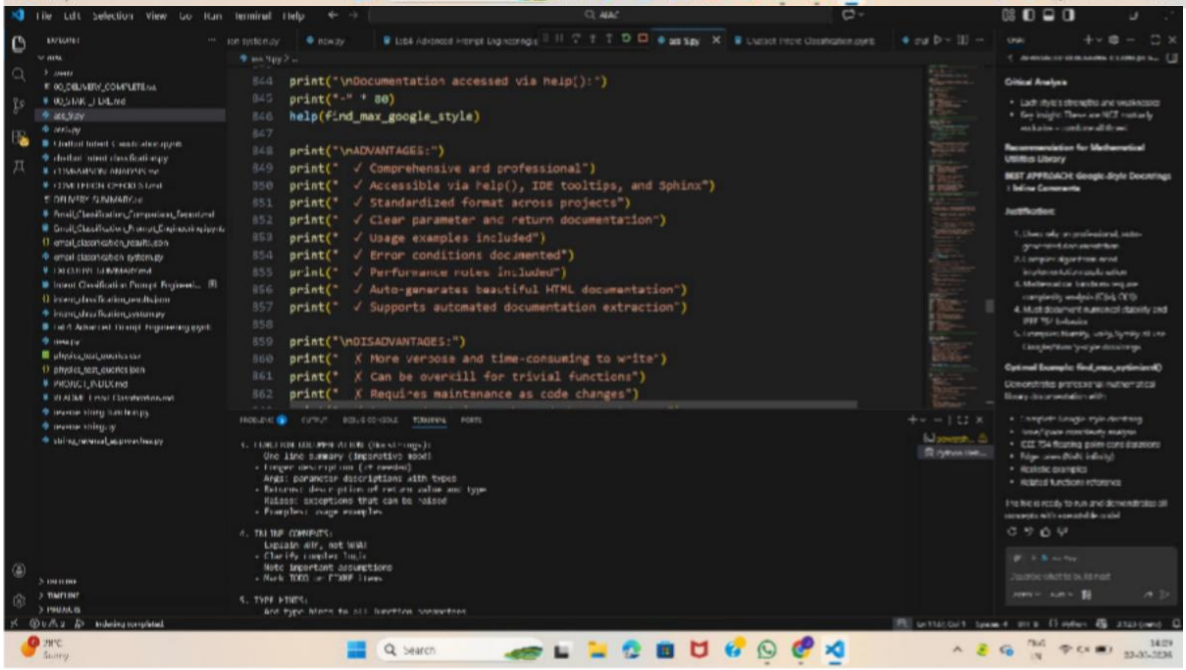

def find_max_optimized(numbers: List[float]) -> float:
Notes:
- Uses Python's built-in max() function which delegates to
optimized C code for better performance
- IEEE 754 floating-point comparisons follow standard rules:
* Positive infinity is greater than all finite values
* NaN comparisons follow IEEE 754 rules
- For datasets with NaN values, use numpy.nanmax() instead
This function is not suitable for very large datasets;
use numpy.max() with memory-mapped arrays instead
Examples:
Basic usage with integers:
>>> find_max_optimized([1, 5, 3, 9, 2])
9
Floating point numbers:
>>> find_max_optimized([1.5, 2.3, 0.9, 3.1])
3.1
Performs a basic arithmetic operation on two numbers.
This function supports addition, subtraction, multiplication, and division.
Search

def find_max_optimized(numbers: List[float]) -> float:
Args:
numbers: A non-empty list of numeric values (int, float).
Can include negative numbers, zero, and positive infinity.
Returns:
float: The maximum value in the list.
Returns positive infinity if present in the list.
Returns NaN if any element is NaN (NaN propagates).
Raises:
ValueError: If the list is empty. Mathematical operations require
at least one value to determine a maximum.
TypeError: If the list contains non-numeric values or mixed types
that cannot be directly compared.
Time Complexity:
O(n) where n is the length of the input list.
Single pass through the entire list.
Performs a basic arithmetic operation on two numbers.
This function supports addition, subtraction, multiplication, and division.
Search

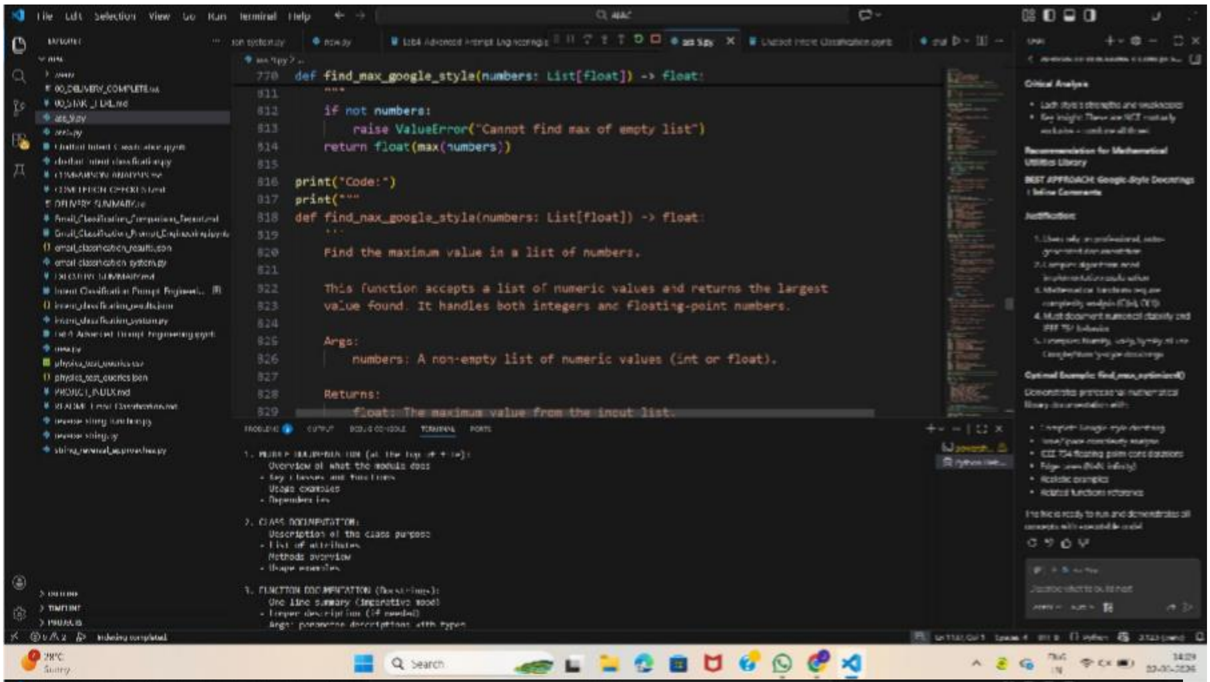

print(recommendation)
# EXAMPLE: OPTIMAL DOCUMENTATION FOR MATH LIBRARY
print("\n" + "=" * 80)
print("EXAMPLE: OPTIMAL DOCUMENTATION FOR MATHEMATICAL UTILITIES LIBRARY")
print("=" * 80 + "\n")
def find_max_optimized(numbers: List[float]) -> float:
Find the maximum value in a list of numbers using a single pass algorithm.
This function is optimized for mathematical and statistical applications
where finding the maximum value efficiently is critical.
Args:
numbers: A non-empty list of numeric values (int, float).
Performs a basic arithmetic operation on two numbers.
This function supports addition, subtraction, multiplication, and division.
Critical Analysis
Recommendation for Mathematical Utilities Library
BEST APPROACH: Google-Style Docstrings + Inline Comments
Justification:

print(analysis)
# RECOMMENDATION FOR MATHEMATICAL UTILITIES LIBRARY
print("\n" + "=" * 80)
print("RECOMMENDATION: BEST APPROACH FOR MATHEMATICAL UTILITIES LIBRARY")
print("=" * 80)
recommendation = """
RECOMMENDED APPROACH: GOOGLE-STYLE DOCSTRINGS + INLINE COMMENTS
JUSTIFICATION:
1. MATHEMATICAL LIBRARIES HAVE HIGH API VISIBILITY:
• Users rely on documentation to understand mathematical semantics
• Type signatures alone insufficient to convey mathematical meaning
Performs a basic arithmetic operation on two numbers.
This function supports addition, subtraction, multiplication, and division.
Critical Analysis
Recommendation for Mathematical Utilities Library
BEST APPROACH: Google-Style Docstrings + Inline Comments
Justification:

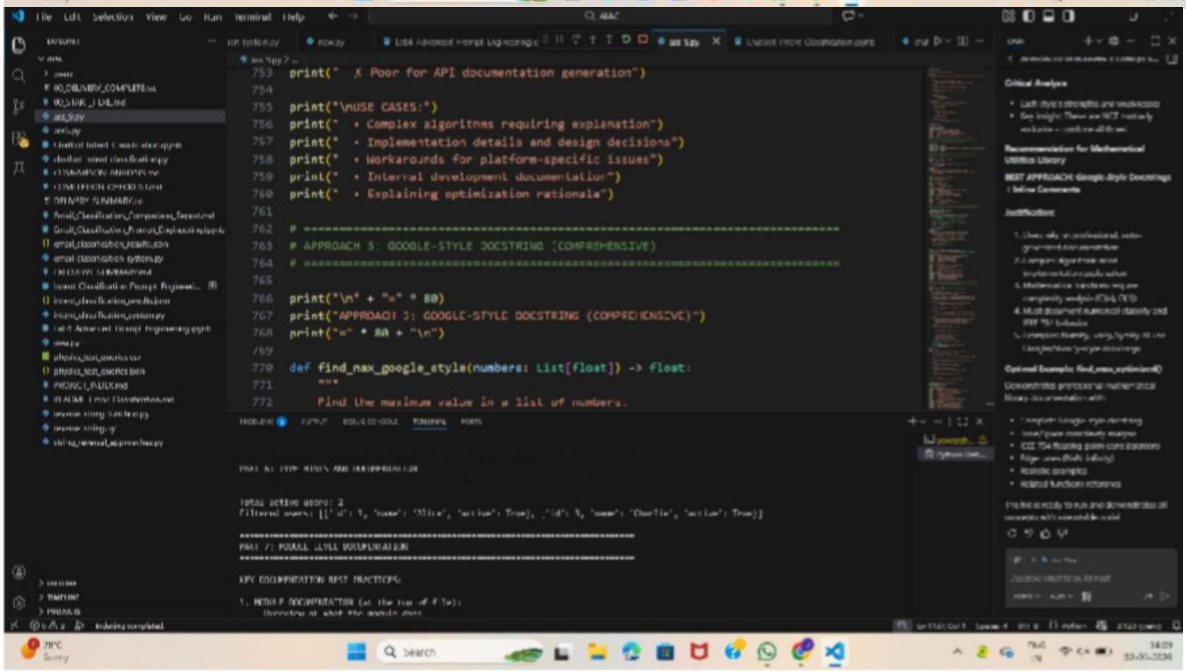

for row in comparison_data:
print(f"{row[0]:<20} {row[1]:<20} {row[2]:<25} {row[3]:<20}")
# CRITICAL ANALYSIS
print("\n" + "=" * 80)
print("CRITICAL ANALYSIS: STRENGTHS AND WEAKNESSES")
print("=" * 80)
analysis = """
1. SIMPLE DOCSTRING - Best for Clarity but Lacks Depth
Strengths:
• Minimal cognitive load
• Easy to maintain
• Perfect for obvious functions
Weaknesses:
Performs a basic arithmetic operation on two numbers.
This function supports addition, subtraction, multiplication, and division.
Args:
x: The first operand (float or int).
Returns:
Critical Analysis
Justification:
Optimal Example: find_max_optimized()

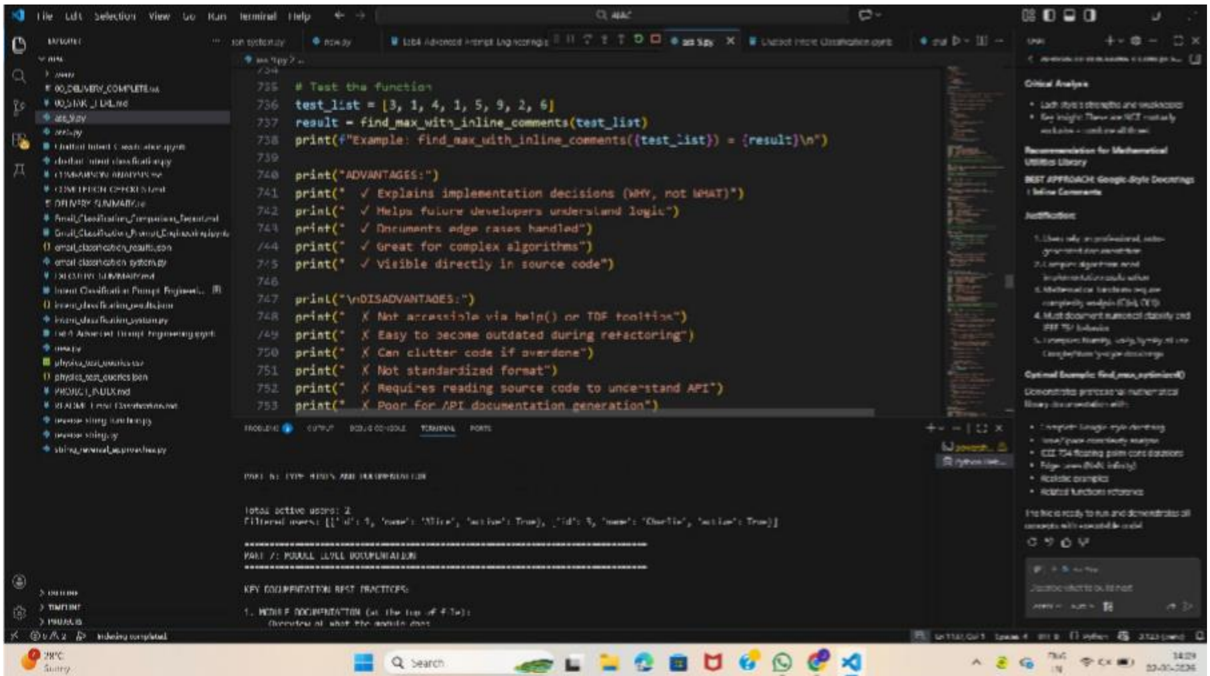

comparison_data = [
("Aspect", "Simple Docstring", "Inline Comments", "Google-Style"),
("-" * 20, "-" * 20, "-" * 25, "-" * 20),
("Length", "1 line", "Multiple lines", "10-20+ lines"),
("Time to write", "Seconds", "Minutes", "5-10 minutes"),
("Readability", "Good", "Excellent", "Excellent"),
("IDE Support", "Basic", "None", "Full"),
("Help() function", "Yes", "No", "Yes"),
("Auto-docs (Sphinx)", "Limited", "No", "Full"),
("Examples provided", "No", "Rarely", "Yes"),
("Error documentation", "No", "No", "Yes"),
("Parameter clarity", "Unclear", "Variable", "Very Clear"),
("Return type clarity", "Type hint", "Variable", "Very Clear"),
("Best for", "Simple code", "Complex logic", "Libraries"),
("Maintenance burden", "Low", "Medium", "High"),
("Professional", "Minimal", "Good", "Excellent"),
]
for row in comparison_data:
1. Using help() function:
Critical Analysis

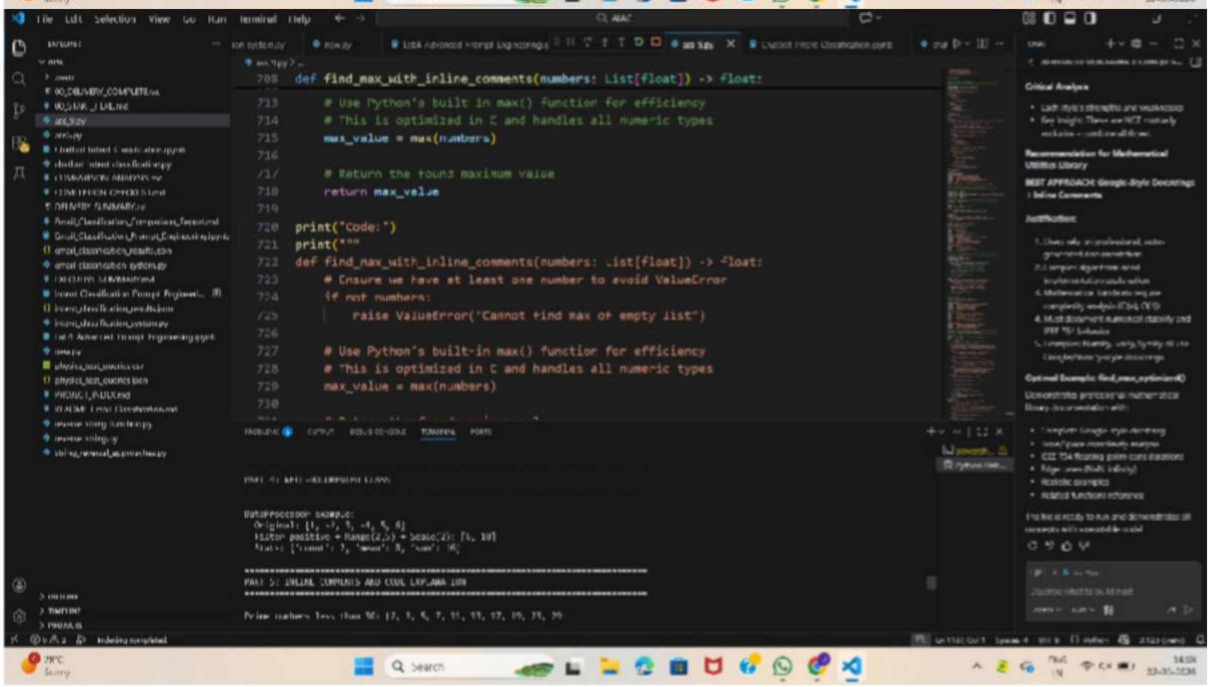

print("\nUSE CASES:")
print(" • Public API functions")
print(" • Library functions")
print(" • Functions with complex parameters")
print(" • Functions that need auto-generated documentation")
print(" • Professional codebases")
print(" • Open-source projects")
print(" • Team-based development")
# DETAILED COMPARISON TABLE
print("\n" + "=" * 80)
print("DETAILED COMPARISON TABLE")
print("=" * 80 + "\n")
comparison_data = [


print("\nDocumentation accessed via help():")
print("-" * 80)
help(find_max_google_style)
print("\nADVANTAGES:")
print(" ✓ Comprehensive and professional")
print(" ✓ Accessible via help(), IDE tooltips, and Sphinx")
print(" ✓ Standardized format across projects")
print(" ✓ Clear parameter and return documentation")
print(" ✓ Usage examples included")
print(" ✓ Error conditions documented")
print(" ✓ Performance notes included")
print(" ✓ Auto-generates beautiful HTML documentation")
print(" ✓ Supports automated documentation extraction")
print("\nDISADVANTAGES:")
print(" ✗ More verbose and time-consuming to write")
print(" ✗ Can be overkill for trivial functions")
print(" ✗ Requires maintenance as code changes")


def find_max_google_style(numbers: List[float]) -> float:
"""
if not numbers:
raise ValueError("Cannot find max of empty list")
return float(max(numbers))
print("Code:")
print("""
def find_max_google_style(numbers: List[float]) -> float:
"""
Find the maximum value in a list of numbers.
This function accepts a list of numeric values and returns the largest
value found. It handles both integers and floating-point numbers.
Args:
numbers: A non-empty list of numeric values (int or float).
Returns:
float: The maximum value from the input list.
Critical Analysis
Recommendation for Mathematical Utilities Library
Justification:


print(" ✗ Poor for API documentation generation")
print("\nUSE CASES:")
print(" • Complex algorithms requiring explanation")
print(" • Implementation details and design decisions")
print(" • Workarounds for platform-specific issues")
print(" • Internal development documentation")
print(" • Explaining optimization rationale")
# APPROACH 3: GOOGLE-STYLE DOCSTRING (COMPREHENSIVE)
print("\n" + "=" * 80)
print("APPROACH 3: GOOGLE-STYLE DOCSTRING (COMPREHENSIVE)")
print("=" * 80 + "\n")
def find_max_google_style(numbers: List[float]) -> float:
"""
Find the maximum value in a list of numbers.
Critical Analysis
Recommendation for Mathematical Utilities Library
Justification:





Problem 2: Consider the following Python function:

def login(user, password, credentials):

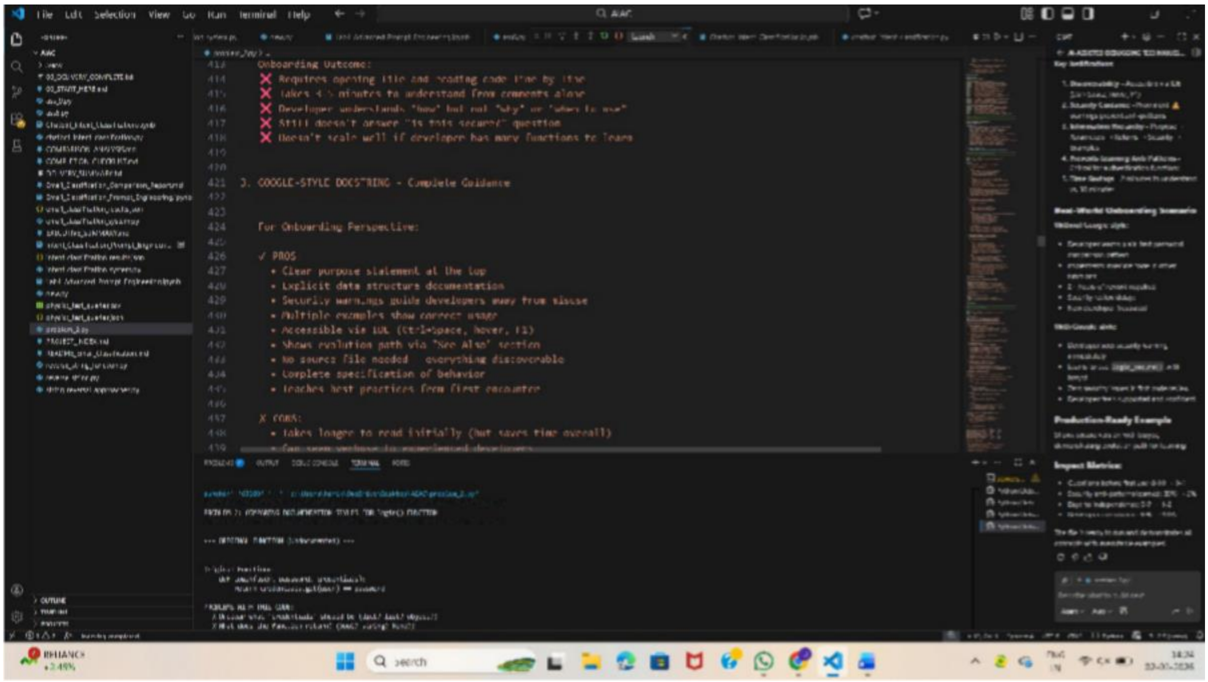
return credentials.get(user) == password

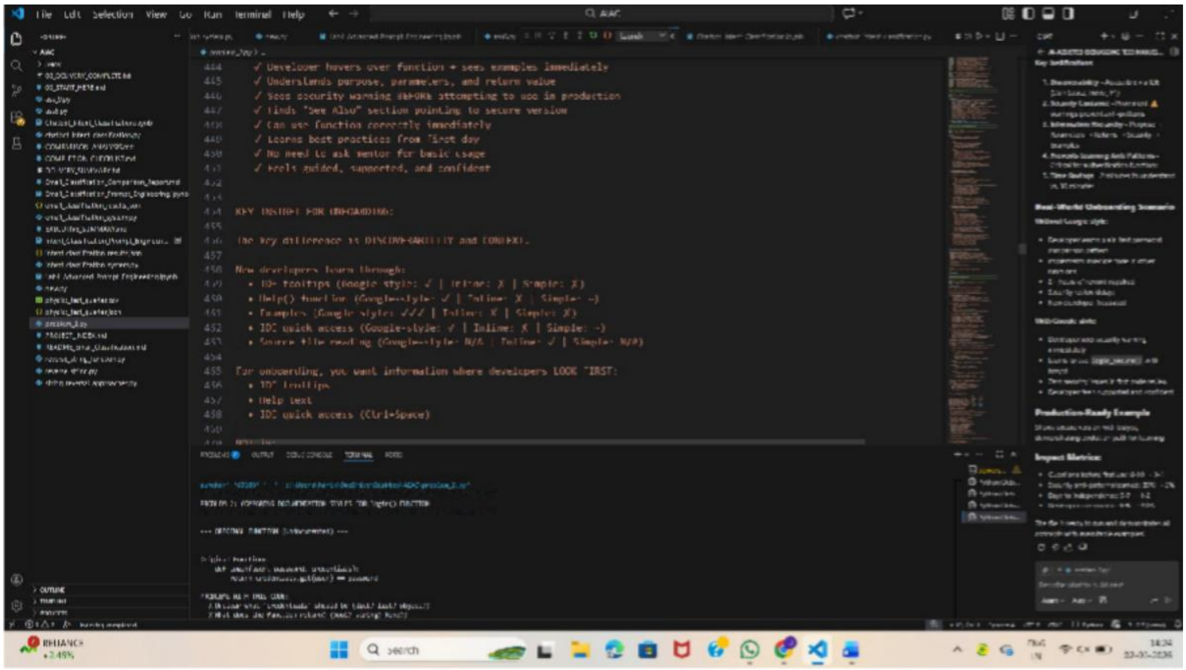
Task:

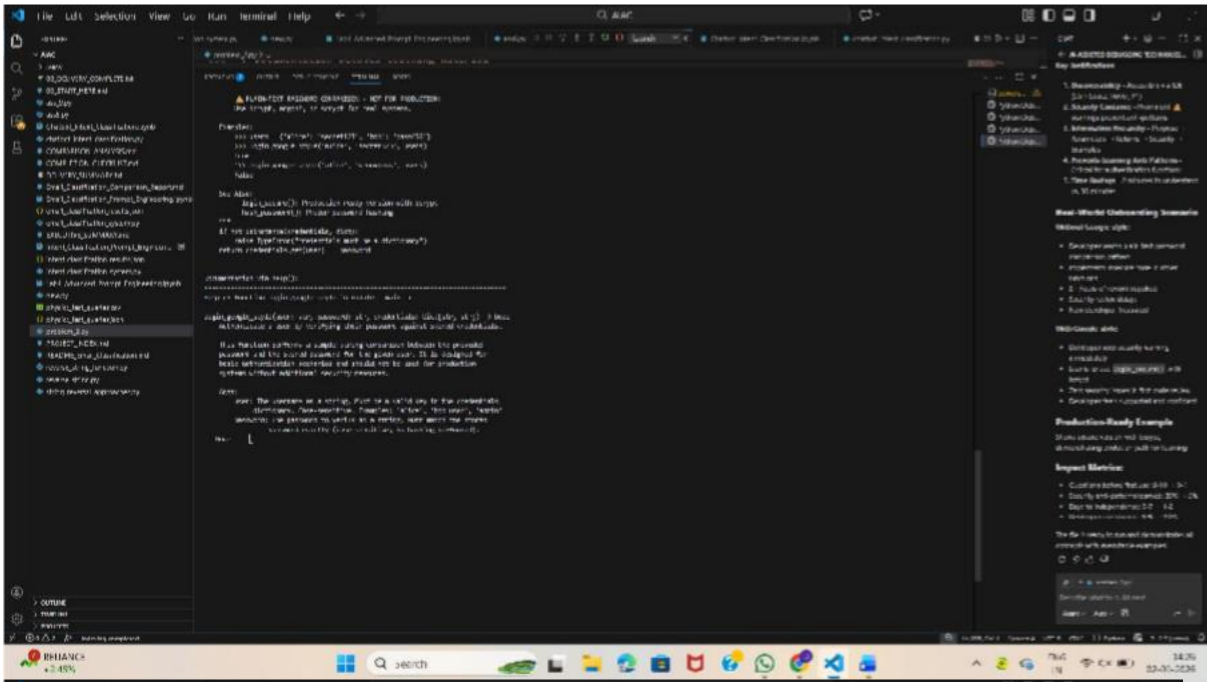
1. Write documentation in all three formats.

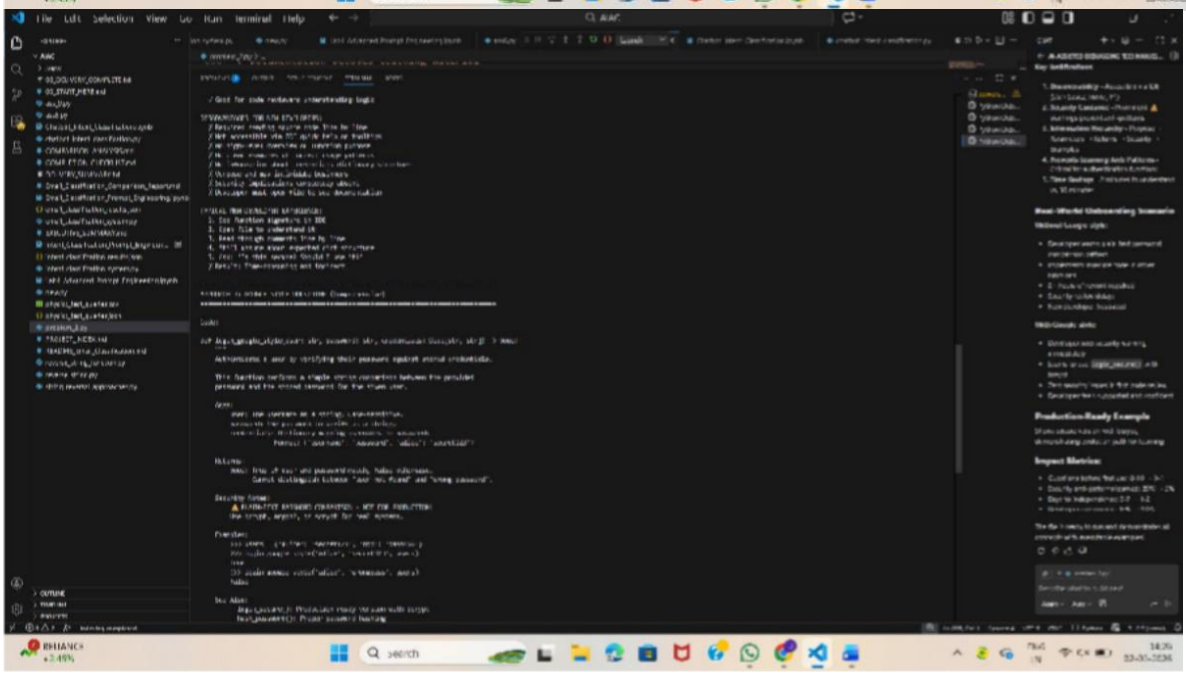
2. Critically compare the approaches.

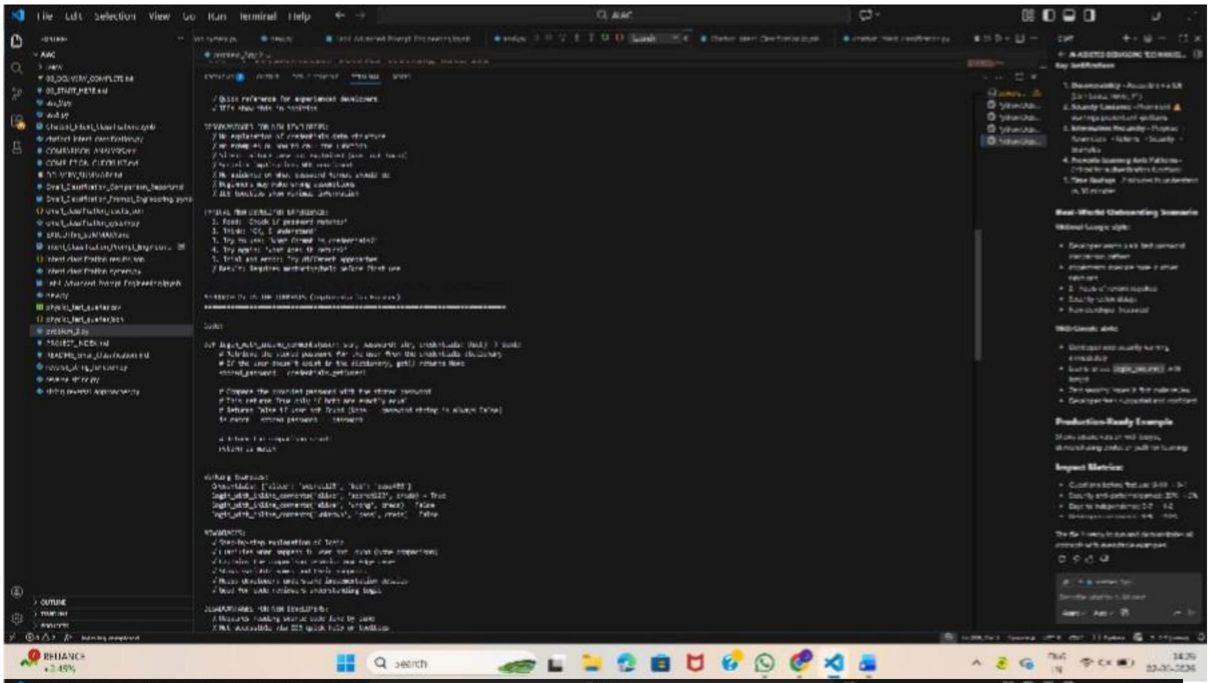
3. Recommend which style would be most helpful for new developers onboarding a project, and justify your choice.

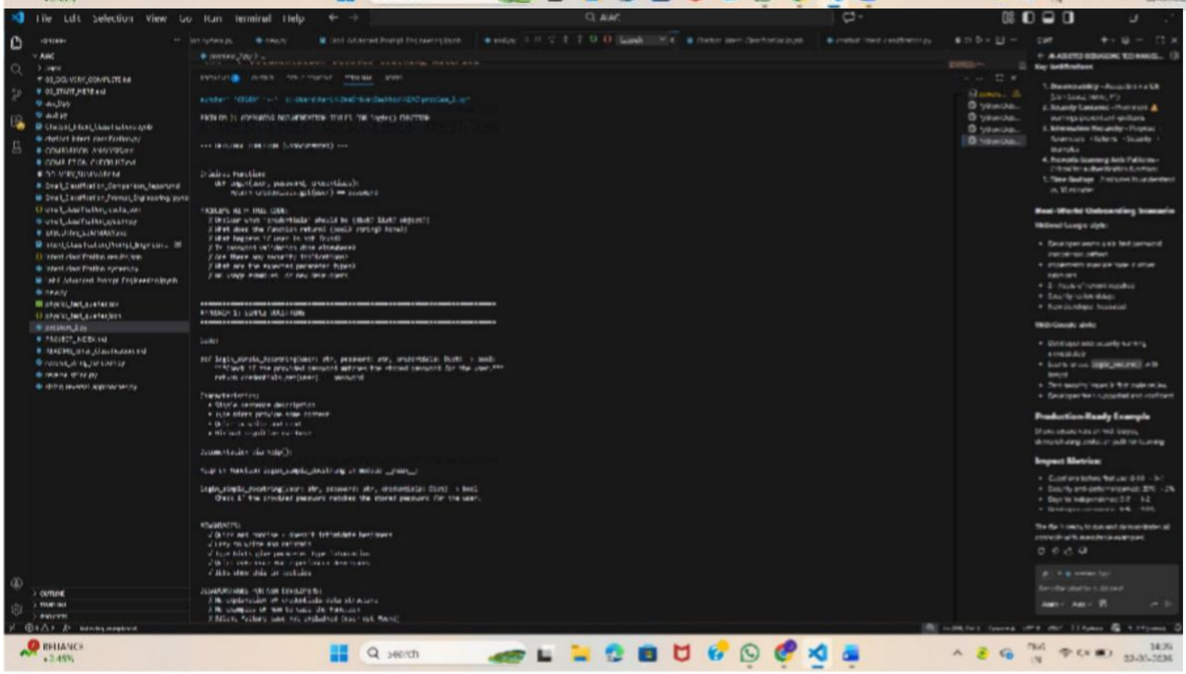


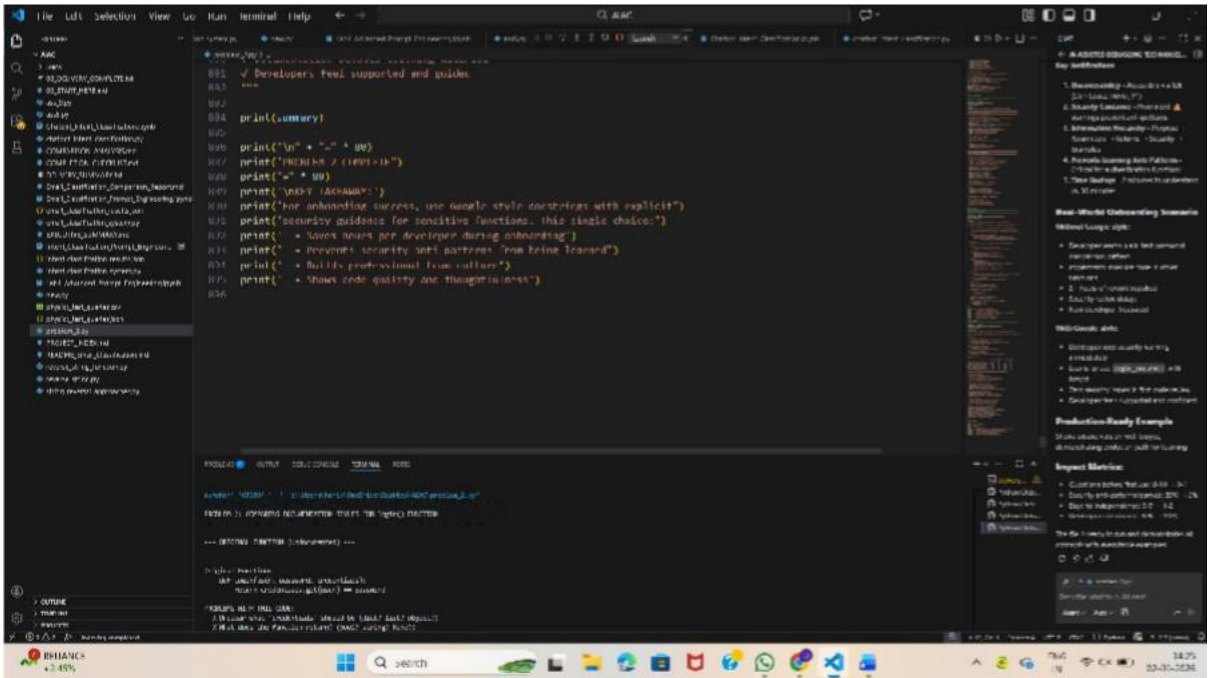


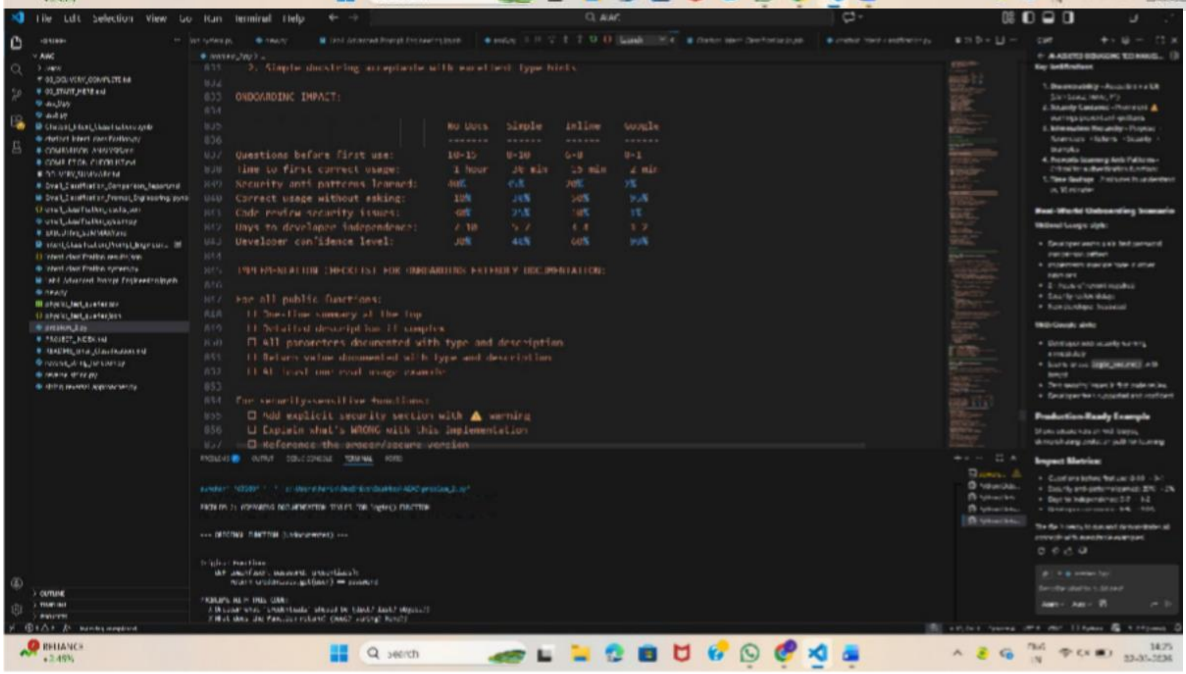

File Edit Selection View Go Run Terminal Help

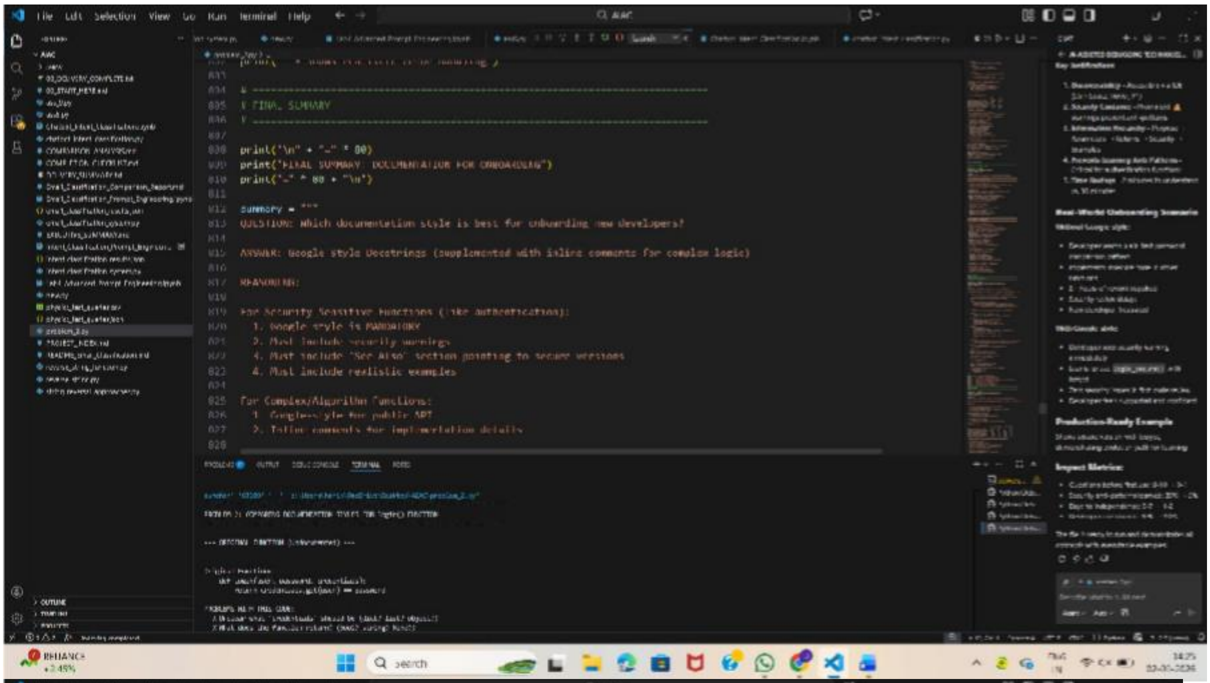
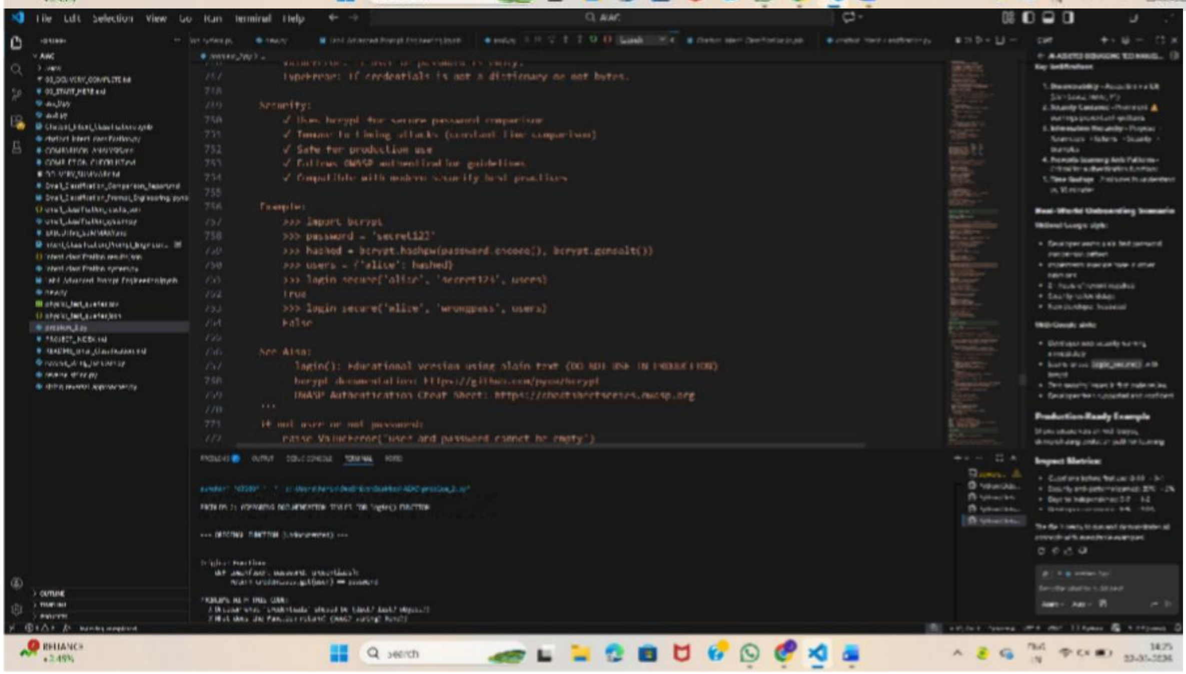

File Edit Selection View Go Run Terminal Help

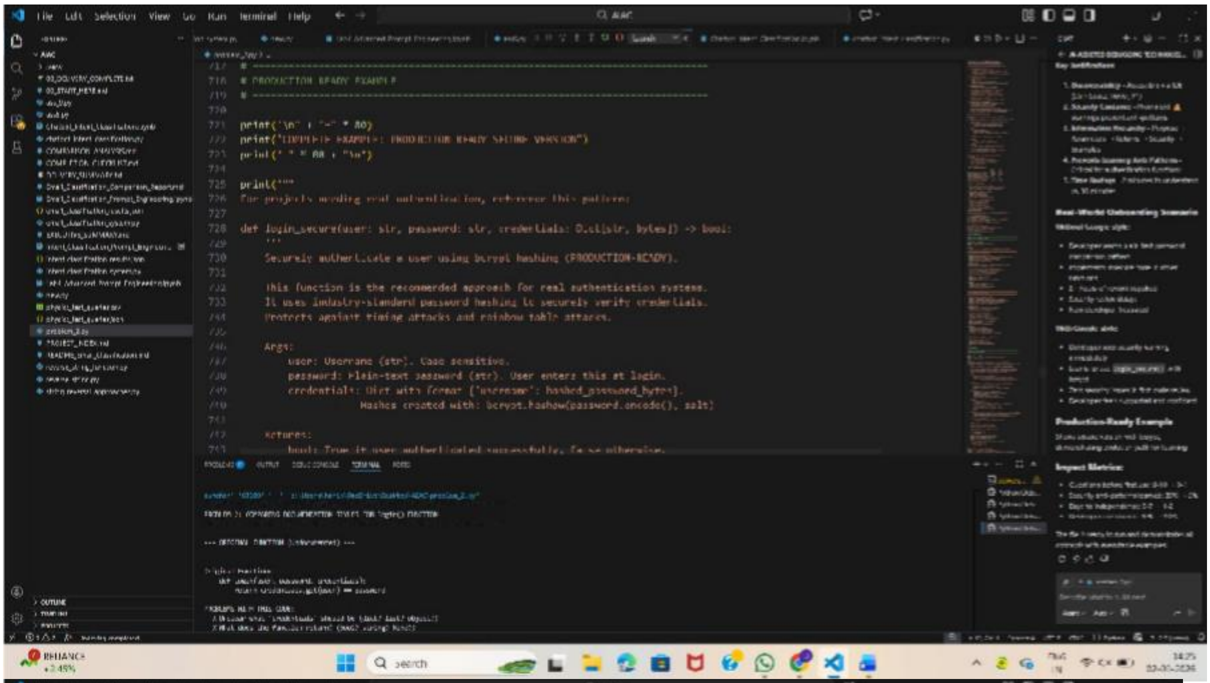


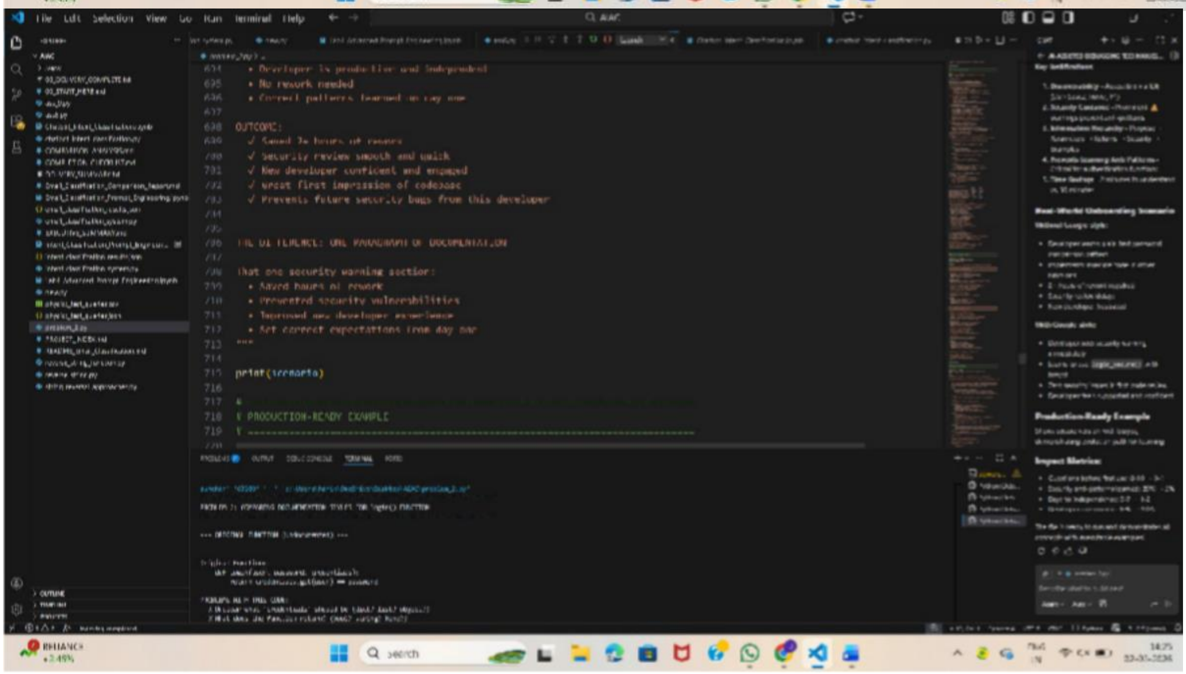


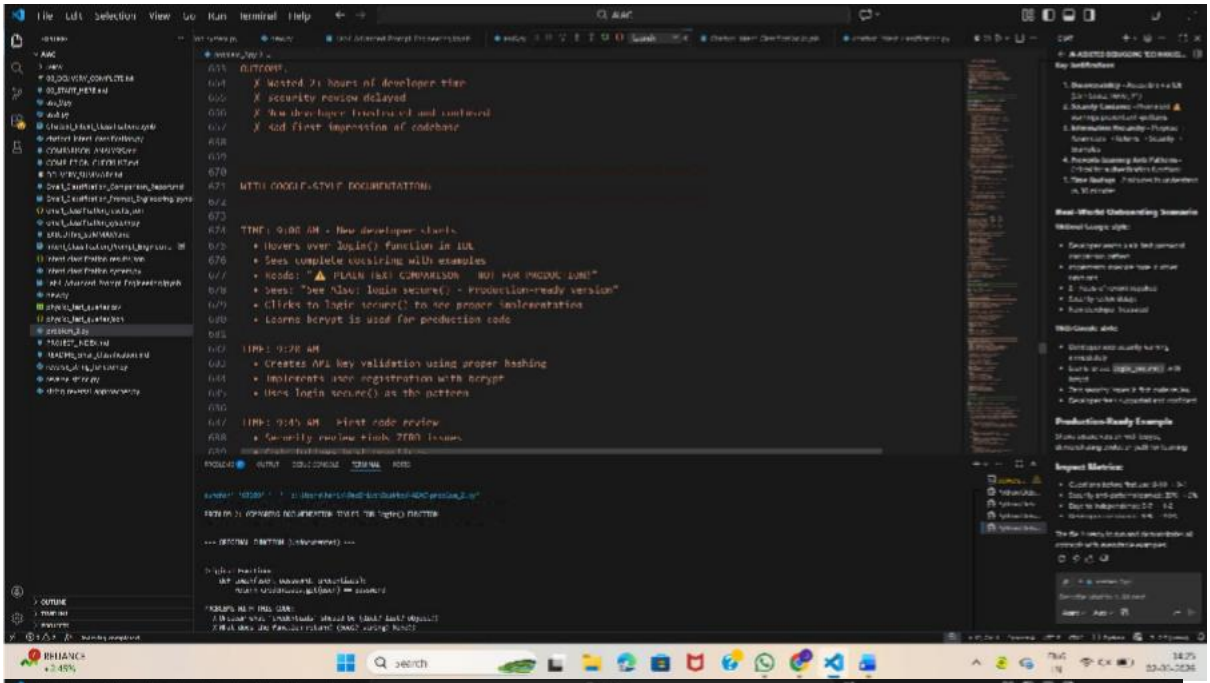

File Edit Selection View Go Run Terminal Help
✓ Developers feel supported and guided
print(summary)
print("=" * 80)
print(" • Saves hours per developer during onboarding")
print(" • Shows code quality and thoughtfulness")

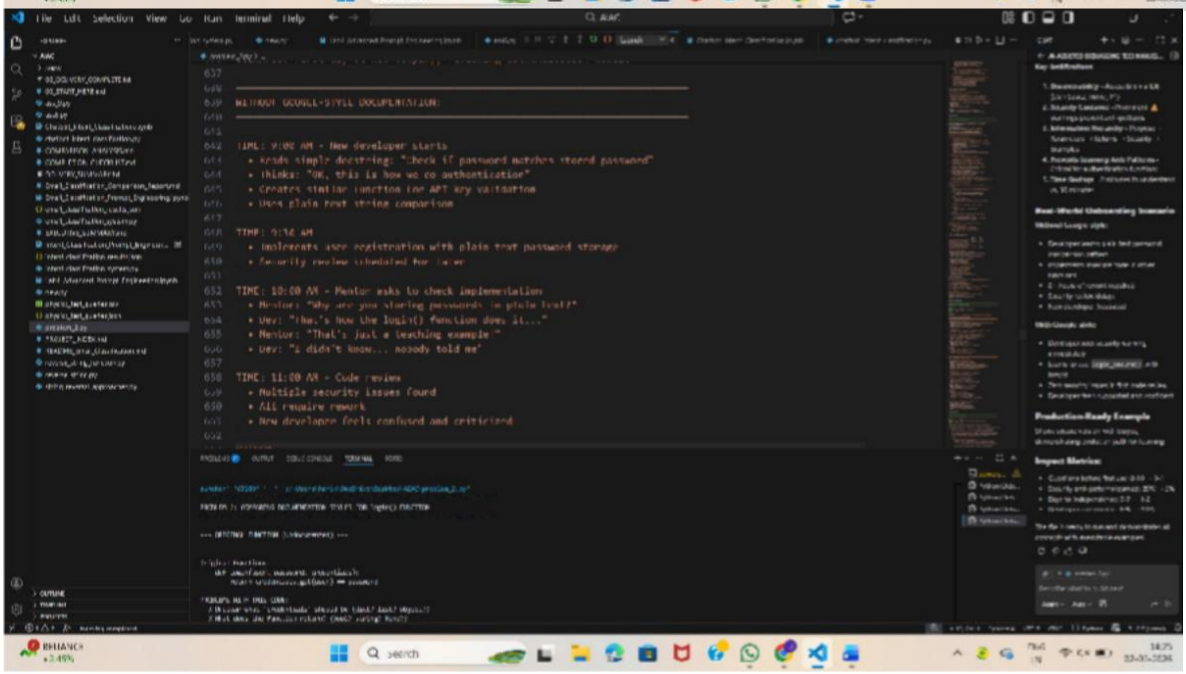

File Edit Selection View Go Run Terminal Help
ONBOARDING IMPACT:
Questions before first use:
Time to first correct usage:
Security anti patterns learned:
Correct usage without asking:
Code review security issues:
Days to developer independence:
Developer confidence level:
For all public functions:
All parameters documented with type and description
Add explicit security section with warning
Explain what's WRONG with this implementation

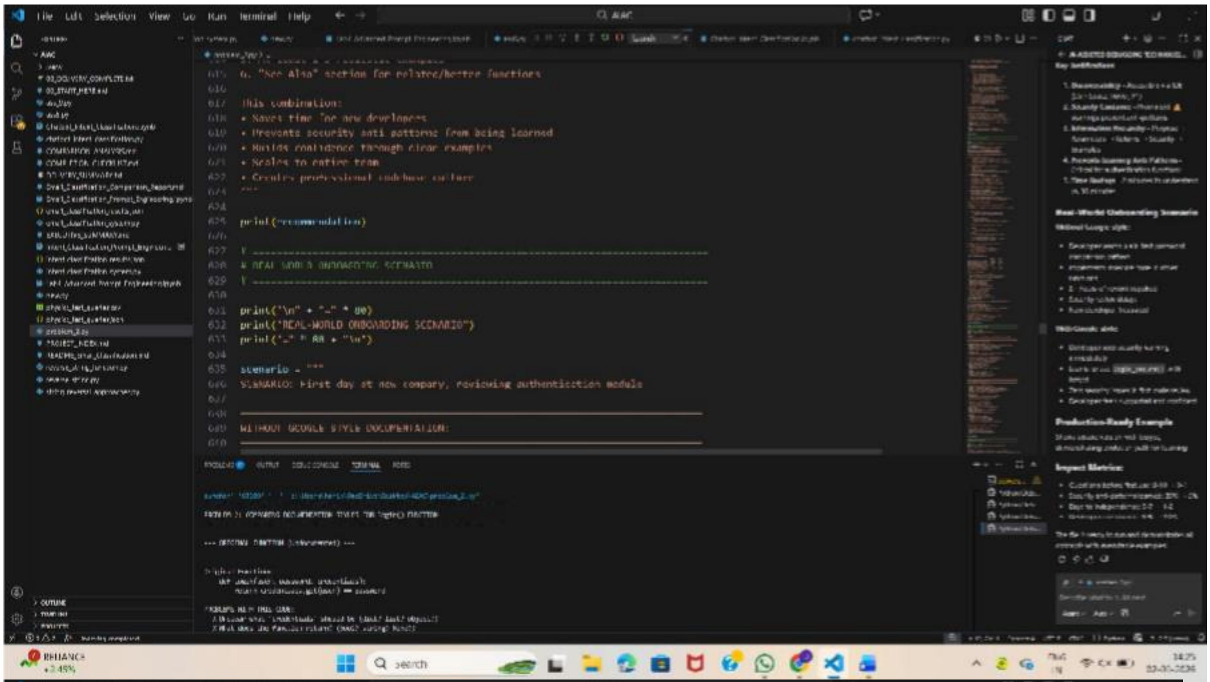
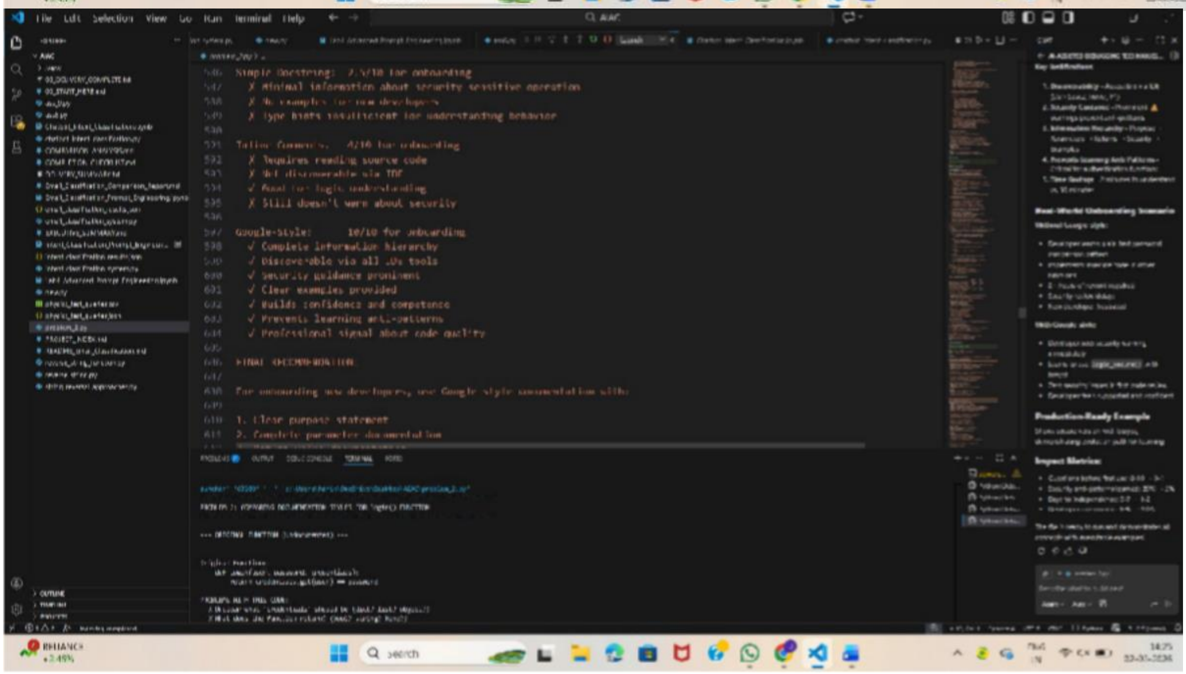

File Edit Selection View Go Run Terminal Help
# FINAL SUMMARY
print("FINAL SUMMARY: DOCUMENTATION FOR ONBOARDING")
summary = """
QUESTION: Which documentation style is best for onboarding new developers?
ANSWER: Google style Docstrings (supplemented with inline comments for complex logic)
REASONING:
For Security Sensitive Functions (like authentication):
1. Google style is MANDATORY
4. Must include realistic examples
For Complex/Algorithm Functions:
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Search

File Edit Selection View Go Run Terminal Help
TypeError: If credentials is not a dictionary or not bytes.
Security:
✓ Safe for production use
Example:
>>> import bcrypt
>>> password = 'secret123'
>>> hashed = bcrypt.hashpw(password.encode(), bcrypt.gensalt())
>>> users = {'alice': hashed}
True
>>> login_secure('alice', 'wrongpass', users)
False
See Also:
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Search

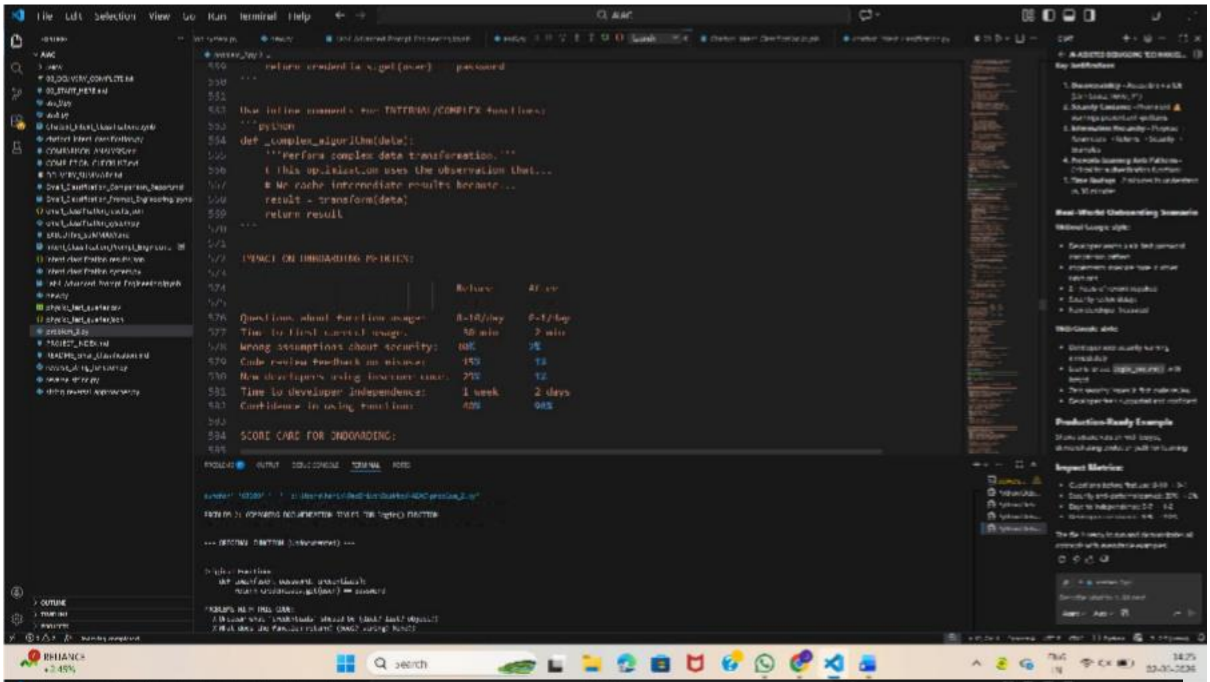

File Edit Selection View Go Run Terminal Help
# PRODUCTION READY EXAMPLE
print("\n" + "=" * 80)
print(" " * 80 + "\n")
print("""
def login_secure(user: str, password: str, credentials: Dict[str, bytes]) -> bool:
Securely authenticate a user using bcrypt hashing (PRODUCTION-READY).
This function is the recommended approach for real authentication systems.
It uses industry-standard password hashing to securely verify credentials.
Protects against timing attacks and rainbow table attacks.
Args:
user: Username (str). Case sensitive.
password: Plain-text password (str). User enters this at login.
credentials: Dict with format {'username': hashed_password_bytes}.
Hashes created with: bcrypt.hashpw(password.encode(), salt)
Returns:
Search

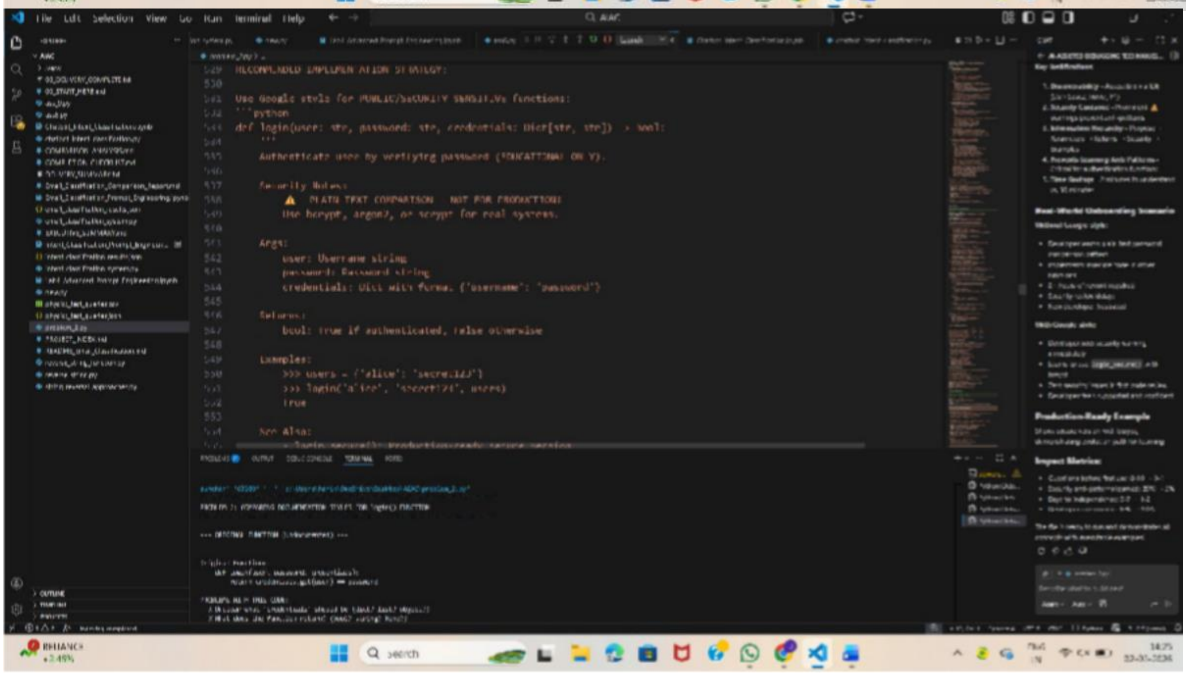

File Edit Selection View Go Run Terminal Help
• No rework needed
OUTCOME:
✓ Security review smooth and quick
✓ New developer confident and engaged
✓ Prevents future security bugs from this developer
That one security warning section:
• Prevented security vulnerabilities
• Set correct expectations from day one
"""
print(scenario)
# PRODUCTION-READY EXAMPLE
Search

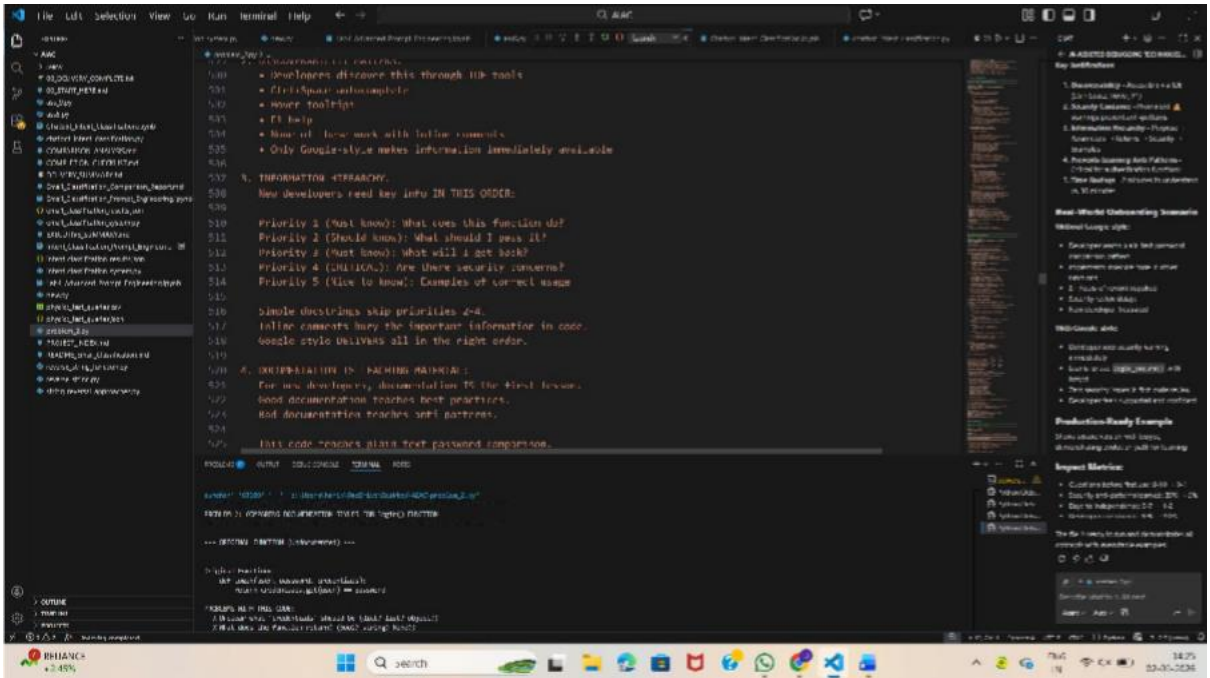

File Edit Selection View Go Run Terminal Help
OUTCOME:
✗ Wasted 2+ hours of developer time
✗ Security review delayed
✗ New developer frustrated and confused
✗ Bad first impression of codebase
WITH GOOGLE-STYLE DOCUMENTATION:
TIME: 9:00 AM - New developer starts
• Hovers over login() function in IDE
• Sees complete docstring with examples
• Reads: "⚠ PLAIN TEXT COMPARISON - NOT FOR PRODUCTION"
• Sees: "See Also: login_secure() - Production-ready version"
• Clicks to login_secure() to see proper implementation
• Learns bcrypt is used for production code
TIME: 9:20 AM
• Creates API key validation using proper hashing
• Implements user registration with bcrypt
• Uses login_secure() as the pattern
TIME: 9:45 AM - First code review
• Security review finds ZERO issues
search

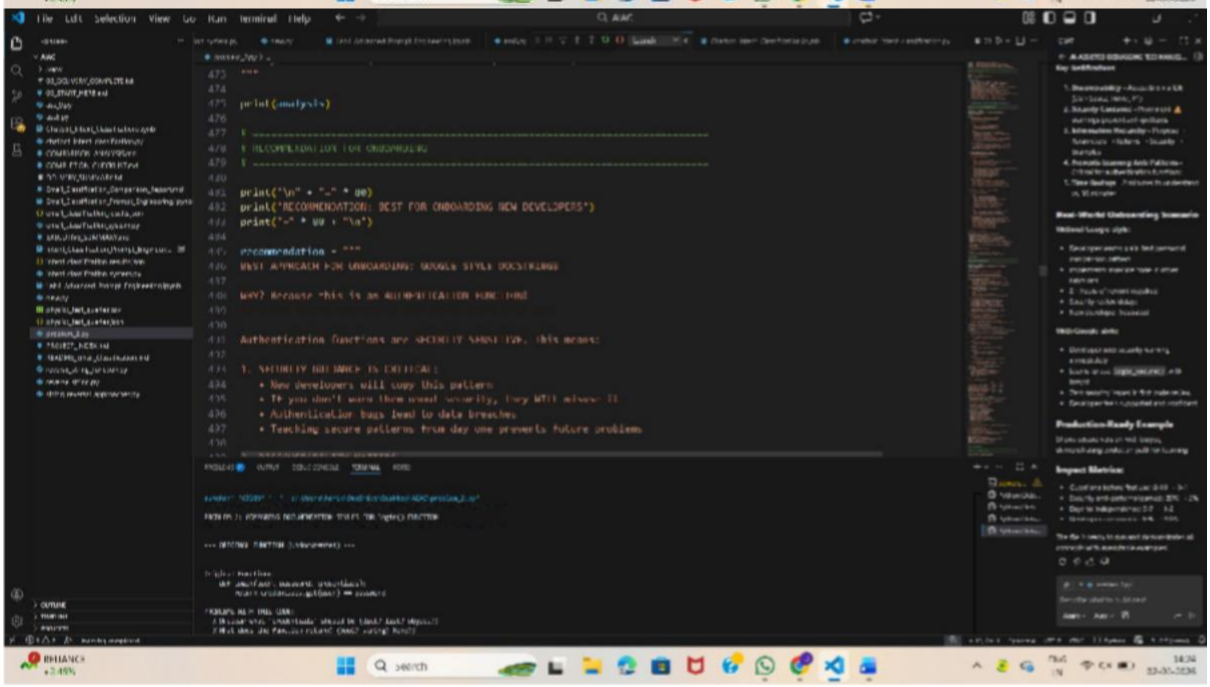

File Edit Selection View Go Run Terminal Help
WITHOUT GOOGLE-STYLE DOCUMENTATION:
TIME: 9:00 AM - New developer starts
• Reads simple docstring: "Check if password matches stored password"
• Thinks: "OK, this is how we do authentication"
• Creates similar function for API key validation
• Uses plain text string comparison
TIME: 9:15 AM
• Implements user registration with plain text password storage
• Security review scheduled for later
TIME: 10:00 AM - Mentor asks to check implementation
• Mentor: "Why are you storing passwords in plain text?"
• Dev: "That's how the login() function does it..."
• Mentor: "That's just a teaching example!"
• Dev: "I didn't know... nobody told me"
TIME: 11:00 AM - Code review
• Multiple security issues found
• All require rework
• New developer feels confused and criticized
search


6. "See Also" section for related/better functions
This combination:
• Saves time for new developers
• Prevents security anti patterns from being learned
• Builds confidence through clear examples
• Scales to entire team
print(recommendation)
print("\n" + "=" * 80)
print("REAL-WORLD ONBOARDING SCENARIO")
scenario = """
SCENARIO: First day at new company, reviewing authentication module
WITHOUT GOOGLE STYLE DOCUMENTATION:
Real-World Onboarding Scenario
Production-Ready Example
Impact Metrics:

Simple Docstring:
✗ Minimal information about security sensitive operation
✗ Type hints insufficient for understanding behavior
✗ Requires reading source code
✗ Still doesn't warn about security
Google-Style: 10/10 for onboarding
✓ Complete information hierarchy
✓ Security guidance prominent
✓ Clear examples provided
✓ Builds confidence and competence
✓ Prevents learning anti-patterns
✓ Professional signal about code quality
1. Clear purpose statement
Real-World Onboarding Scenario
Production-Ready Example
Impact Metrics:


def _complex_algorithm(data):
"""Perform complex data transformation."""
# We cache intermediate results because...
result = transform(data)
return result
Time to developer independence: 1 week 2 days
SCORE CARD FOR ONBOARDING:


def login(user: str, password: str, credentials: Dict[str, str]) -> bool:
Authenticate user by verifying password
Use bcrypt, argon2, or scrypt for real systems.
Args:
user: Username string
password: Password string
Returns:
bool: True if authenticated, False otherwise
Examples:
See Also:





Problem 3: Calculator (Automatic Documentation Generation)

Task: Design a Python module named calculator.py and demonstrate automatic documentation generation.

Instructions:

1. Create a Python module calculator.py that includes the following functions, each written with appropriate docstrings:

o add(a, b) – returns the sum of two numbers

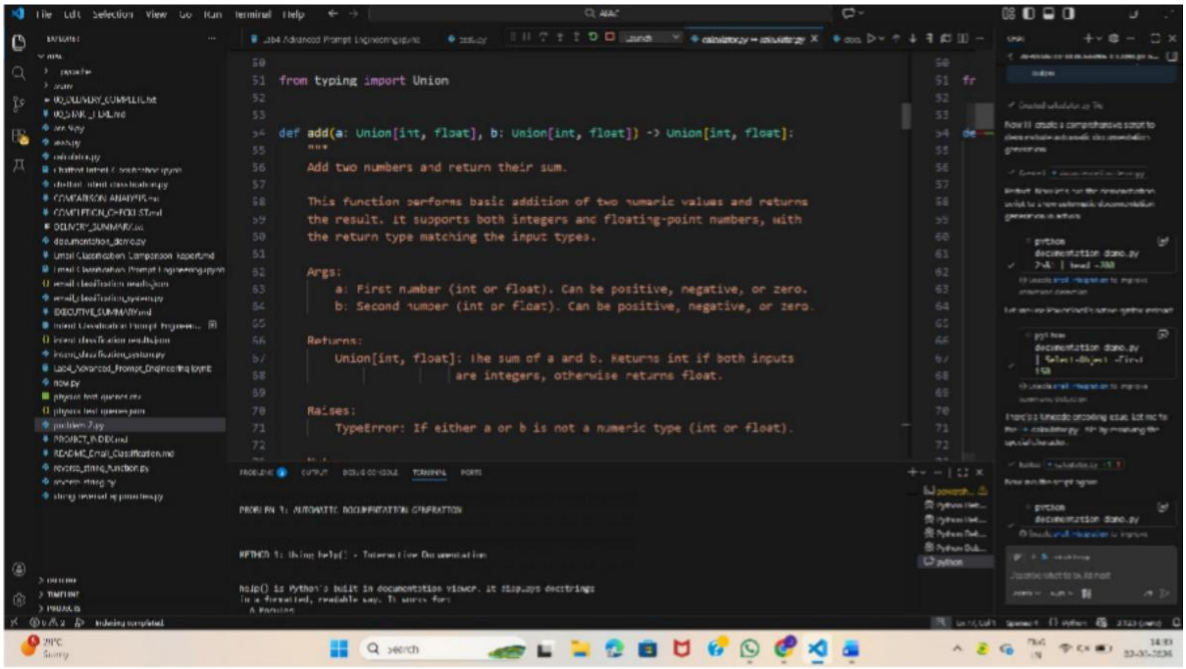
o subtract(a, b) – returns the difference of two numbers

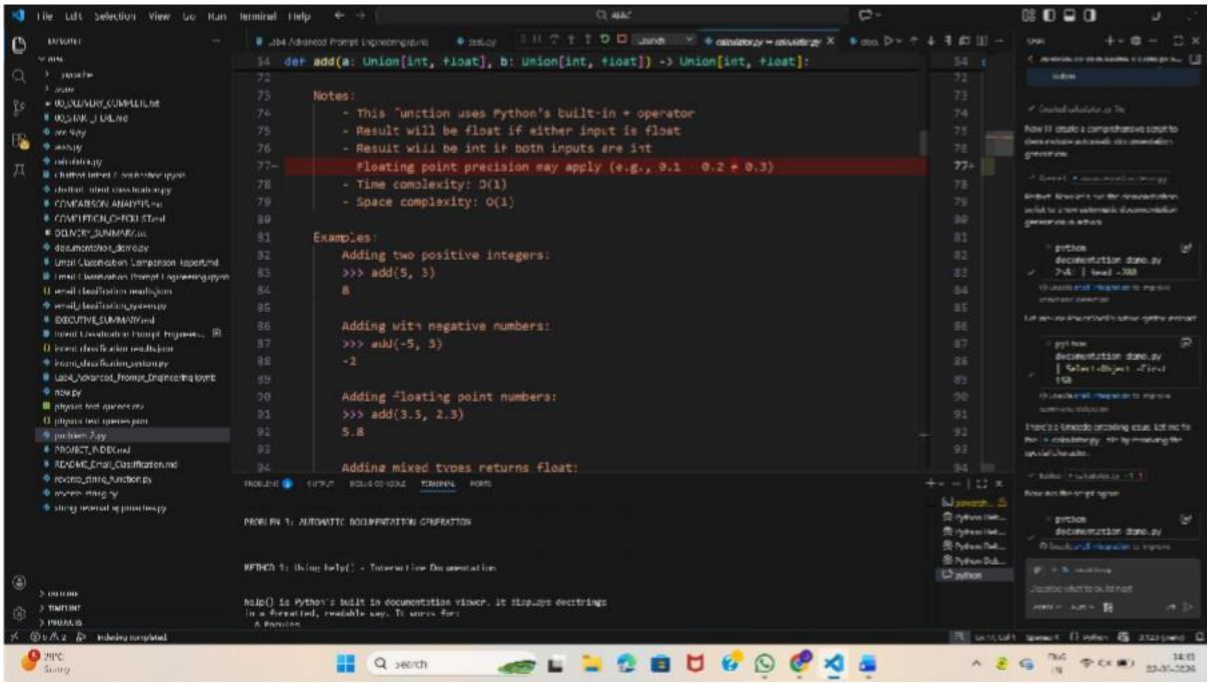
o multiply(a, b) – returns the product of two numbers

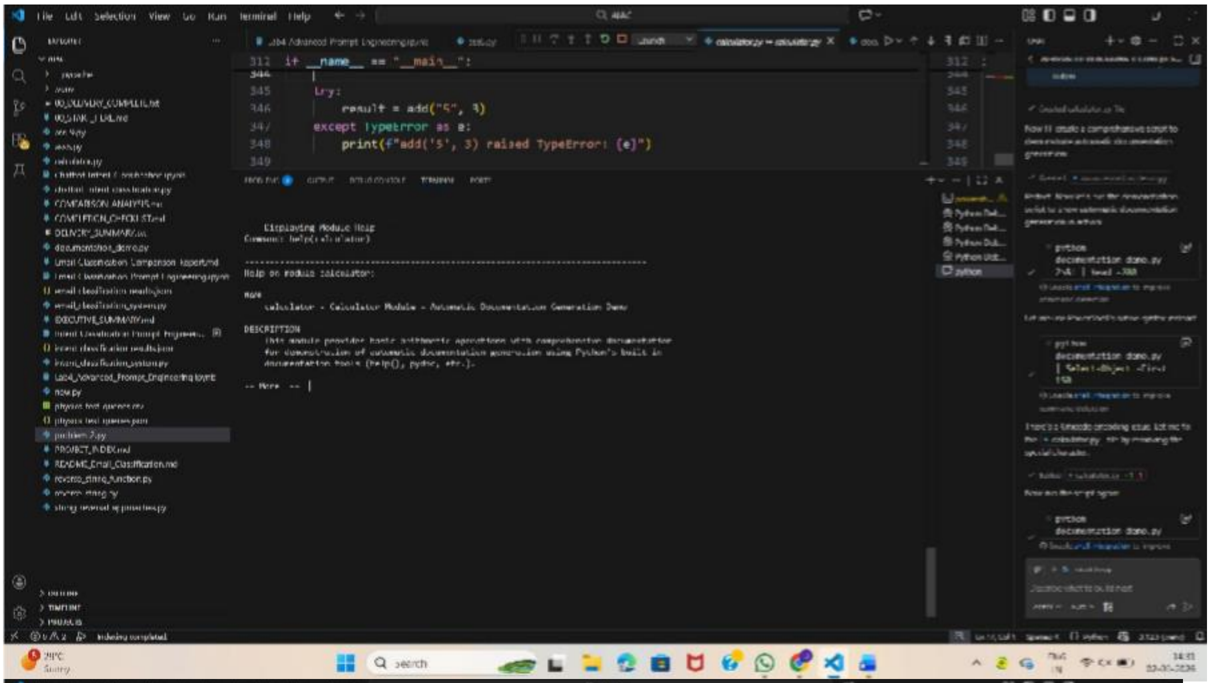
o divide(a, b) – returns the quotient of two numbers

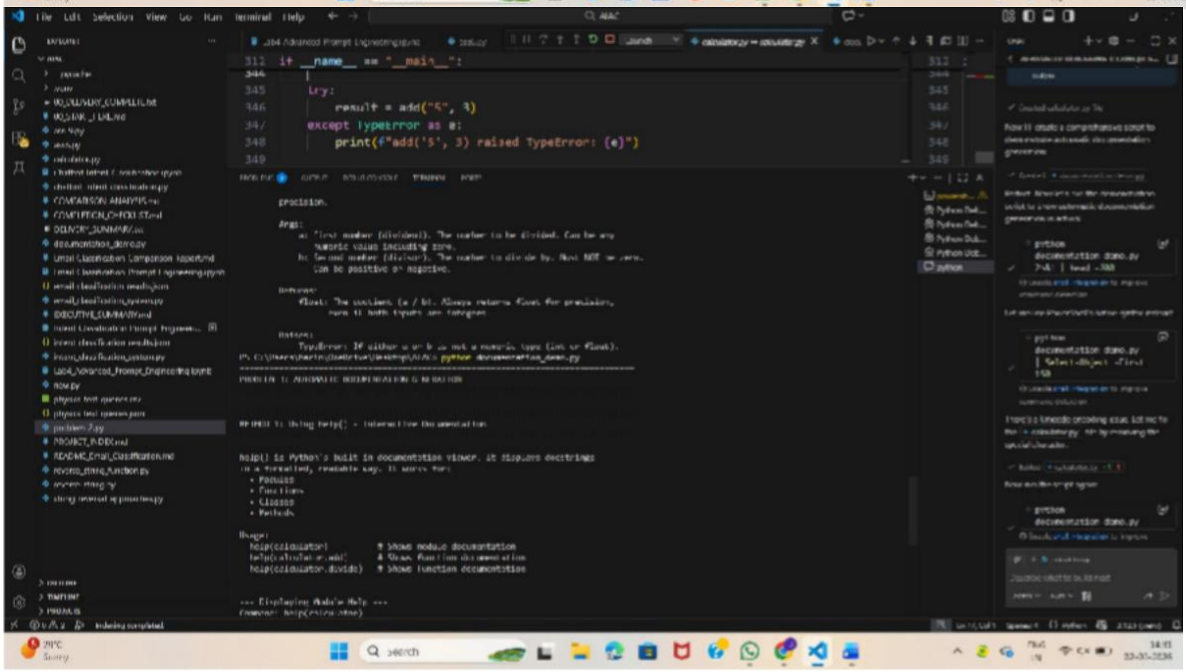
2. Display the module documentation in the terminal using Python's documentation tools.

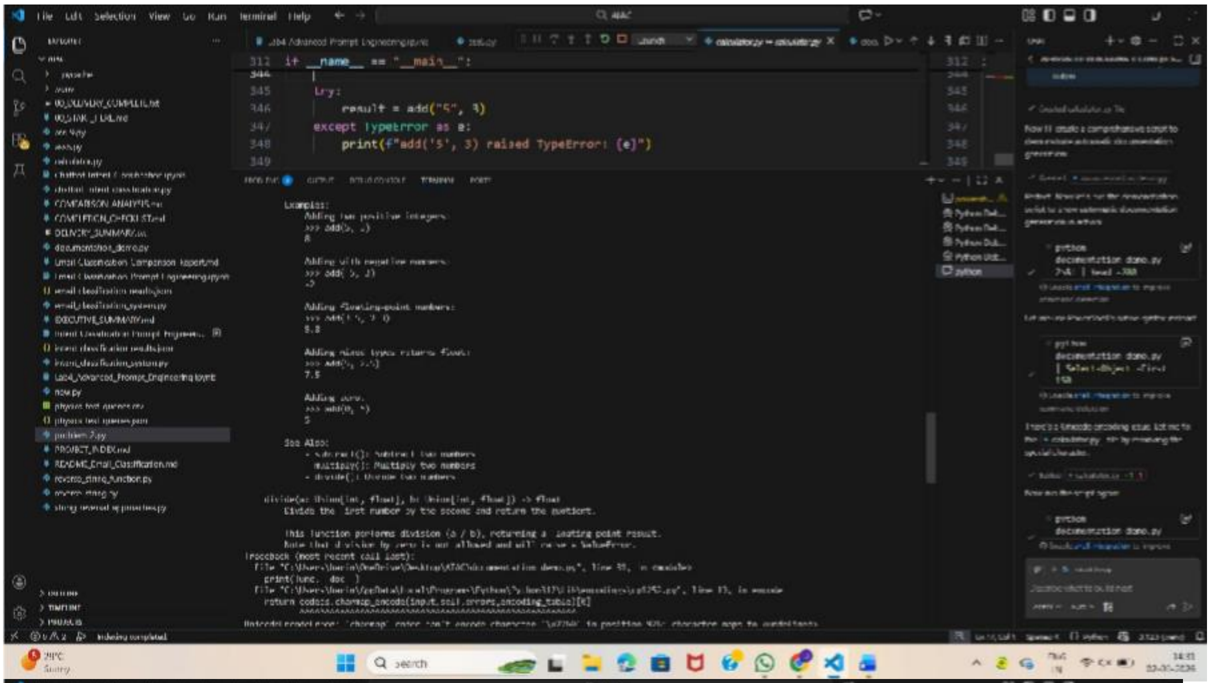
3. Generate and export the module documentation in HTML format using the pydoc utility, and open the generated HTML file in a web browser to verify the output

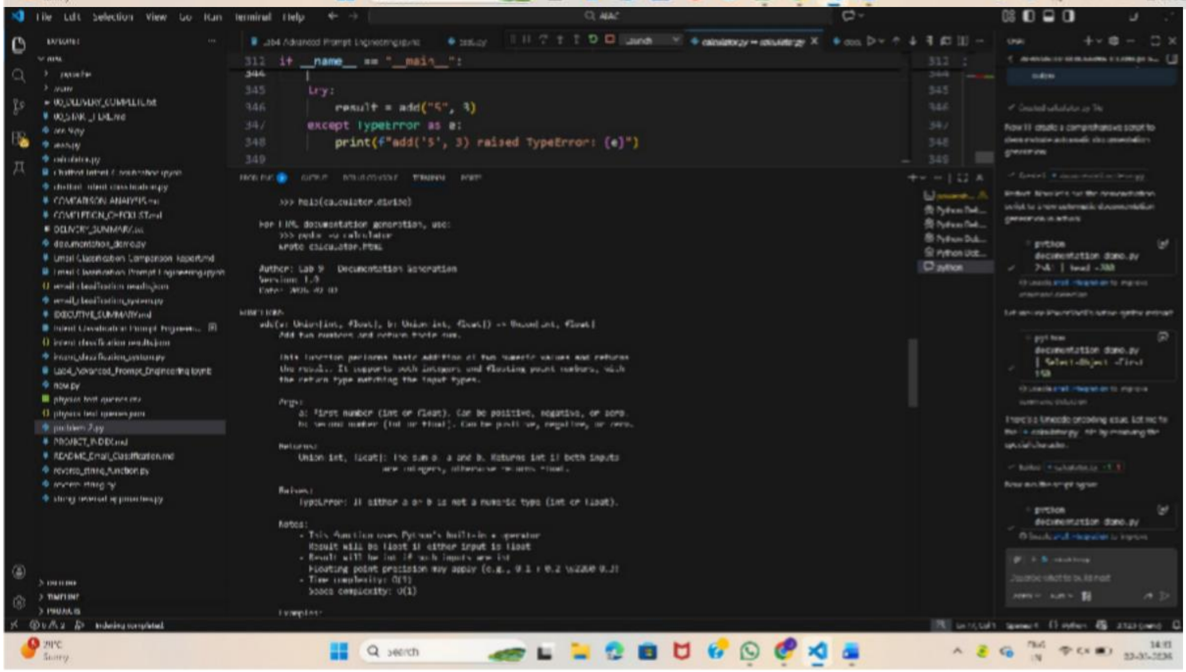


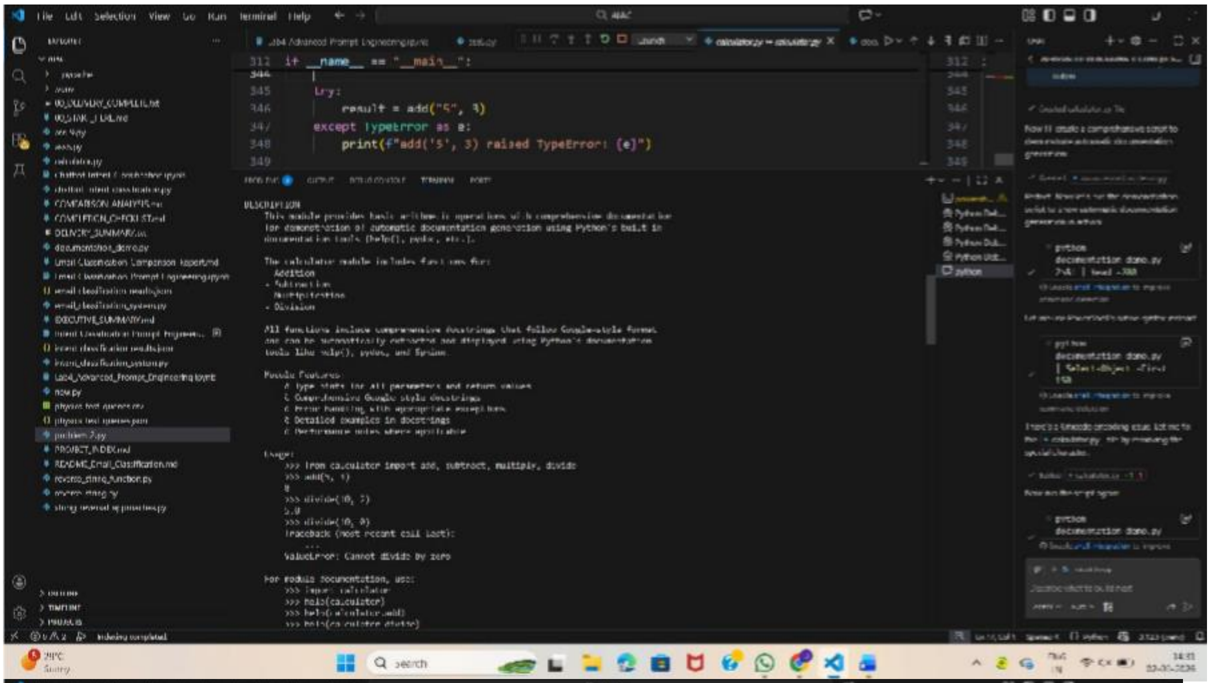

def add(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
Notes:
- This function uses Python's built-in + operator
- Result will be float if either input is float
- Result will be int if both inputs are int
- Floating point precision may apply (e.g., 0.1 + 0.2 ≠ 0.3)
- Time complexity: O(1)
- Space complexity: O(1)
Examples:
Adding two positive integers:
>>> add(5, 3)
8
Adding with negative numbers:
>>> add(-5, 3)
-2
Adding floating point numbers:
>>> add(3.5, 2.3)
5.8
Adding mixed types returns float:
PROBLEM 3: AUTOMATIC DOCUMENTATION GENERATION
METHOD 1: Using help() - Interactive Documentation
help() is Python's built in documentation viewer. It displays docstrings
in a formatted, readable way.


if __name__ == "__main__":
try:
result = add("5", 3)
except TypeError as e:
print(f"add('5', 3) raised TypeError: {e}")
Help on module calculator:
DESCRIPTION
-- More --

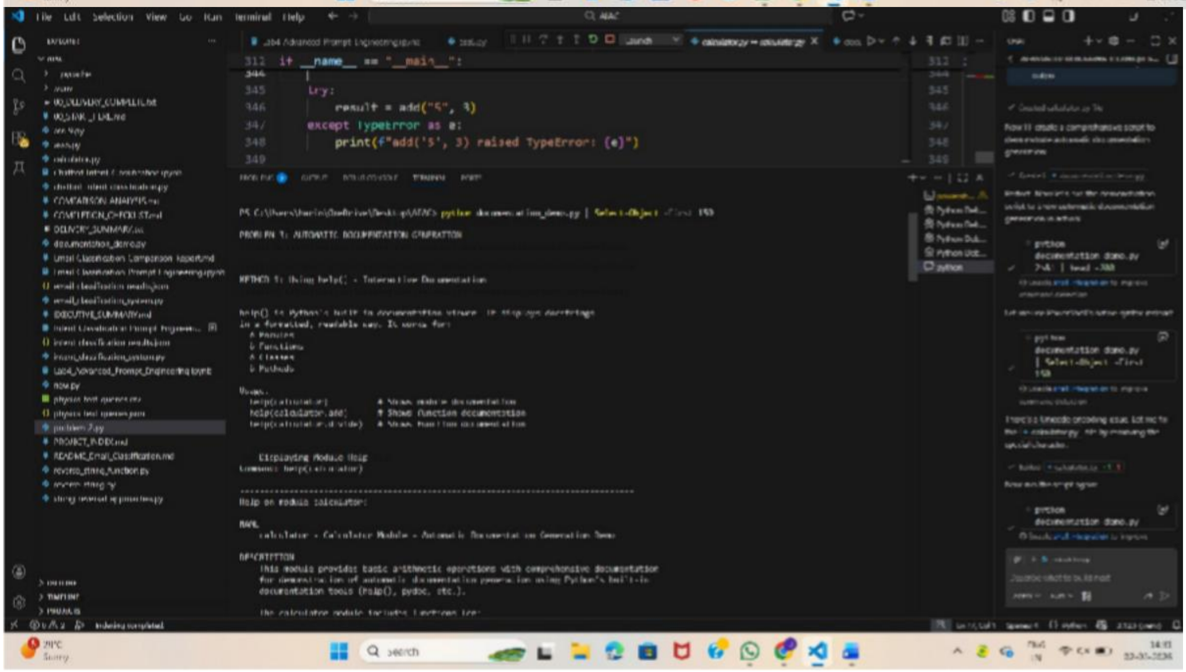

try:
result = add("5", 3)
except TypeError as e:
print(f"add('5', 3) raised TypeError: {e}")
precision.
Args:
Can be positive or negative.
Usage:
help(calculator)
help(calculator.divide)


if __name__ == "__main__":
try:
result = add("5", 3)
except TypeError as e:
print(f"add('5', 3) raised TypeError: {e}")

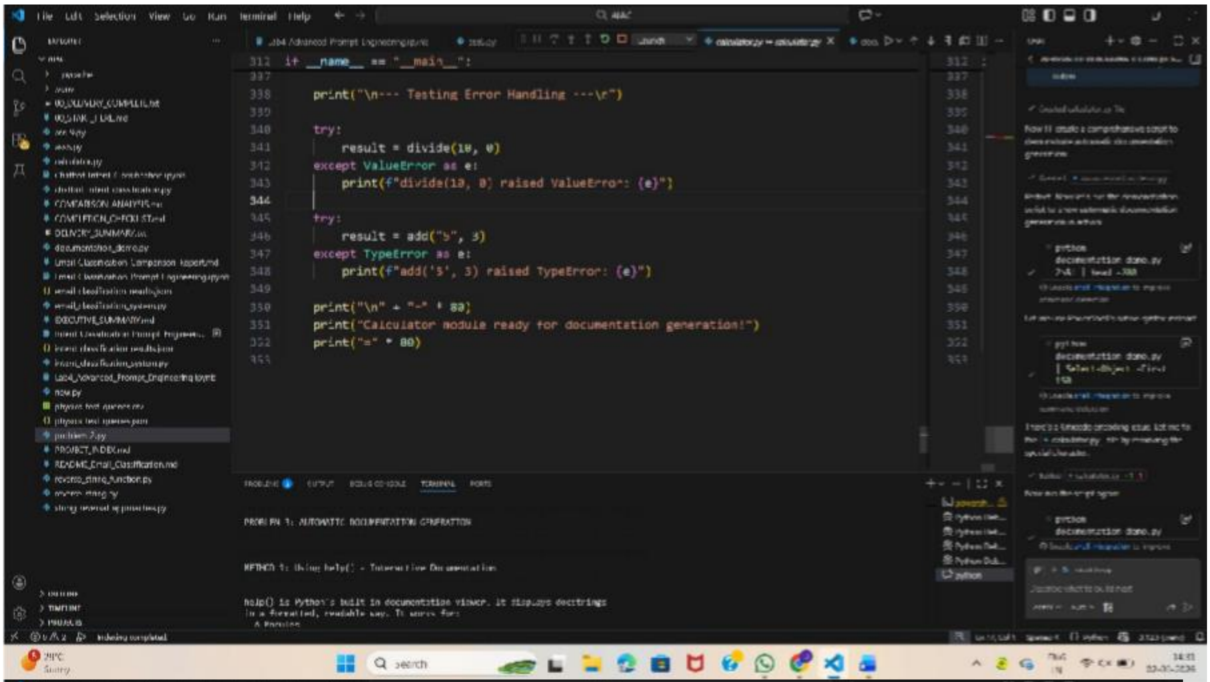

if __name__ == "__main__":
try:
result = add("5", 3)
except TypeError as e:
print(f"add('5', 3) raised TypeError: {e}")

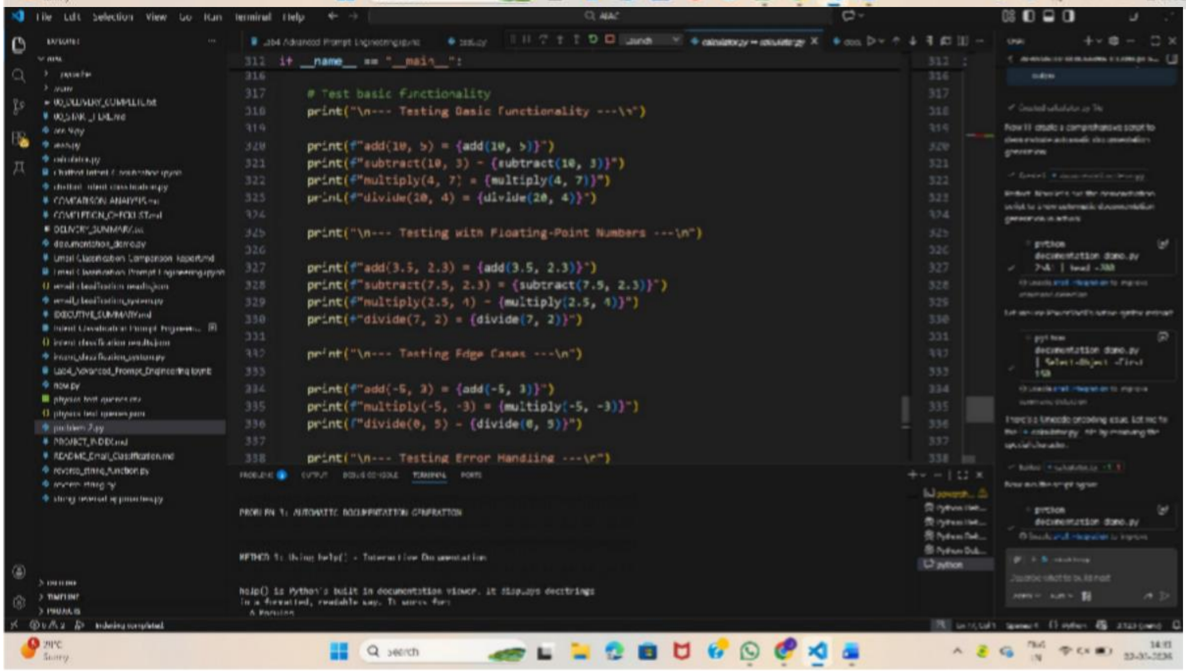


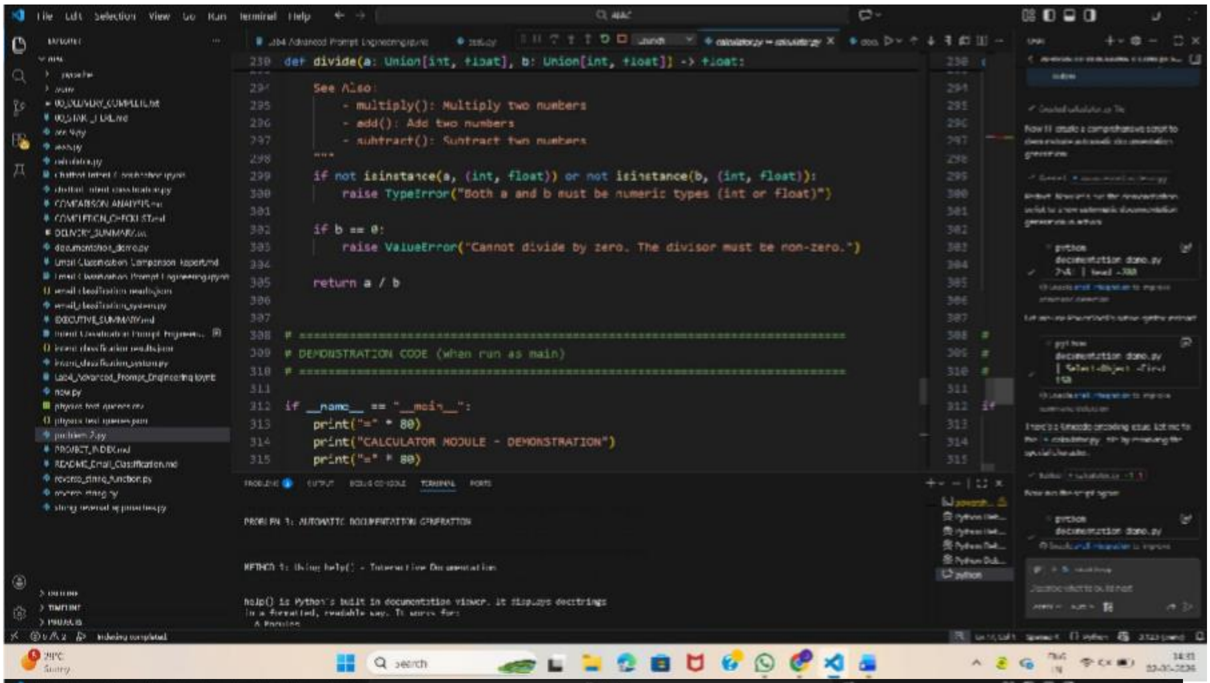
if __name__ == "__main__":
try:
result = add("5", 3)
except TypeError as e:
print(f"add('5', 3) raised TypeError: {e}")
DESCRIPTION
The calculator module includes functions for:
Addition
- Division
>>> from calculator import add, subtract, multiply, divide
>>> divide(10, 0)
Traceback (most recent call last):
ValueError: Cannot divide by zero
For module documentation, use:
>>> help(calculator)
Search

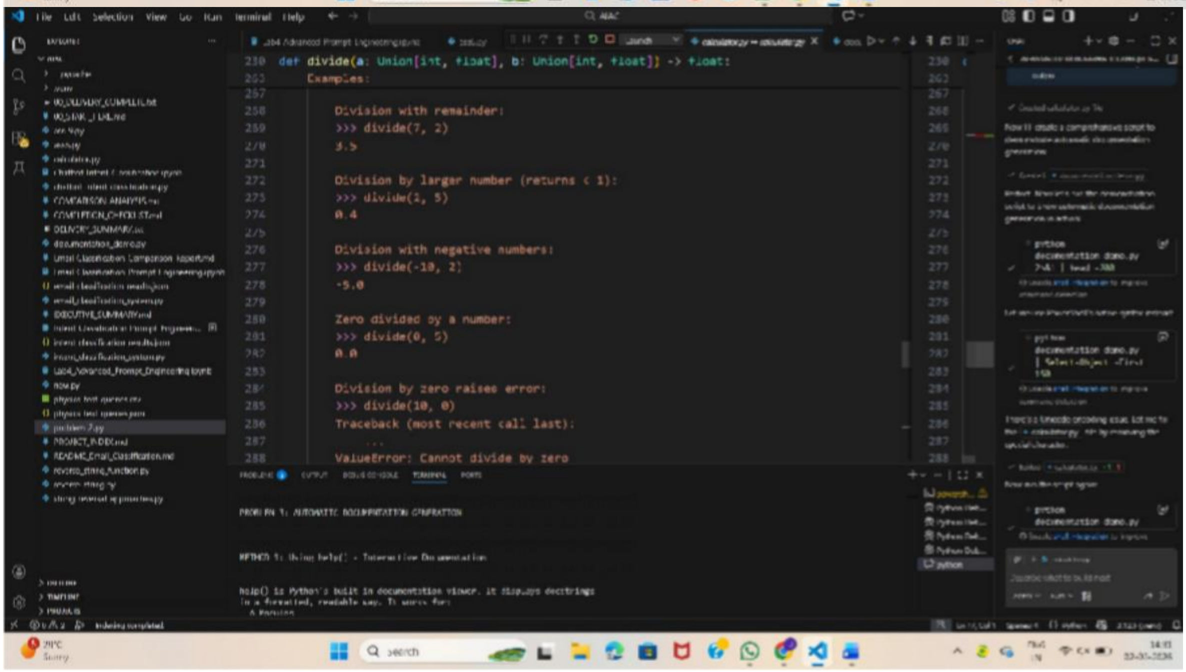


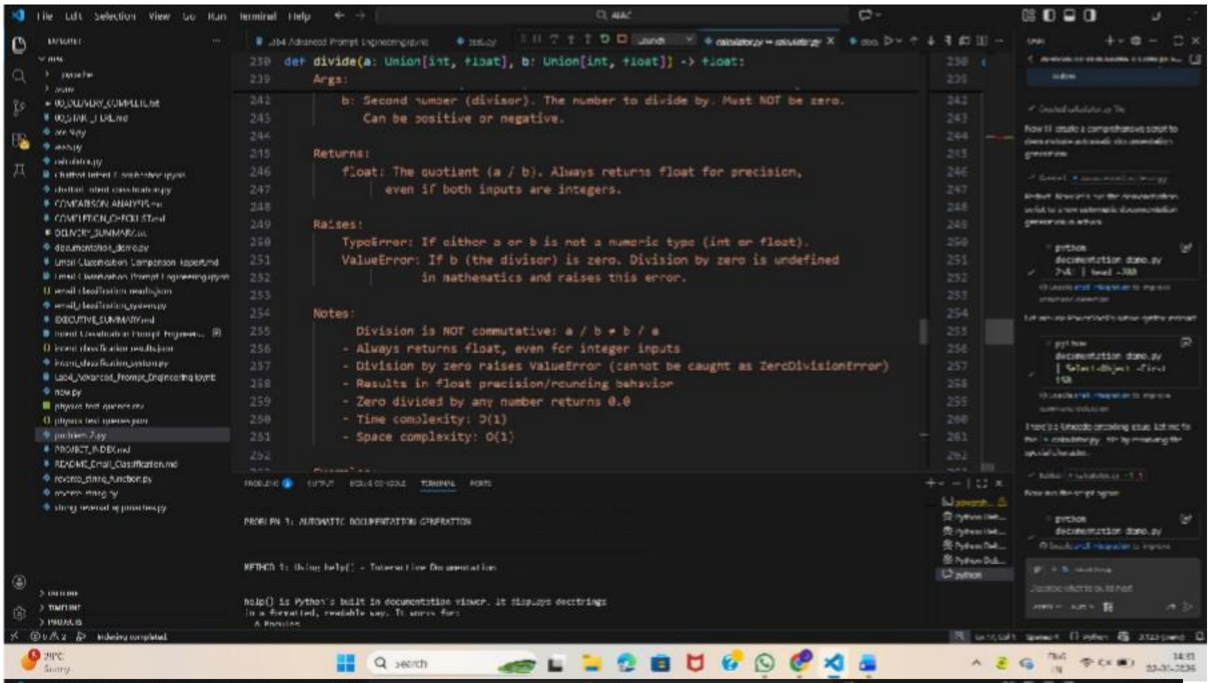
if __name__ == "__main__":
try:
result = add("5", 3)
except TypeError as e:
print(f"add('5', 3) raised TypeError: {e}")
help(calculator.add)
Show function documentation
Help on module calculator:
This module provides basic arithmetic operations with comprehensive documentation
documentation tools (help(), pydoc, etc.).
Search

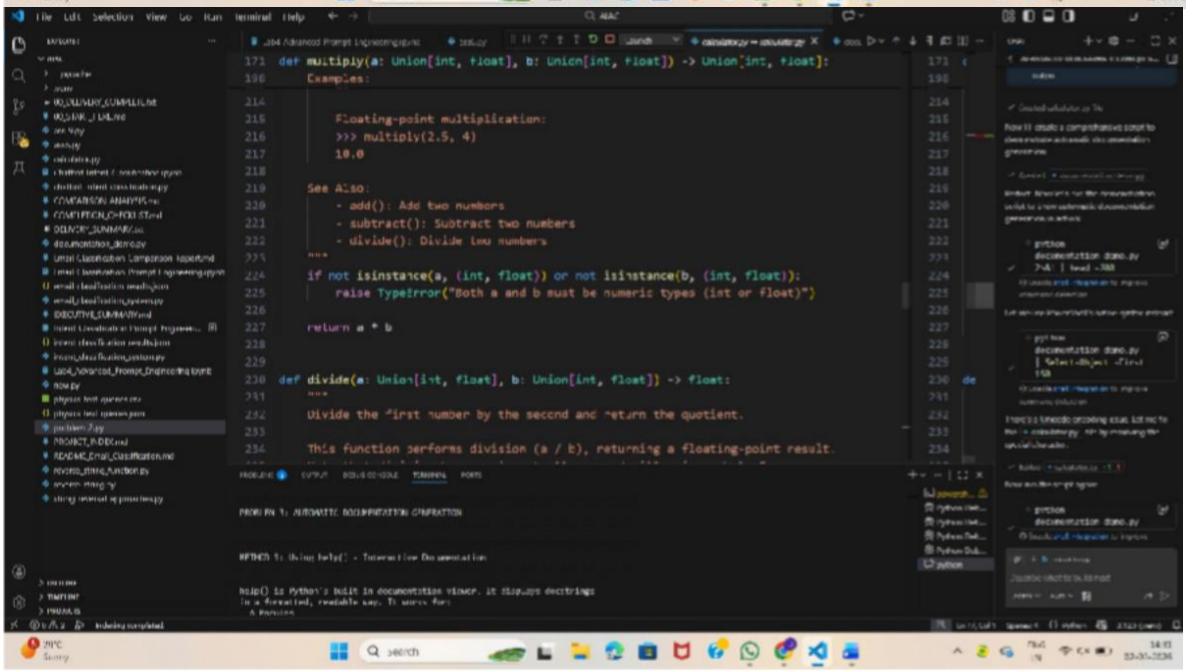

if __name__ == "__main__":
print("\n--- Testing Error Handling ---\n")
try:
result = divide(10, 0)
except ValueError as e:
print(f"divide(10, 0) raised ValueError: {e}")
try:
result = add("5", 3)
except TypeError as e:
print(f"add('5', 3) raised TypeError: {e}")
print("\n" + "=" * 80)
print("Calculator module ready for documentation generation!")
print("=" * 80)
PROBLEM 3: AUTOMATIC DOCUMENTATION GENERATION
METHOD 1: Using help() - Interactive Documentation
help() is Python's built-in documentation viewer. It displays docstrings
in a formatted, readable way.

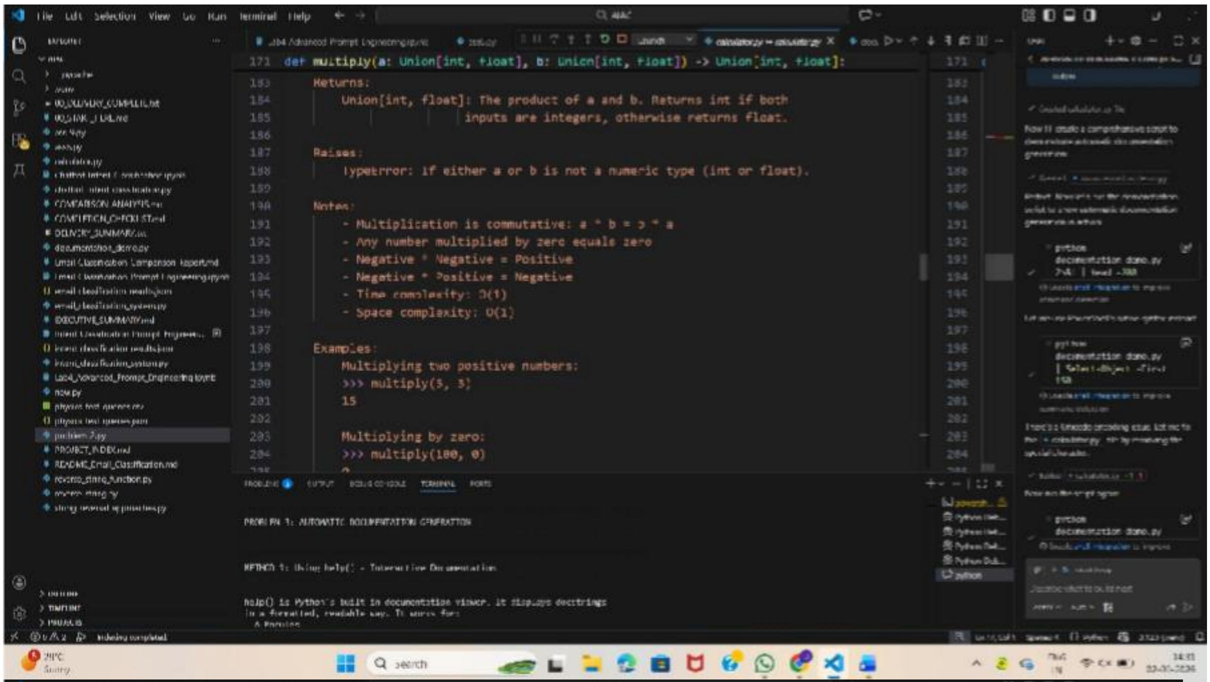
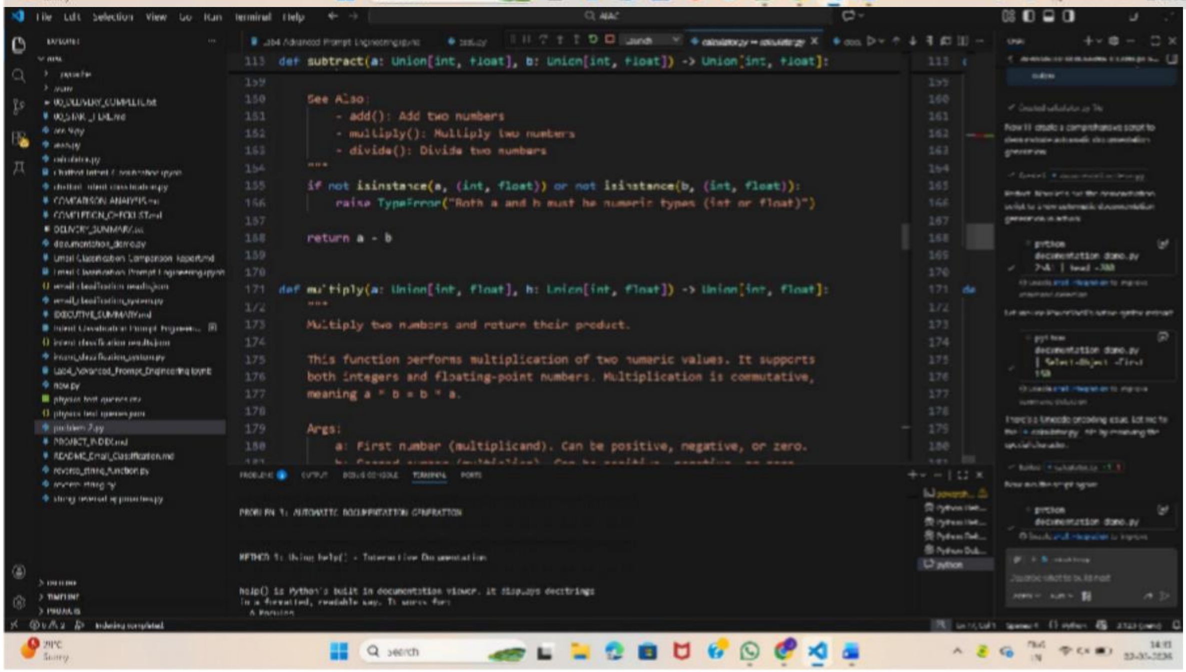

if __name__ == "__main__":
# Test basic functionality
print("\n--- Testing Basic Functionality ---\n")
print(f"add(10, 5) = {add(10, 5)}")
print(f"subtract(10, 3) = {subtract(10, 3)}")
print(f"multiply(4, 7) = {multiply(4, 7)}")
print(f"divide(20, 4) = {divide(20, 4)}")
print("\n--- Testing with Floating-Point Numbers ---\n")
print(f"add(3.5, 2.3) = {add(3.5, 2.3)}")
print(f"subtract(7.5, 2.3) = {subtract(7.5, 2.3)}")
print(f"multiply(2.5, 4) = {multiply(2.5, 4)}")
print(f"divide(7, 2) = {divide(7, 2)}")
print("\n--- Testing Edge Cases ---\n")
print(f"add(-5, 3) = {add(-5, 3)}")
print(f"multiply(-5, -3) = {multiply(-5, -3)}")
print(f"divide(0, 5) = {divide(0, 5)}")
print("\n--- Testing Error Handling ---\n")
PROBLEM 3: AUTOMATIC DOCUMENTATION GENERATION
METHOD 1: Using help() - Interactive Documentation
help() is Python's built-in documentation viewer. It displays docstrings
in a formatted, readable way.

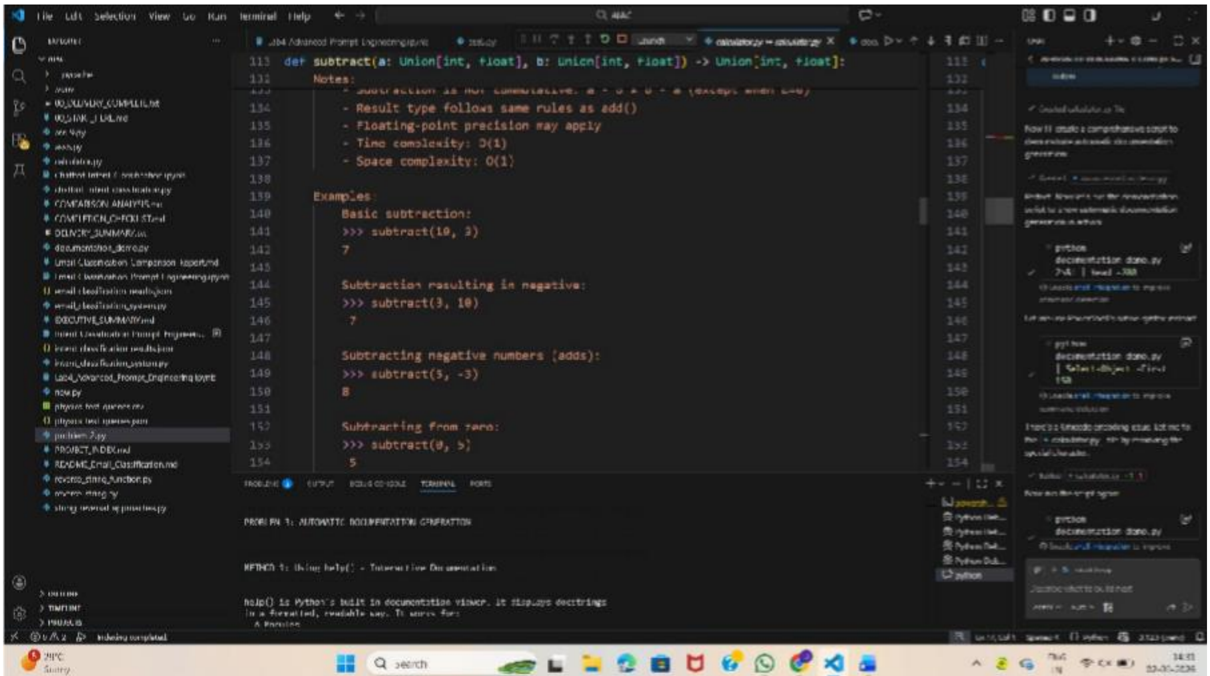

def divide(a: Union[int, float], b: Union[int, float]) -> float:
See Also:
- multiply(): Multiply two numbers
- add(): Add two numbers
- subtract(): Subtract two numbers
"""
if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
raise TypeError("Both a and b must be numeric types (int or float)")
if b == 0:
raise ValueError("Cannot divide by zero. The divisor must be non-zero.")
return a / b
# DEMONSTRATION CODE (when run as main)
if __name__ == "__main__":
print("=" * 80)
print("CALCULATOR MODULE - DEMONSTRATION")
print("=" * 80)
PROBLEM 1: AUTOMATIC DOCUMENTATION GENERATION
help() is Python's built in documentation viewer. It displays docstrings

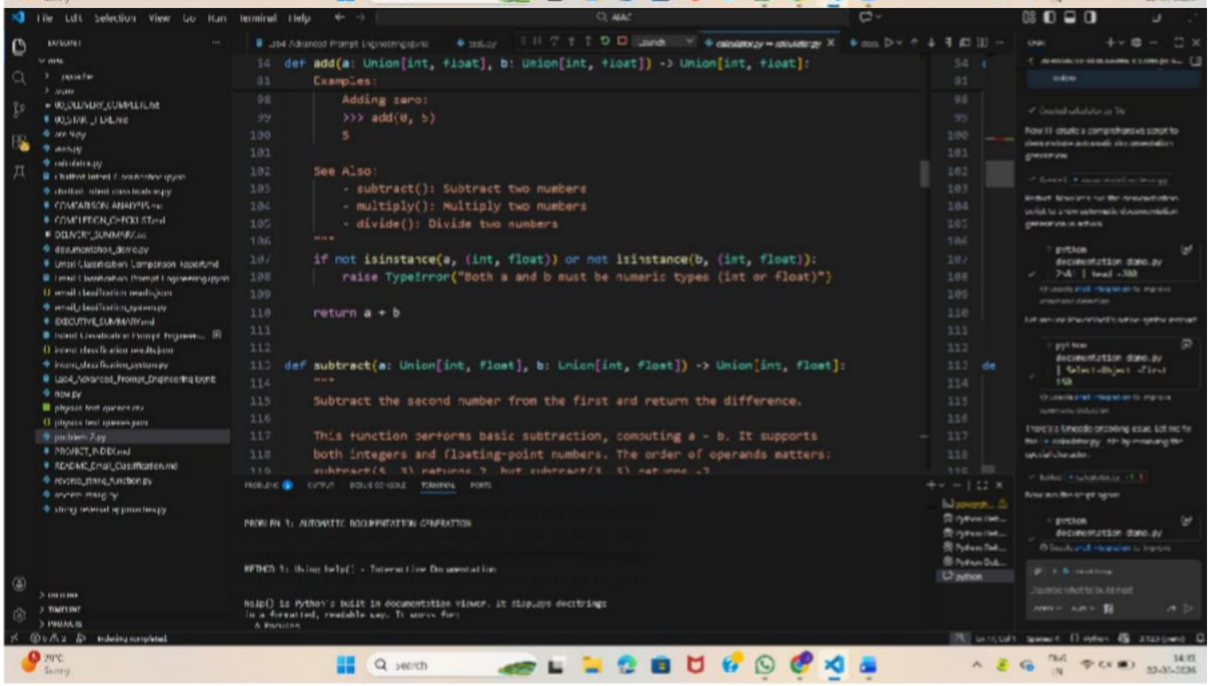

def divide(a: Union[int, float], b: Union[int, float]) -> float:
Examples:
Division with remainder:
>>> divide(7, 2)
3.5
Division by larger number (returns < 1):
>>> divide(2, 5)
0.4
Division with negative numbers:
>>> divide(-10, 2)
-5.0
Zero divided by a number:
>>> divide(0, 5)
0.0
Division by zero raises error:
>>> divide(10, 0)
Traceback (most recent call last):
...
ValueError: Cannot divide by zero
PROBLEM 1: AUTOMATIC DOCUMENTATION GENERATION
help() is Python's built in documentation viewer. It displays docstrings


def divide(a: Union[int, float], b: Union[int, float]) -> float:
Args:
b: Second number (divisor). The number to divide by. Must NOT be zero.
Can be positive or negative.
Returns:
float: The quotient (a / b). Always returns float for precision,
even if both inputs are integers.
Raises:
TypeError: If either a or b is not a numeric type (int or float).
ValueError: If b (the divisor) is zero. Division by zero is undefined
in mathematics and raises this error.
Notes:
Division is NOT commutative: a / b ≠ b / a
- Always returns float, even for integer inputs
- Division by zero raises ValueError (cannot be caught as ZeroDivisionError)
- Results in float precision/rounding behavior
- Zero divided by any number returns 0.0
- Time complexity: O(1)
- Space complexity: O(1)
PROBLEM 3: AUTOMATIC DOCUMENTATION GENERATION
METHOD 1: Using help() - Interactive Documentation


def multiply(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
Examples:
Floating-point multiplication:
>>> multiply(2.5, 4)
10.0
See Also:
- add(): Add two numbers
- subtract(): Subtract two numbers
- divide(): Divide two numbers
"""
if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
raise TypeError("Both a and b must be numeric types (int or float)")
return a * b
def divide(a: Union[int, float], b: Union[int, float]) -> float:
"""
Divide the first number by the second and return the quotient.
This function performs division (a / b), returning a floating-point result.
PROBLEM 3: AUTOMATIC DOCUMENTATION GENERATION
METHOD 1: Using help() - Interactive Documentation


def multiply(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
Returns:
Union[int, float]: The product of a and b. Returns int if both
inputs are integers, otherwise returns float.
Raises:
TypeError: If either a or b is not a numeric type (int or float).
Notes:
- Multiplication is commutative: a * b = b * a
- Any number multiplied by zero equals zero
- Negative * Negative = Positive
- Negative * Positive = Negative
- Time complexity: O(1)
- Space complexity: O(1)
Examples:
Multiplying two positive numbers:
>>> multiply(5, 3)
15
Multiplying by zero:
>>> multiply(100, 0)
PROBLEM 3: AUTOMATIC DOCUMENTATION GENERATION
METHOD 1: Using help() - Interactive Documentation
help() is Python's built in documentation viewer. It displays docstrings
in a formatted, readable way.

def subtract(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
See Also:
- add(): Add two numbers
- multiply(): Multiply two numbers
- divide(): Divide two numbers
"""
if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
raise TypeError("Both a and b must be numeric types (int or float)")
return a - b
def multiply(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
"""
Multiply two numbers and return their product.
This function performs multiplication of two numeric values. It supports
both integers and floating-point numbers. Multiplication is commutative,
meaning a * b = b * a.
Args:
a: First number (multiplicand). Can be positive, negative, or zero.
PROBLEM 3: AUTOMATIC DOCUMENTATION GENERATION
METHOD 1: Using help() - Interactive Documentation
help() is Python's built in documentation viewer. It displays docstrings
in a formatted, readable way.





Problem 4: Conversion Utilities Module

Task:

1. Write a module named conversion.py with functions:

o decimal_to_binary(n)

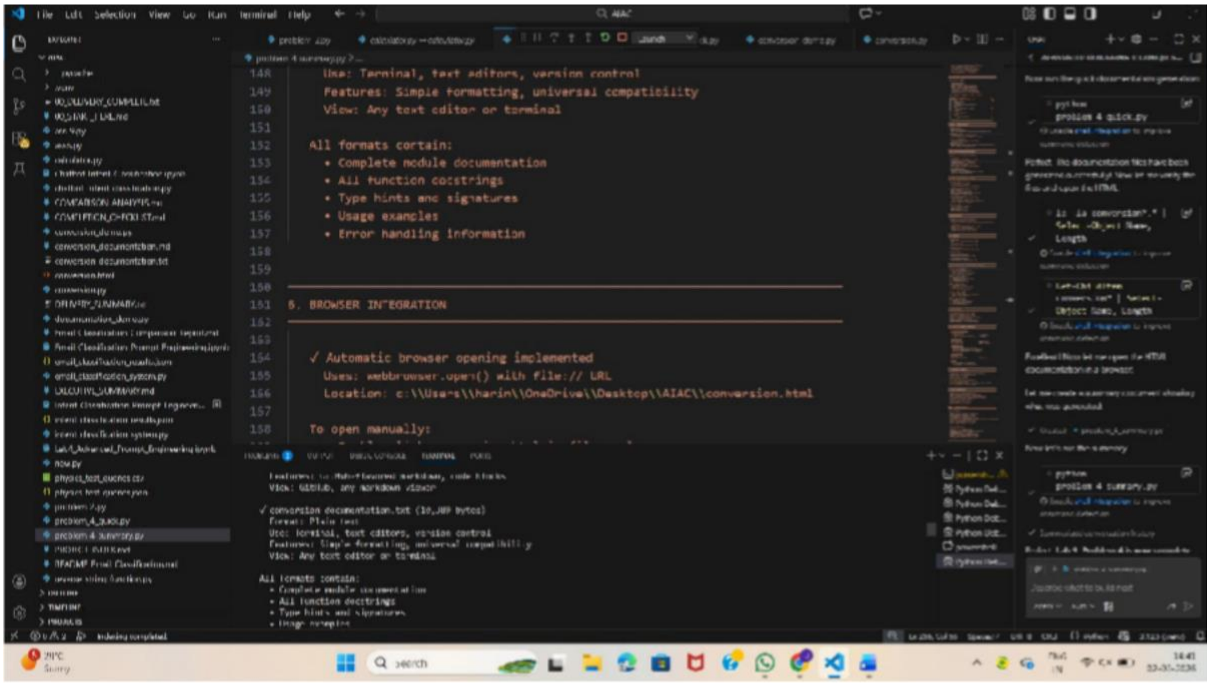
o binary_to_decimal(b)

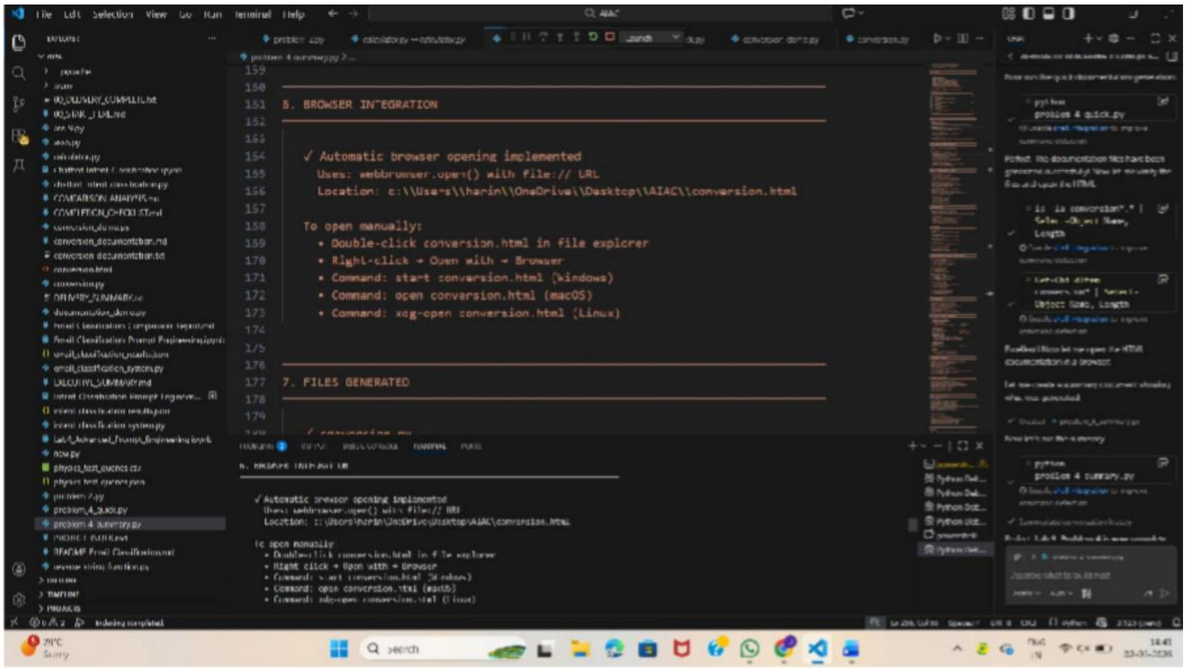
o decimal_to_hexadecimal(n)

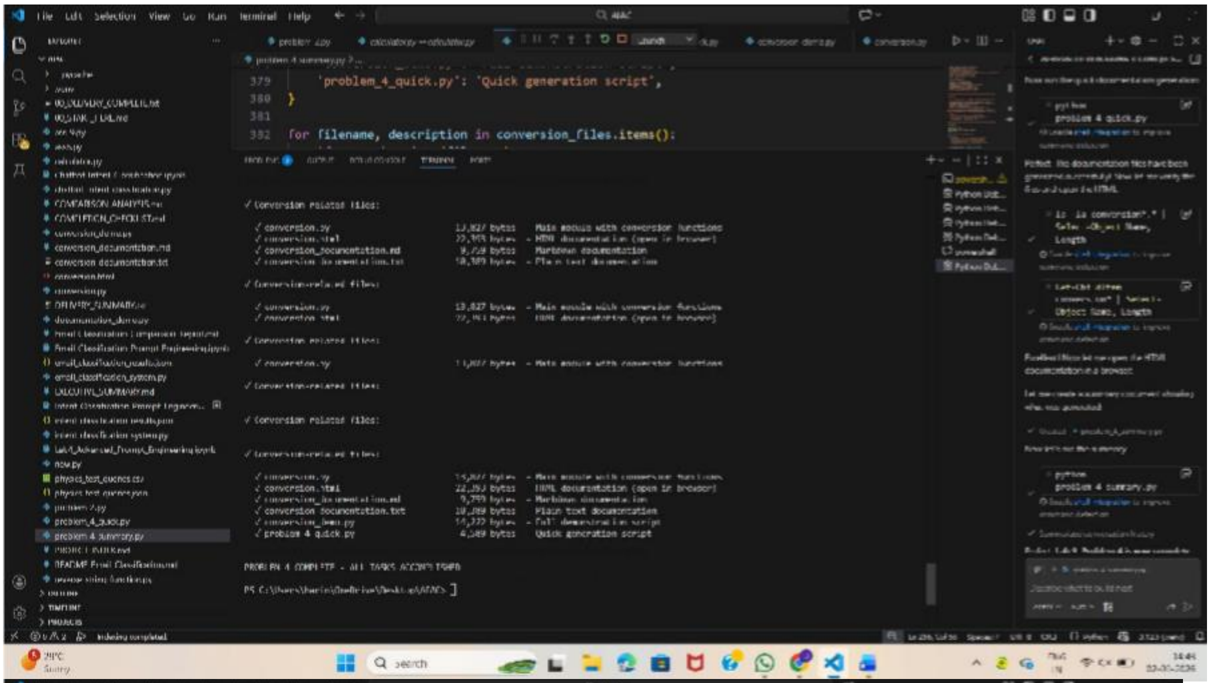
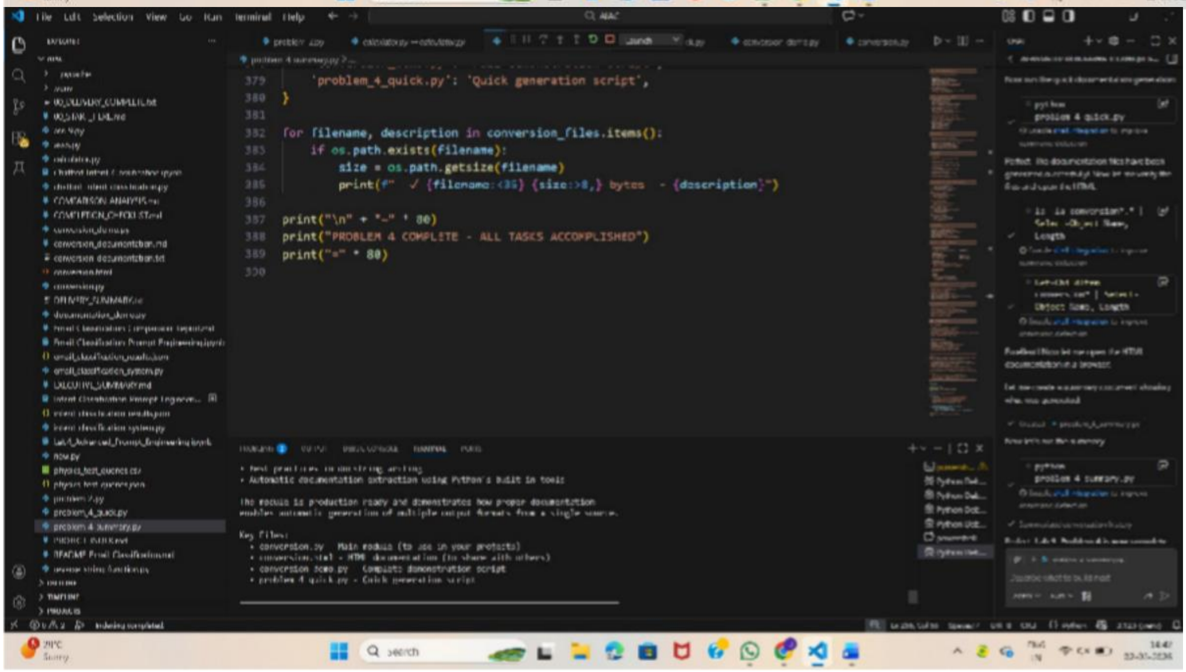
2. Use Copilot for auto-generating docstrings.

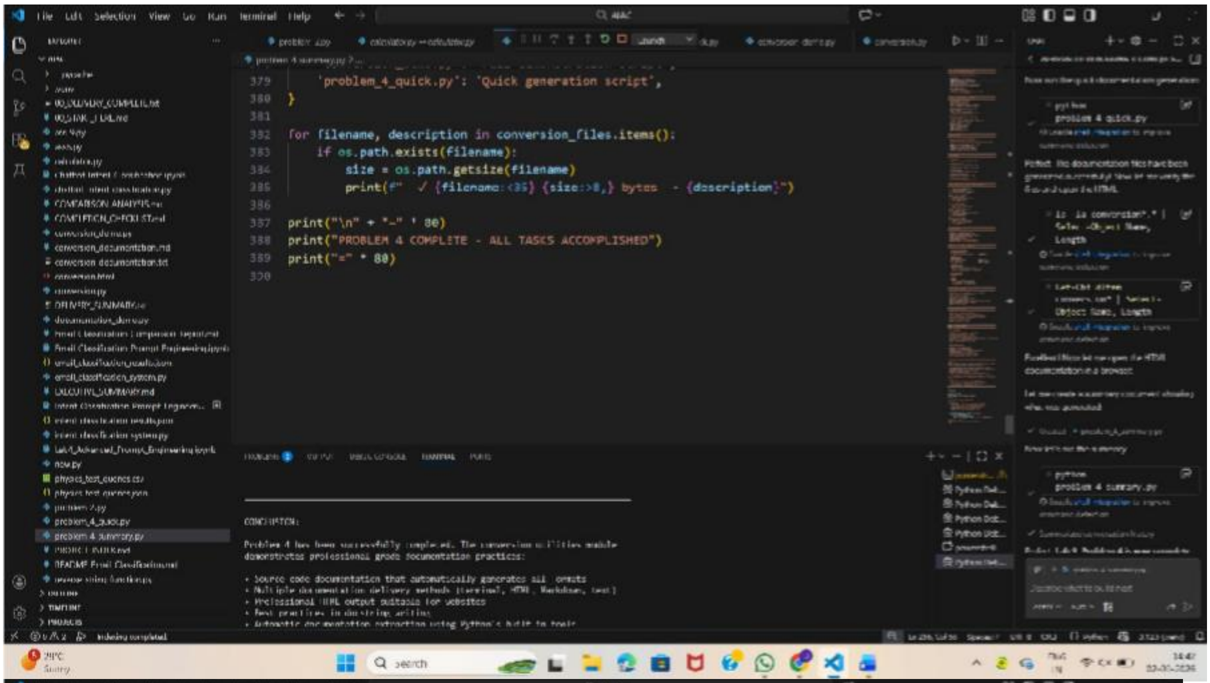
3. Generate documentation in the terminal.

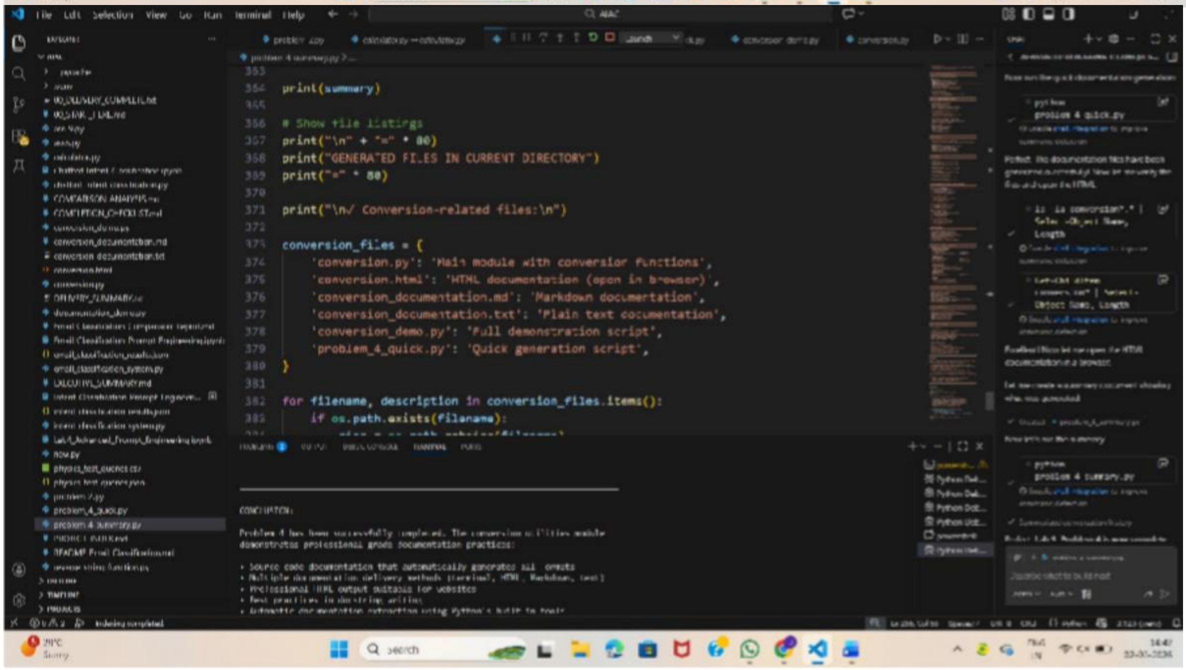
4. Export the documentation in HTML format and open it in a Browser

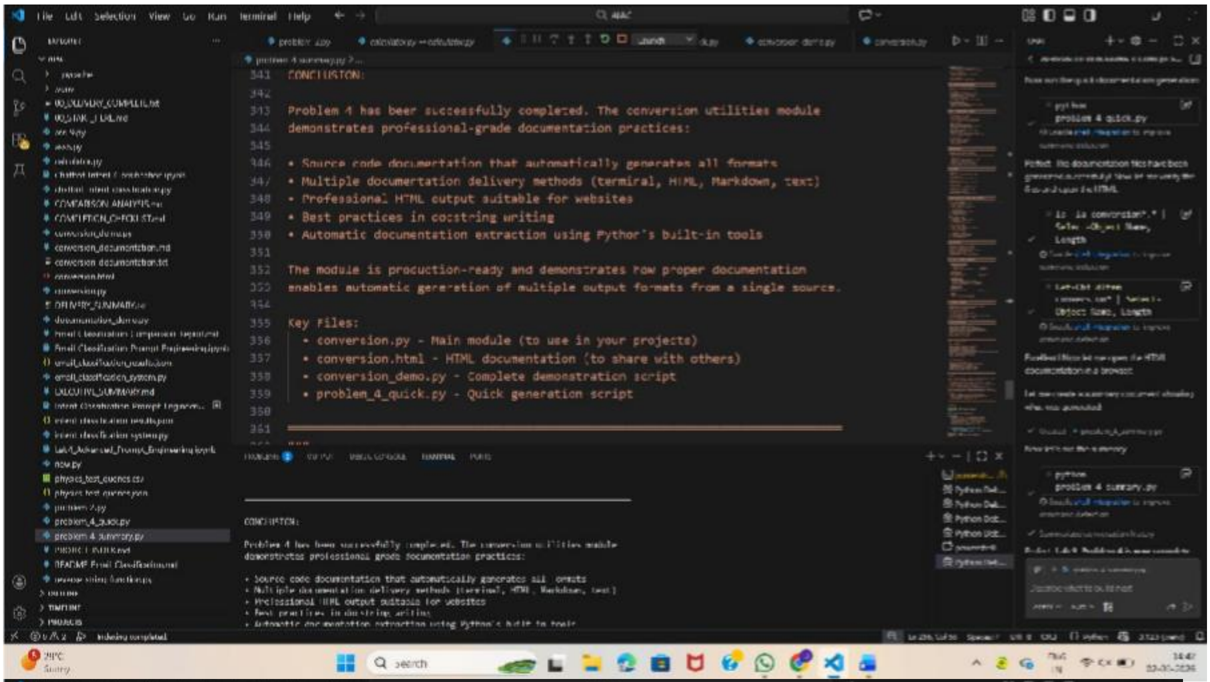


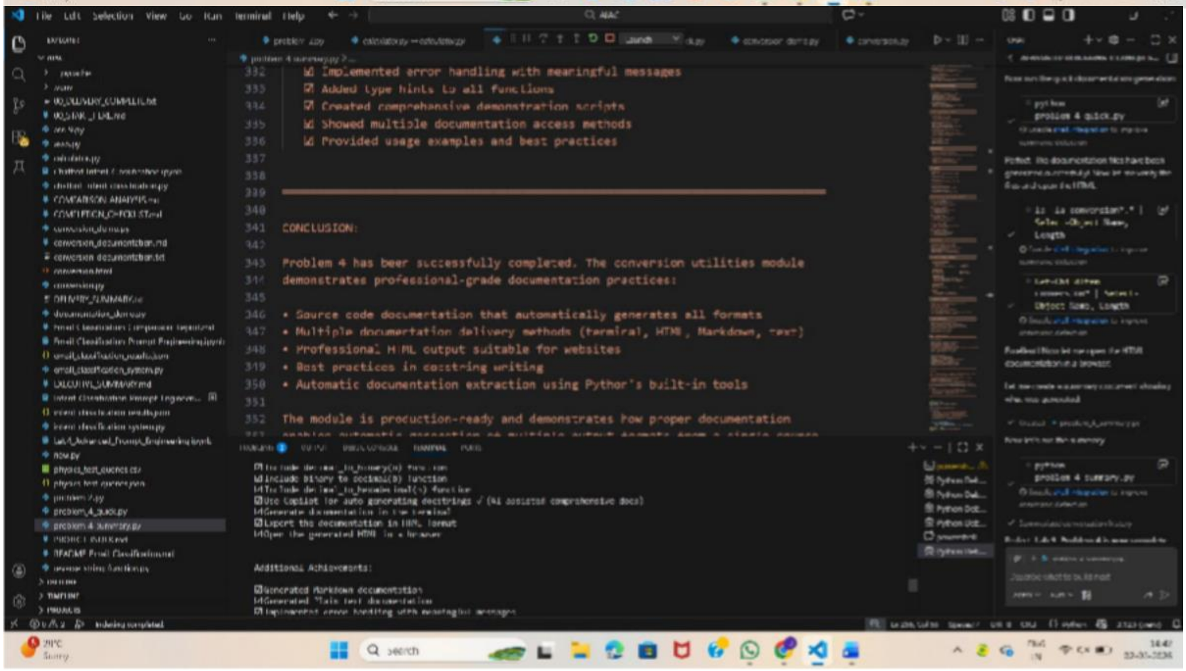


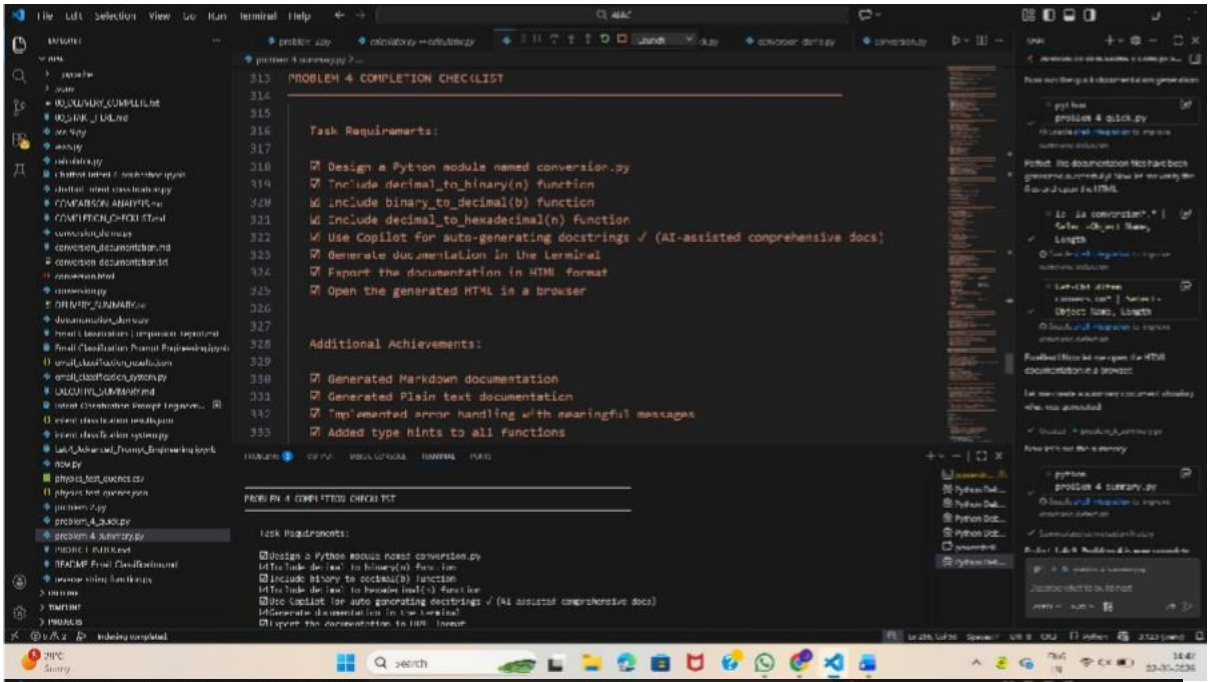

'problem_4_quick.py': 'Quick generation script',
}
for filename, description in conversion_files.items():
✓ Conversion related files:
✓ conversion.py
✓ conversion_documentation.md
Markdown documentation
✓ conversion.html
HTML documentation (open in browser)
✓ conversion documentation.txt
Plain text documentation
✓ problem 4 quick.py
Quick generation script

'problem_4_quick.py': 'Quick generation script',
}
for filename, description in conversion_files.items():
if os.path.exists(filename):
size = os.path.getsize(filename)
print("\n" + "=" * 80)
print("PROBLEM 4 COMPLETE - ALL TASKS ACCOMPLISHED")
print("=" * 80)
• Automatic documentation extraction using Python's built in tools
Key files:
• conversion.py Main module (to use in your projects)
• conversion demo.py Complete demonstration script

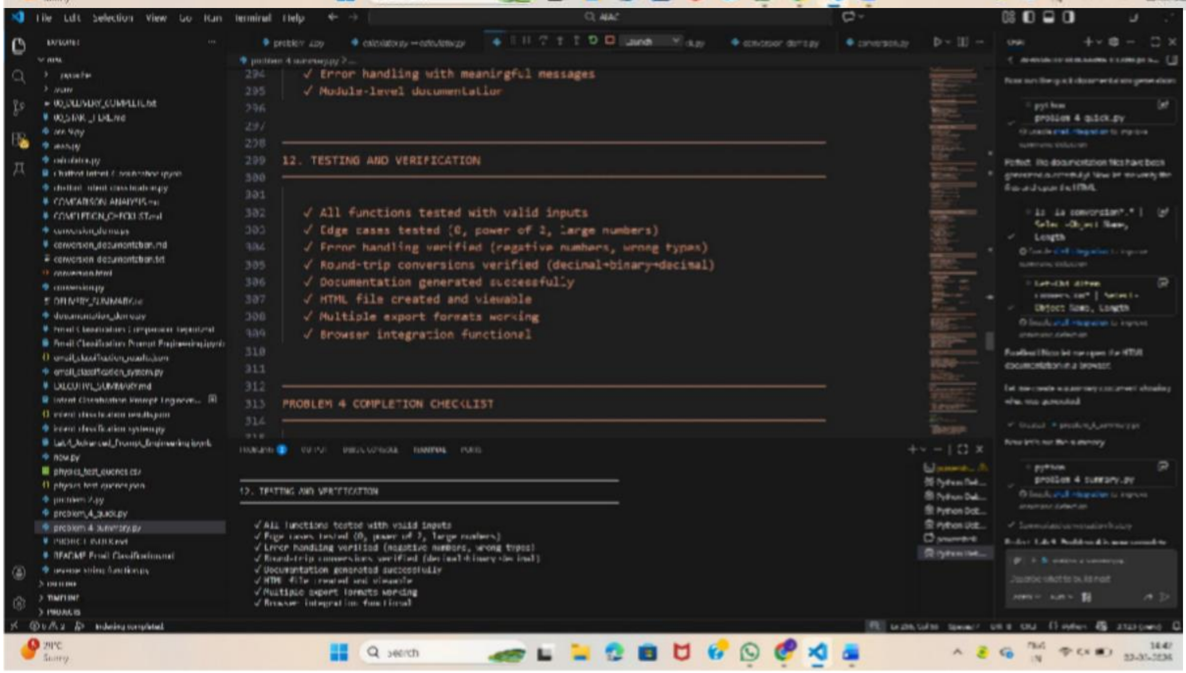

'problem_4_quick.py': 'Quick generation script',
}
for filename, description in conversion_files.items():
if os.path.exists(filename):
size = os.path.getsize(filename)
print("\n" + "-" * 80)
print("PROBLEM 4 COMPLETE - ALL TASKS ACCOMPLISHED")
print("=" * 80)

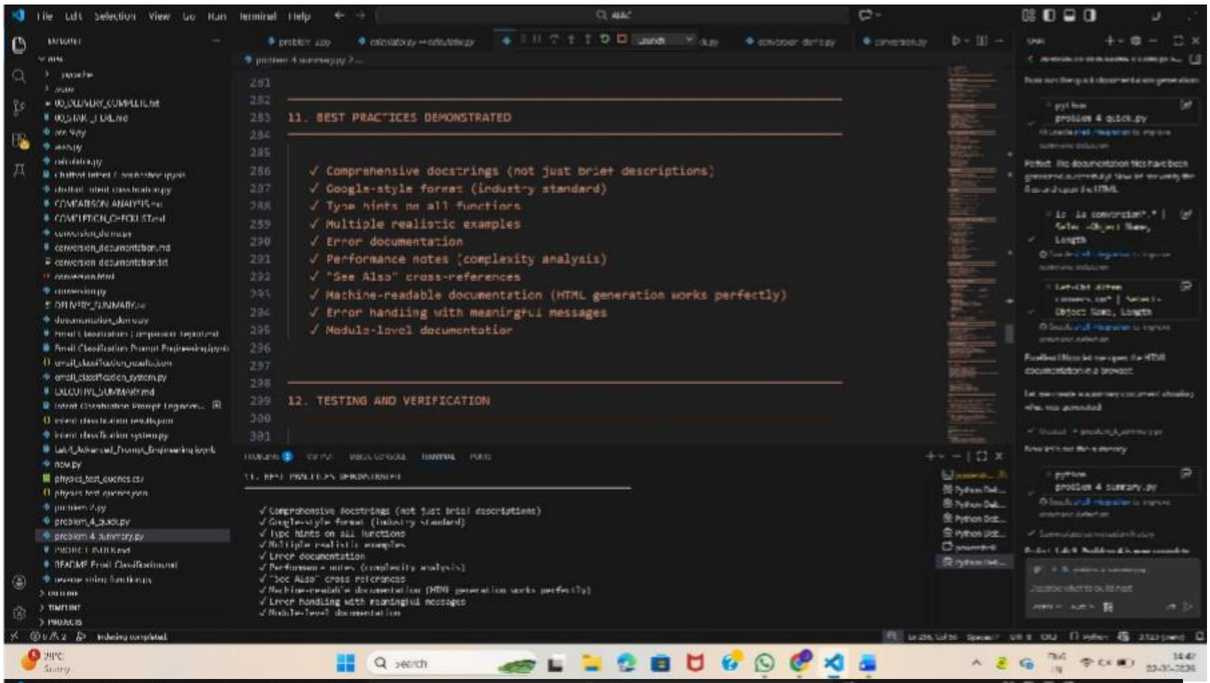

print(summary)
# Show file listings
print("GENERATED FILES IN CURRENT DIRECTORY")
print("\n✓ Conversion-related files:\n")
conversion_files = {
'conversion.py': 'Main module with conversion functions',
'conversion.html': 'HTML documentation (open in browser)',
'conversion_documentation.md': 'Markdown documentation',
'conversion_documentation.txt': 'Plain text documentation',
'conversion_demo.py': 'Full demonstration script',
'problem_4_quick.py': 'Quick generation script',
}
for filename, description in conversion_files.items():
if os.path.exists(filename):


CONCLUSION:
Problem 4 has been successfully completed. The conversion utilities module
demonstrates professional-grade documentation practices:
• Source code documentation that automatically generates all formats
• Multiple documentation delivery methods (terminal, HTML, Markdown, text)
• Professional HTML output suitable for websites
• Best practices in docstring writing
• Automatic documentation extraction using Python's built-in tools
The module is production-ready and demonstrates how proper documentation
enables automatic generation of multiple output formats from a single source.
Key Files:
• conversion.py - Main module (to use in your projects)
• conversion.html - HTML documentation (to share with others)
• conversion_demo.py - Complete demonstration script
• problem_4_quick.py - Quick generation script

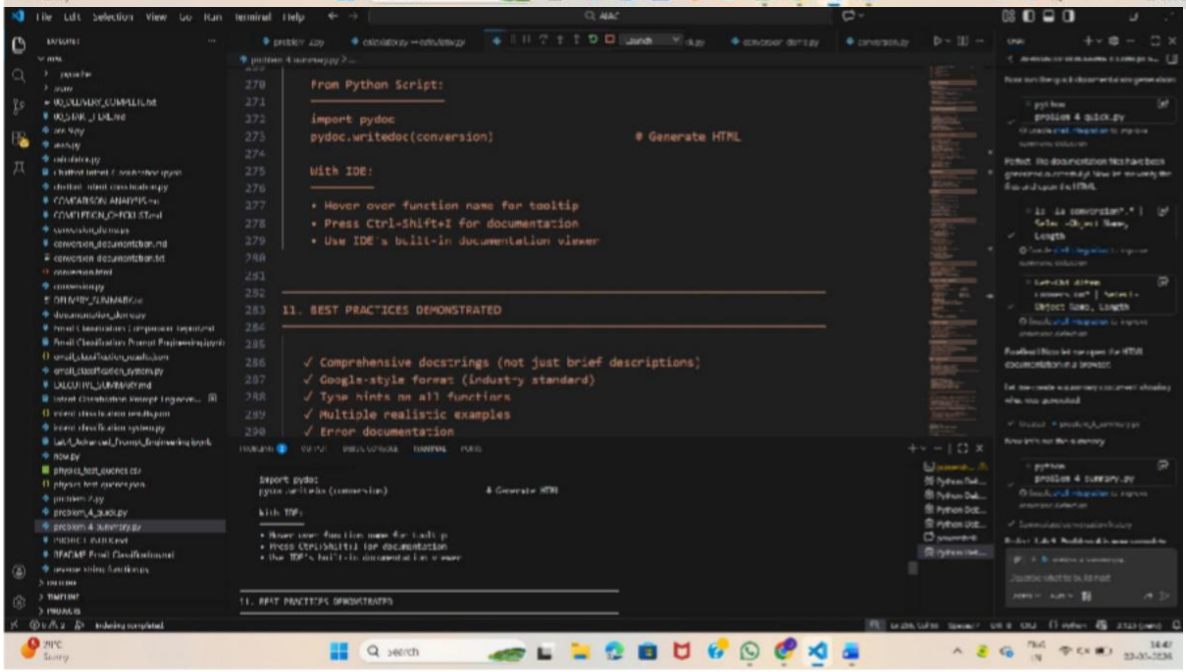

☑ Implemented error handling with meaningful messages
☑ Added type hints to all functions
☑ Created comprehensive demonstration scripts
☑ Showed multiple documentation access methods
☑ Provided usage examples and best practices
CONCLUSION:
Problem 4 has been successfully completed. The conversion utilities module
demonstrates professional-grade documentation practices:
• Source code documentation that automatically generates all formats
• Multiple documentation delivery methods (terminal, HTML, Markdown, text)
• Professional HTML output suitable for websites
• Best practices in docstring writing
• Automatic documentation extraction using Python's built-in tools
The module is production-ready and demonstrates how proper documentation
Additional Achievements:


PROBLEM 4 COMPLETION CHECKLIST
Task Requirements:
☑ Design a Python module named conversion.py
☑ Include decimal_to_binary(n) function
☑ Include binary_to_decimal(b) function
☑ Include decimal_to_hexadecimal(n) function
☑ Use Copilot for auto-generating docstrings ✓ (AI-assisted comprehensive docs)
☑ Generate documentation in the terminal
☑ Export the documentation in HTML format
☑ Open the generated HTML in a browser
Additional Achievements:
☑ Generated Markdown documentation
☑ Generated Plain text documentation
☑ Implemented error handling with meaningful messages
☑ Added type hints to all functions

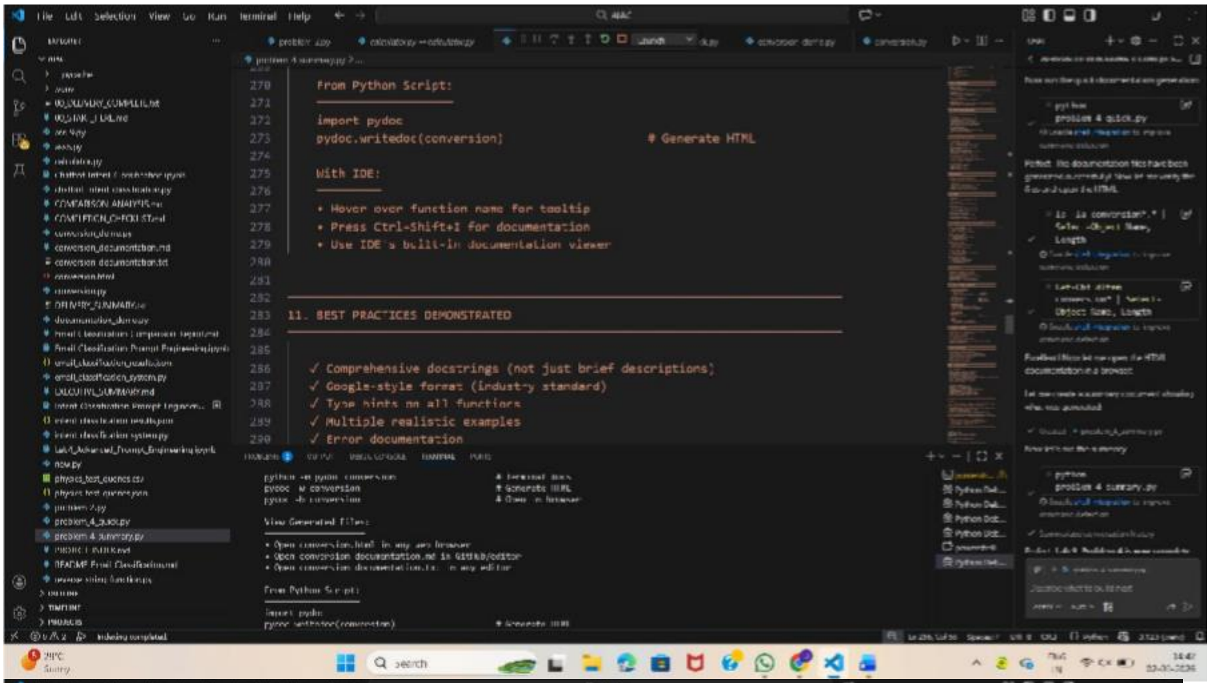

✓ Error handling with meaningful messages
✓ Module-level documentation
12. TESTING AND VERIFICATION
✓ All functions tested with valid inputs
✓ Edge cases tested (0, power of 2, large numbers)
✓ Error handling verified (negative numbers, wrong types)
✓ Round-trip conversions verified (decimal→binary→decimal)
✓ Documentation generated successfully
✓ HTML file created and viewable
✓ Multiple export formats working
✓ Browser integration functional
PROBLEM 4 COMPLETION CHECKLIST

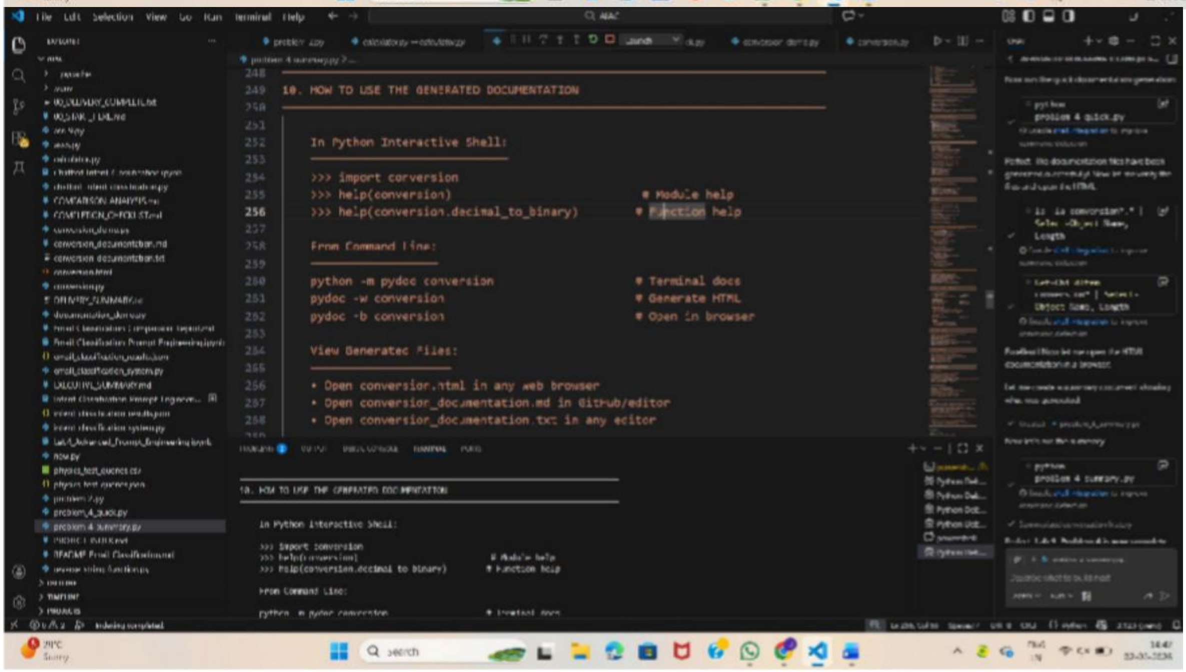


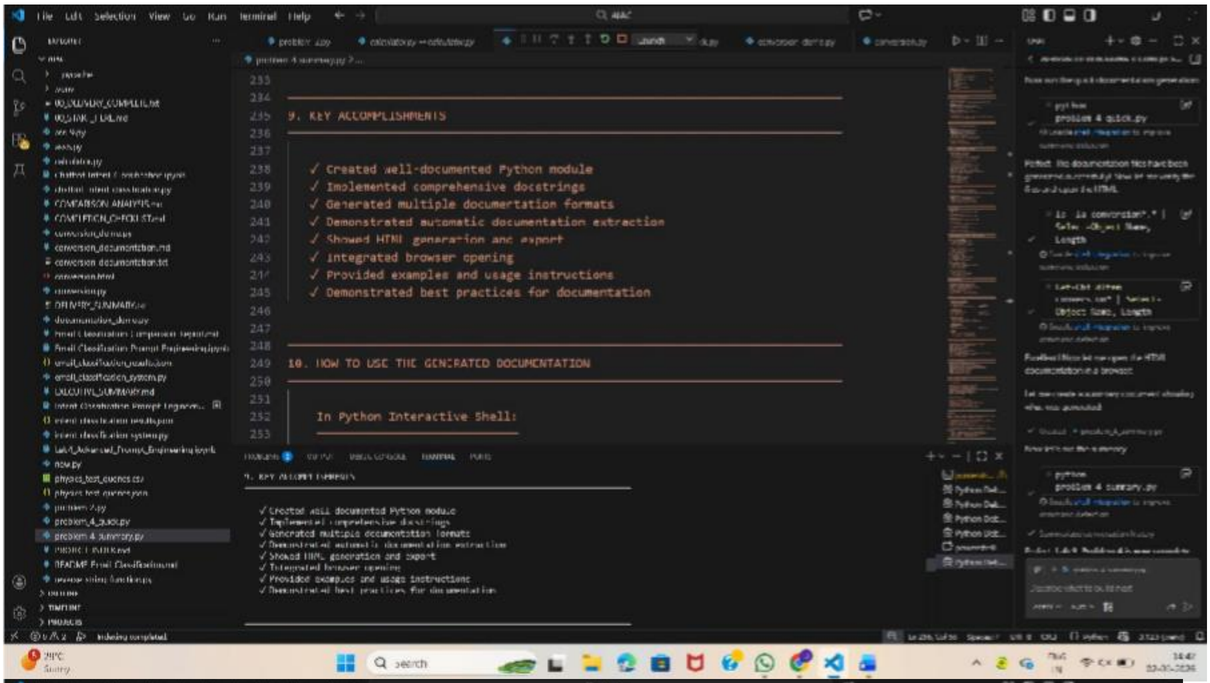
11. BEST PRACTICES DEMONSTRATED
✓ Comprehensive docstrings (not just brief descriptions)
✓ Google-style format (industry standard)
✓ Type hints on all functions
✓ Multiple realistic examples
✓ Error documentation
✓ Performance notes (complexity analysis)
✓ "See Also" cross-references
✓ Machine-readable documentation (HTML generation works perfectly)
✓ Error handling with meaningful messages
✓ Module-level documentation
12. TESTING AND VERIFICATION

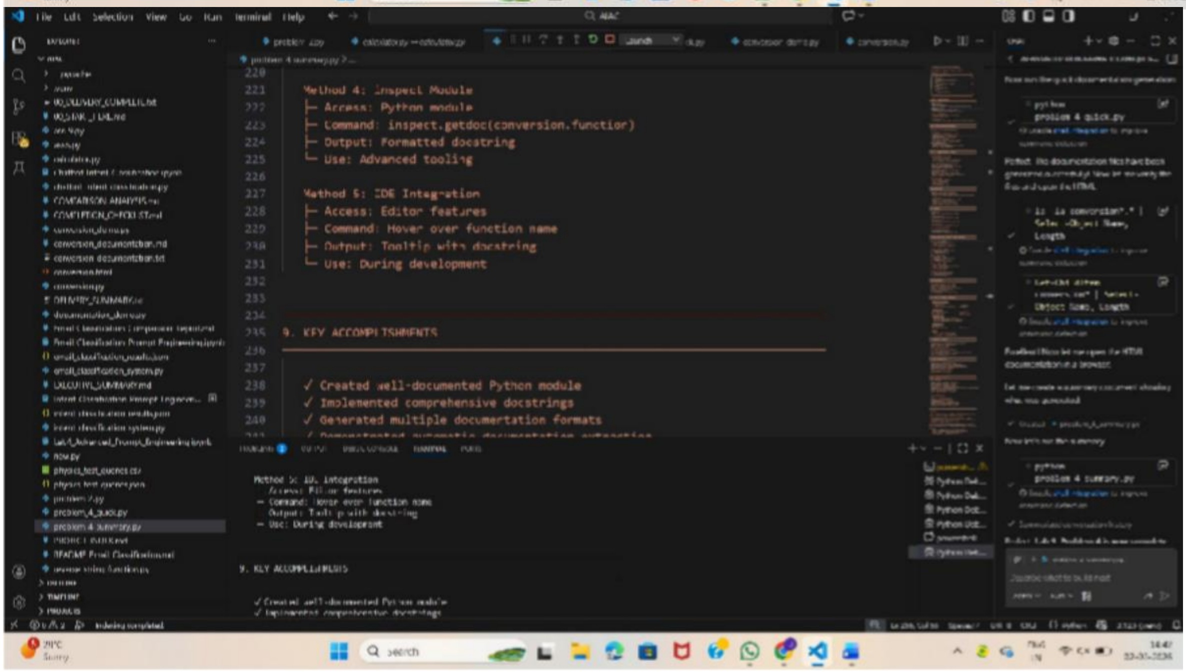


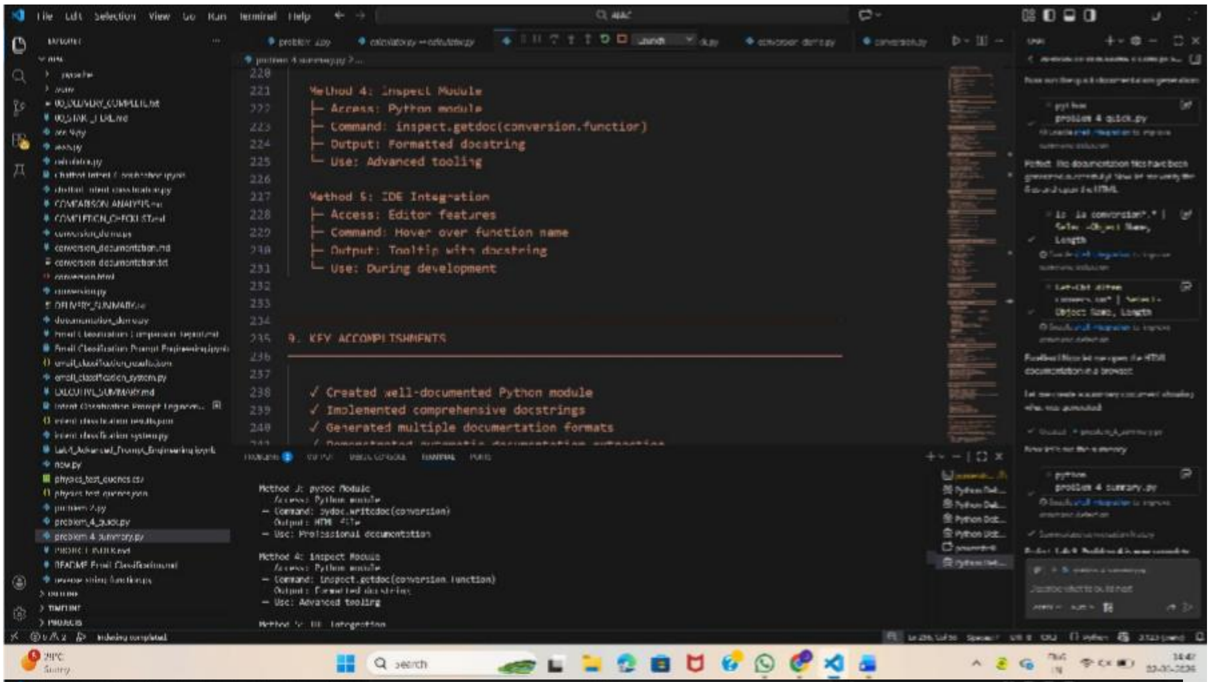
From Python Script:
import pydoc
pydoc.writedoc(conversion)
# Generate HTML
With IDE:
• Hover over function name for tooltip
• Press Ctrl+Shift+I for documentation
• Use IDE's built-in documentation viewer
11. BEST PRACTICES DEMONSTRATED
✓ Comprehensive docstrings (not just brief descriptions)
✓ Google-style format (industry standard)
✓ Type hints on all functions
✓ Multiple realistic examples
✓ Error documentation

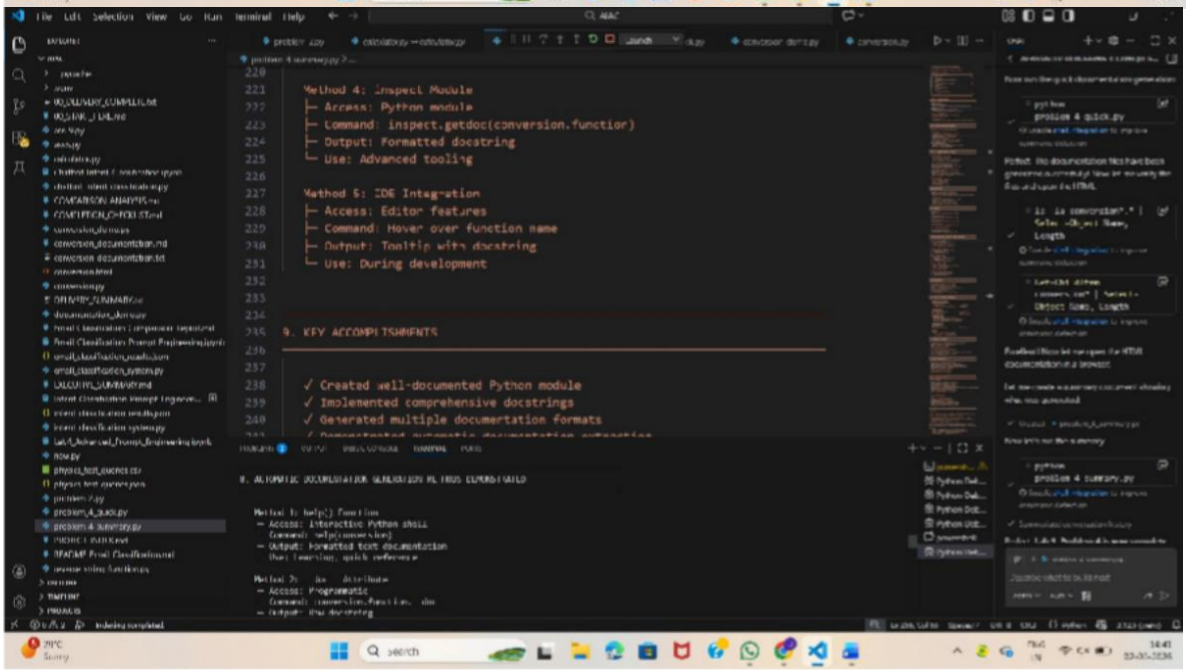

From Python Script:
import pydoc
pydoc.writedoc(conversion)
# Generate HTML
With IDE:
• Hover over function name for tooltip
• Press Ctrl-Shift+I for documentation
• Use IDE's built-in documentation viewer
11. BEST PRACTICES DEMONSTRATED
✓ Comprehensive docstrings (not just brief descriptions)
✓ Google-style format (industry standard)
✓ Type hints on all functions
✓ Multiple realistic examples
✓ Error documentation

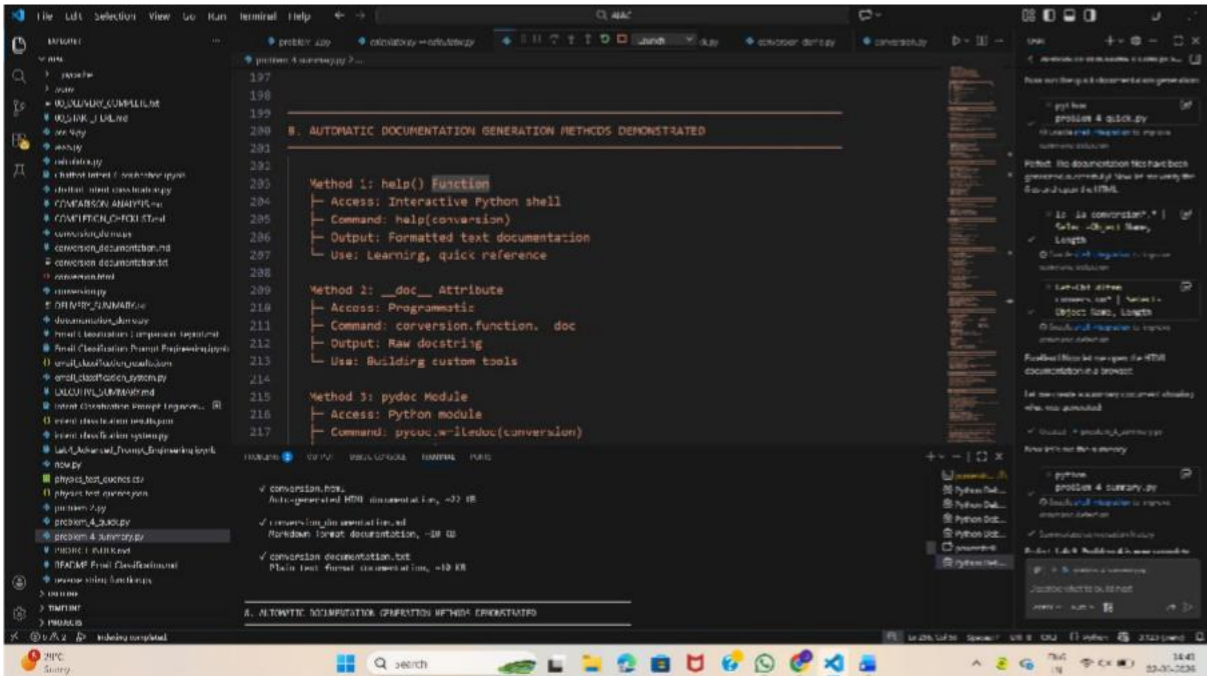

10. HOW TO USE THE GENERATED DOCUMENTATION
In Python Interactive Shell:
>>> import conversion
>>> help(conversion)
# Module help
>>> help(conversion.decimal_to_binary)
# Function help
From Command Line:
python -m pydoc conversion
Terminal docs
pydoc -w conversion
Generate HTML
pydoc -b conversion
Open in browser
View Generated Files:
• Open conversion.html in any web browser
• Open conversion_documentation.md in GitHub/editor
• Open conversion_documentation.txt in any editor

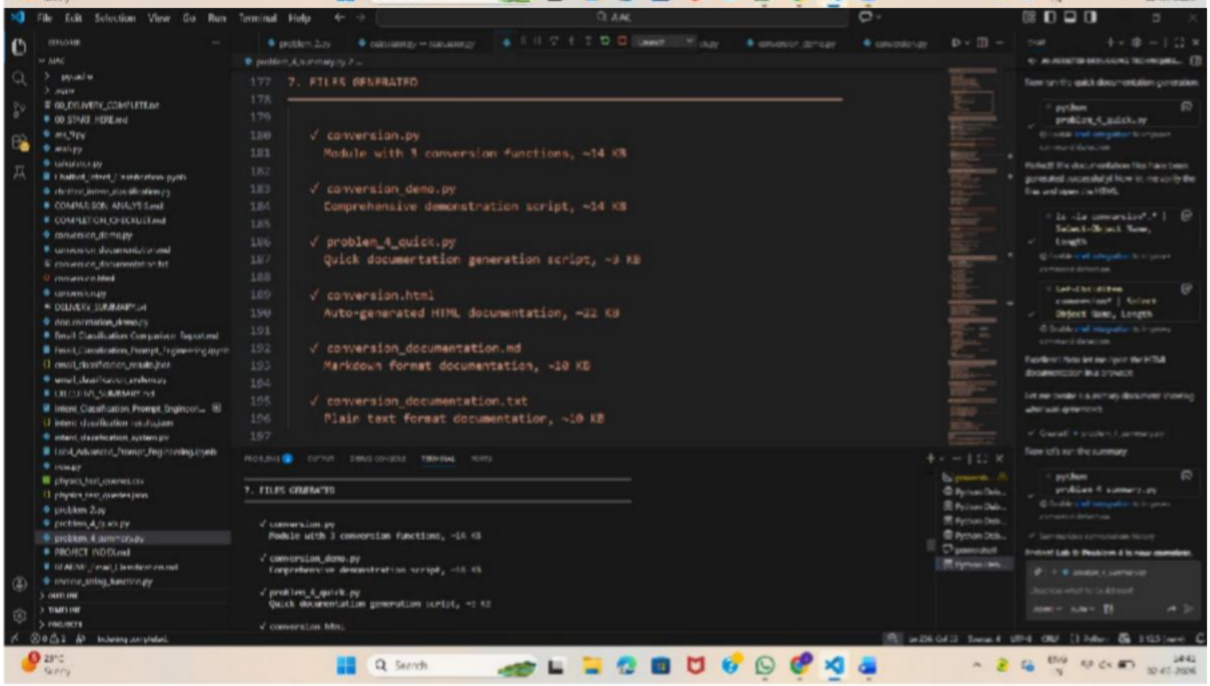

9. KEY ACCOMPLISHMENTS
✓ Created well-documented Python module
✓ Implemented comprehensive docstrings
✓ Generated multiple documentation formats
✓ Demonstrated automatic documentation extraction
✓ Showed HTML generation and export
✓ Integrated browser opening
✓ Provided examples and usage instructions
✓ Demonstrated best practices for documentation
10. HOW TO USE THE GENERATED DOCUMENTATION
In Python Interactive Shell:


Method 4: Inspect Module
├─ Access: Python module
├─ Command: inspect.getdoc(conversion.function)
├─ Output: Formatted docstring
└─ Use: Advanced tooling
Method 5: IDE Integration
├─ Access: Editor features
├─ Command: Hover over function name
├─ Output: Tooltip with docstring
└─ Use: During development
9. KEY ACCOMPLISHMENTS
✓ Created well-documented Python module
✓ Implemented comprehensive docstrings
✓ Generated multiple documentation formats


Method 4: Inspect Module
├─ Access: Python module
├─ Command: inspect.getdoc(conversion.function)
├─ Output: Formatted docstring
└─ Use: Advanced tooling
Method 5: IDE Integration
├─ Access: Editor features
├─ Command: Hover over function name
├─ Output: Tooltip with docstring
└─ Use: During development
9. KEY ACCOMPLISHMENTS
✓ Created well-documented Python module
✓ Implemented comprehensive docstrings
✓ Generated multiple documentation formats
Method 3: pydoc Module
Method 4: Inspect Module
— Command: inspect.getdoc(conversion.function)
— Use: Advanced tooling
problem 4 quick.py
problem 4 summary.py


Method 4: Inspect Module
├─ Access: Python module
├─ Command: inspect.getdoc(conversion.function)
├─ Output: Formatted docstring
└─ Use: Advanced tooling
Method 5: IDE Integration
├─ Access: Editor features
├─ Command: Hover over function name
├─ Output: Tooltip with docstring
└─ Use: During development
9. KEY ACCOMPLISHMENTS
✓ Created well-documented Python module
✓ Implemented comprehensive docstrings
✓ Generated multiple documentation formats
problem 4 quick.py
problem 4 summary.py





Problem 5 – Course Management Module

Task:

1. Create a module course.py with functions:

o add_course(course_id, name, credits)

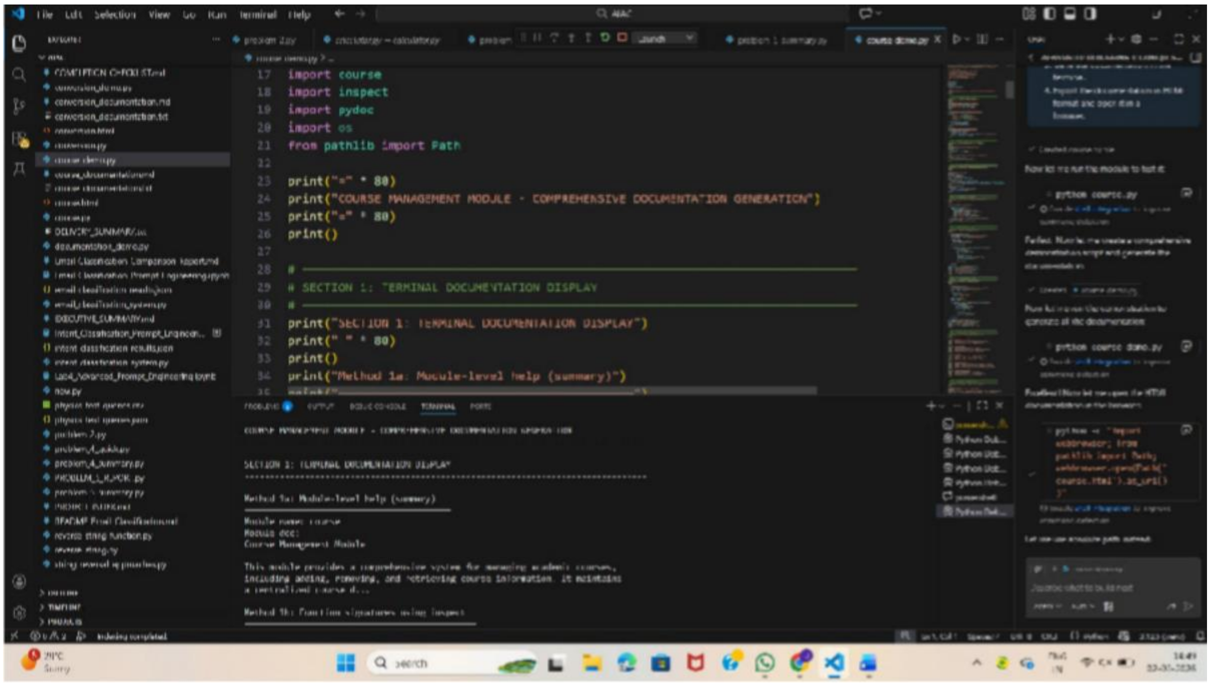
o remove_course(course_id)

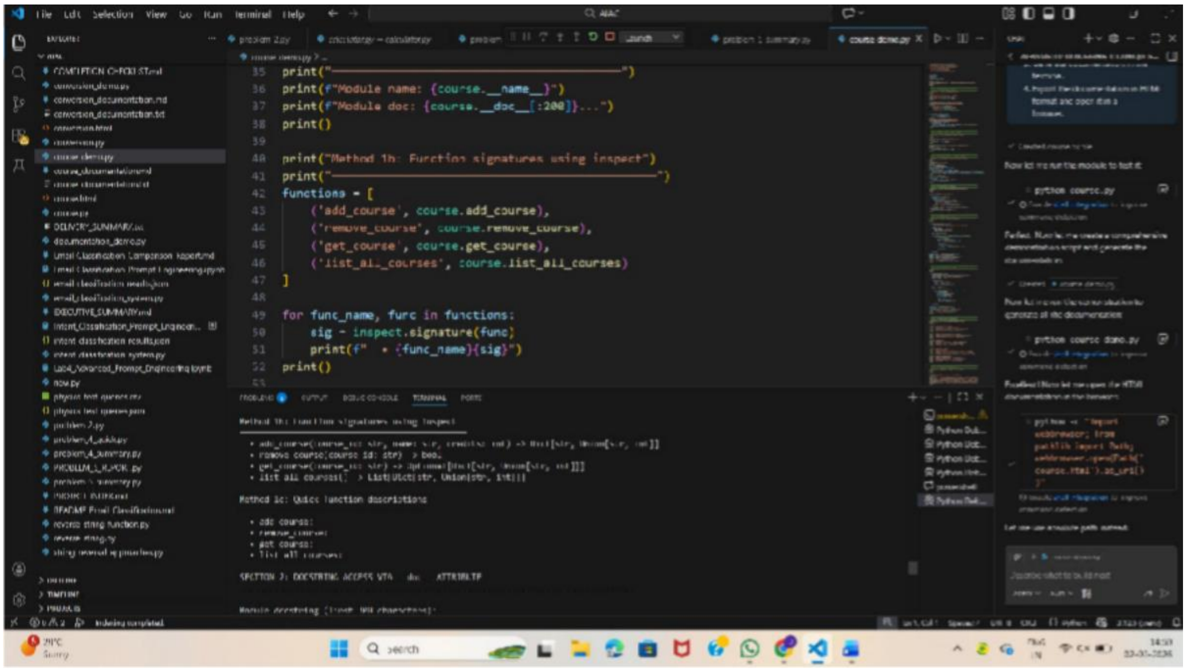
o get_course(course_id)

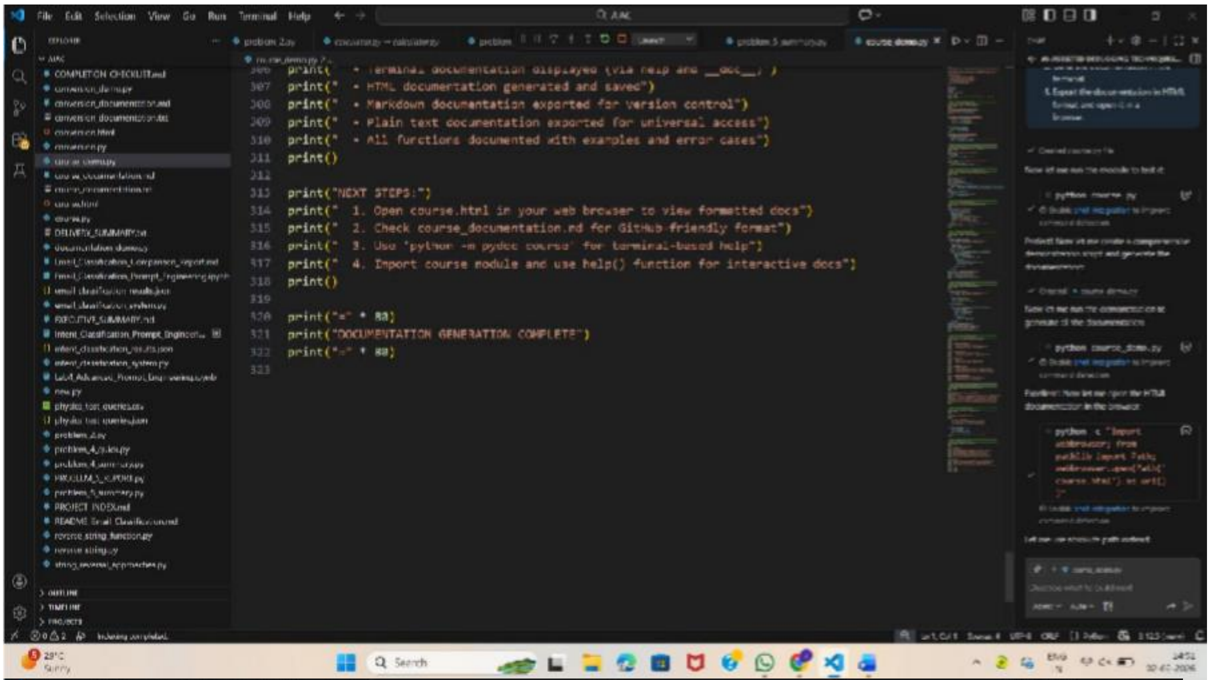
2. Add docstrings with Copilot.

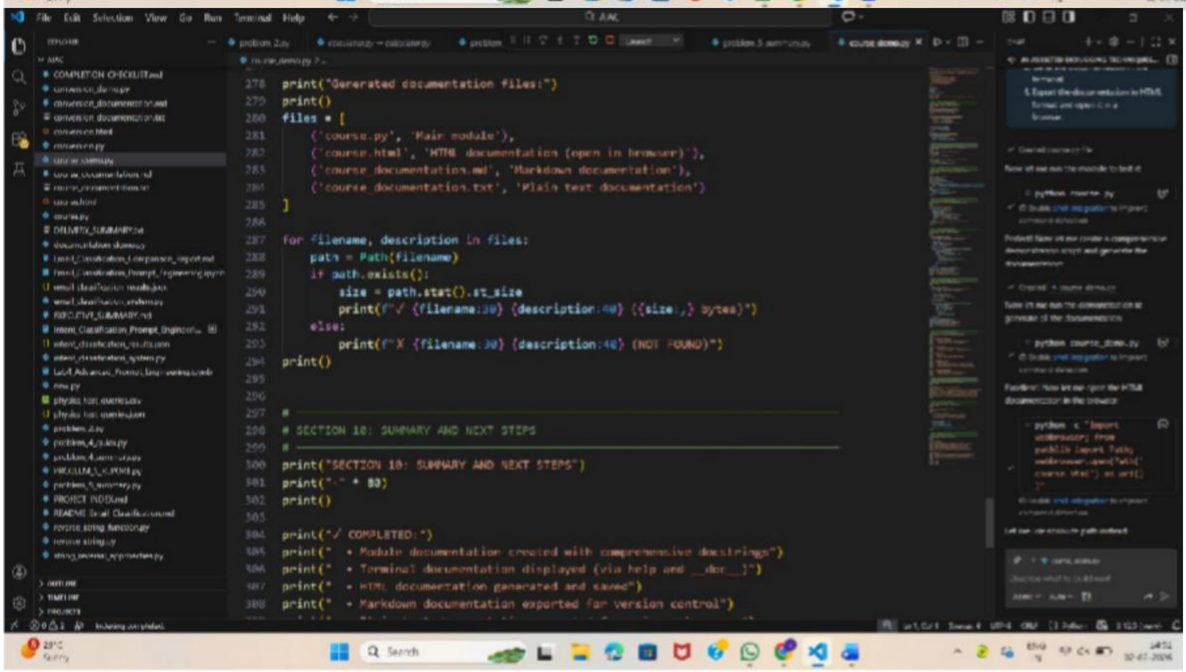
3. Generate documentation in the terminal.

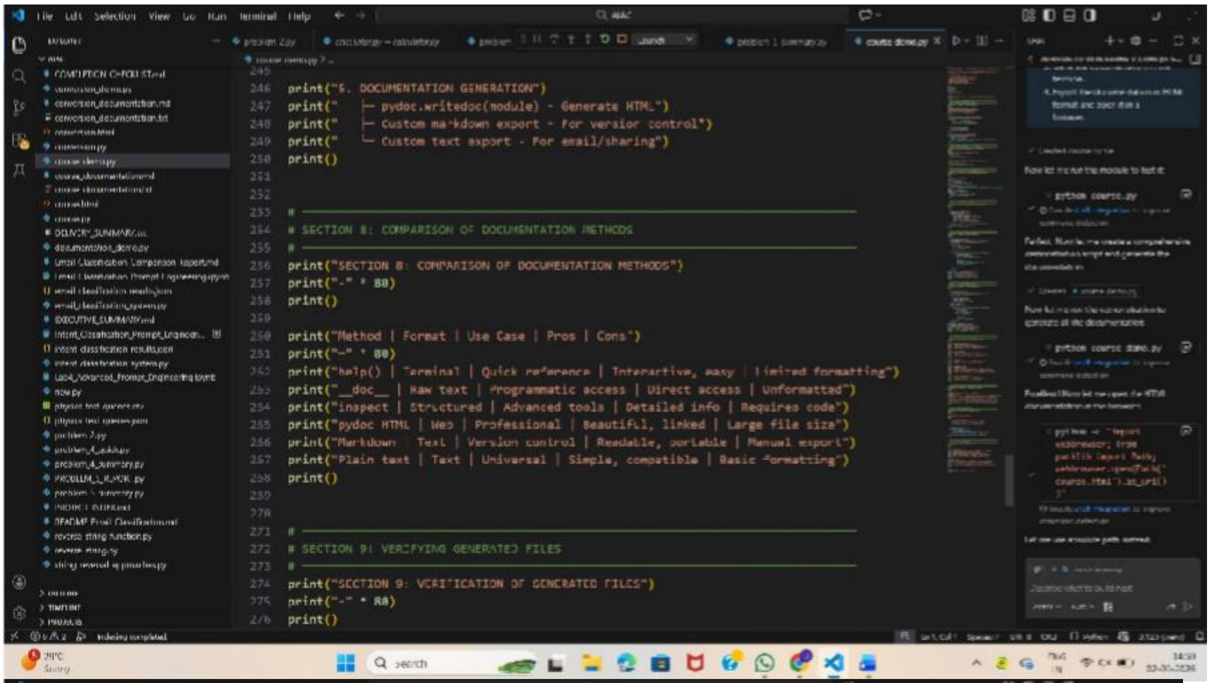
4. Export the documentation in HTML format and open it in a browser.



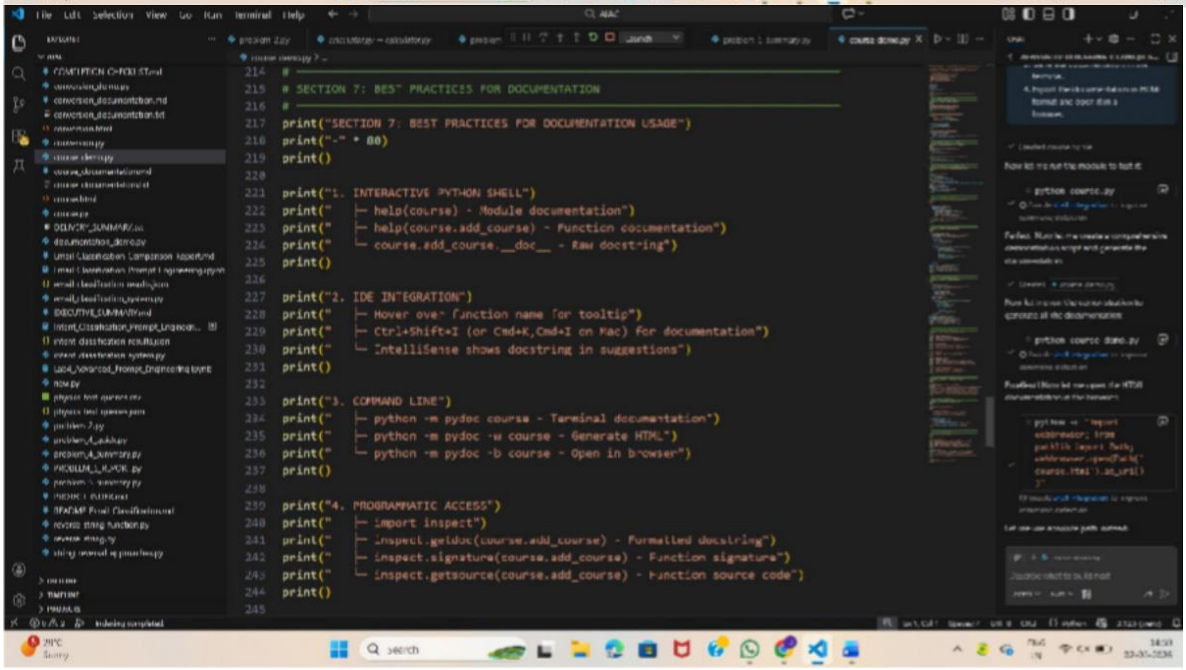



print(" • HTML documentation generated and saved")
print(" • Markdown documentation exported for version control")
print(" • Plain text documentation exported for universal access")
print(" • All functions documented with examples and error cases")
print()
print("NEXT STEPS:")
print(" 1. Open course.html in your web browser to view formatted docs")
print(" 2. Check course_documentation.md for GitHub-friendly format")
print(" 3. Use 'python -m pydoc course' for terminal-based help")
print(" 4. Import course module and use help() function for interactive docs")
print()
print("=" * 80)
print("DOCUMENTATION GENERATION COMPLETE")
print("=" * 80)

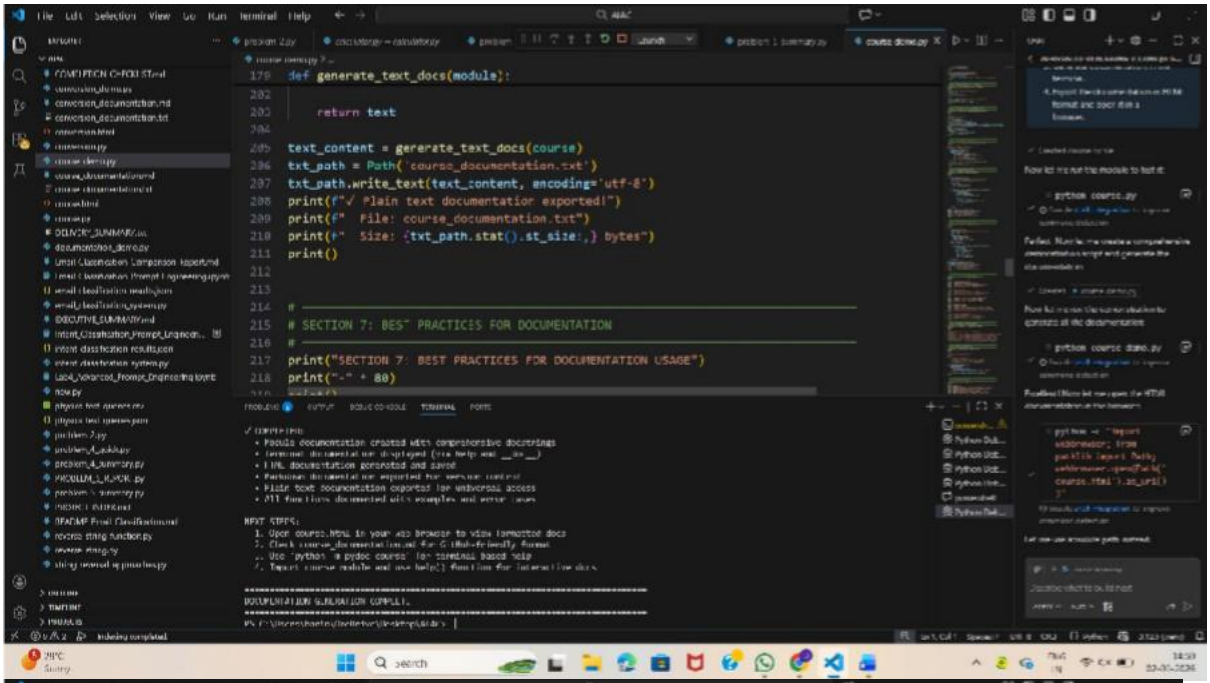

print("Generated documentation files:")
print()
files = [
('course.py', 'Main module'),
('course.html', 'HTML documentation (open in browser)'),
('course_documentation.md', 'Markdown documentation'),
('course_documentation.txt', 'Plain text documentation')
]
for filename, description in files:
path = Path(filename)
if path.exists():
size = path.stat().st_size
print(f"✓ {filename:30} {description:40} ({size:,} bytes)")
else:
print(f"X {filename:30} {description:40} (NOT FOUND)")
print()
# SECTION 10: SUMMARY AND NEXT STEPS
print("SECTION 10: SUMMARY AND NEXT STEPS")
print("-" * 80)
print()
print("✓ COMPLETED:")
print(" • Module documentation created with comprehensive docstrings")
print(" • Terminal documentation displayed (via help and __doc__)")
print(" • HTML documentation generated and saved")
print(" • Markdown documentation exported for version control")

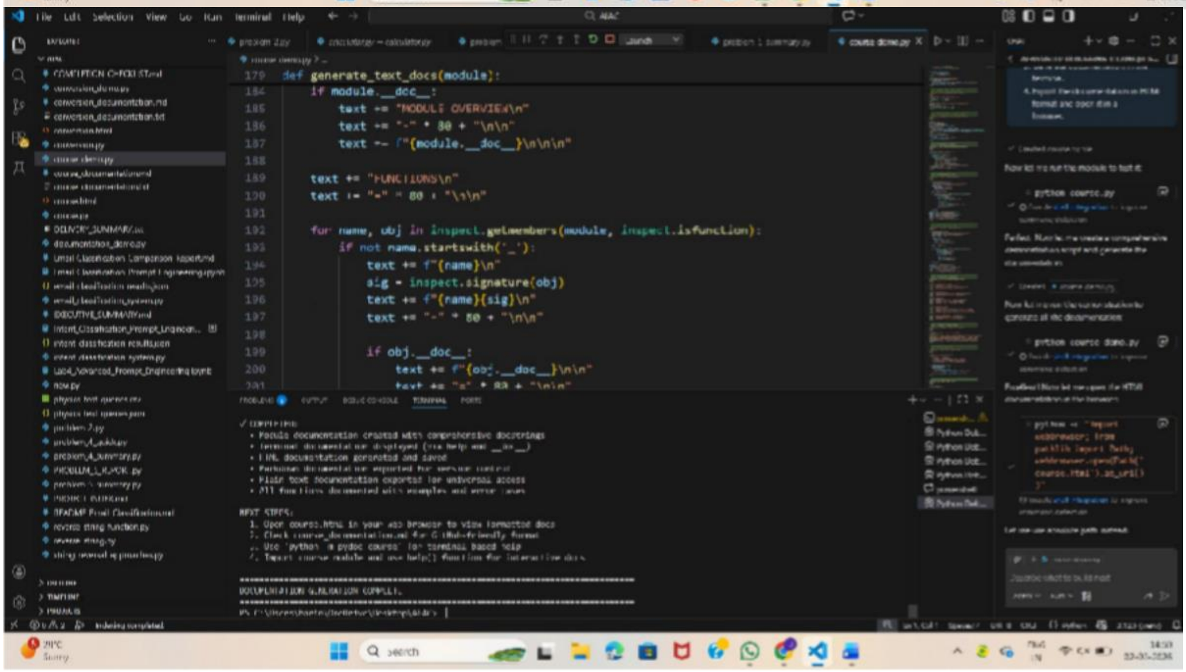


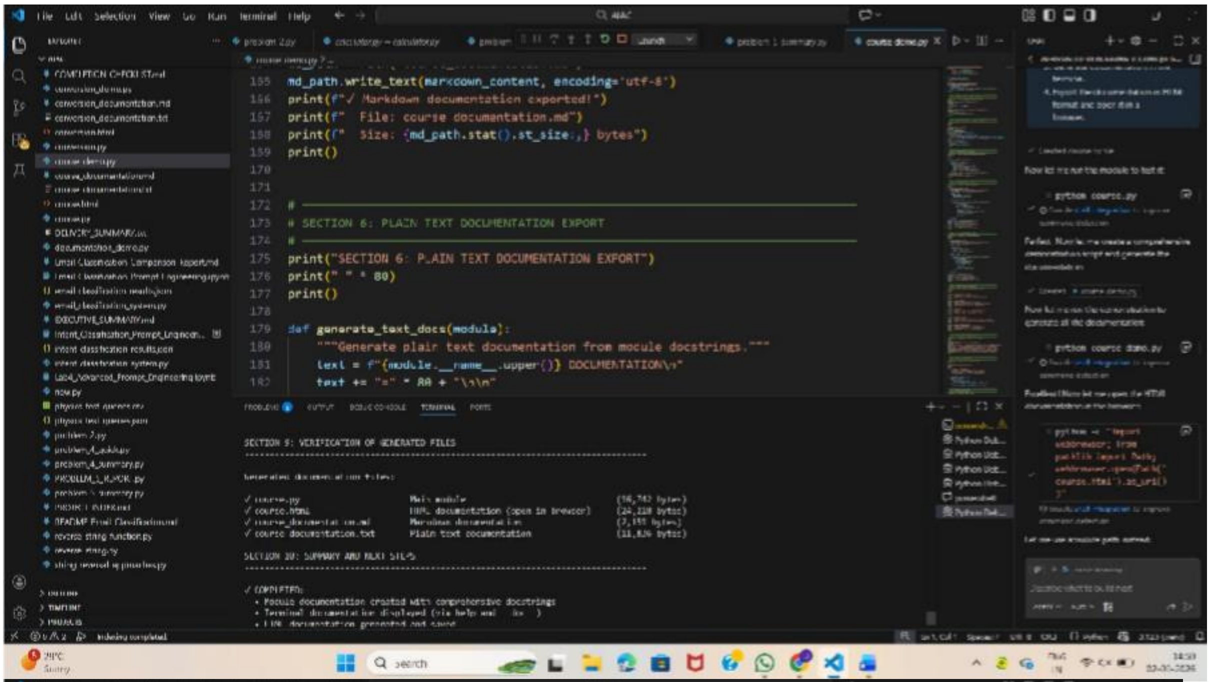
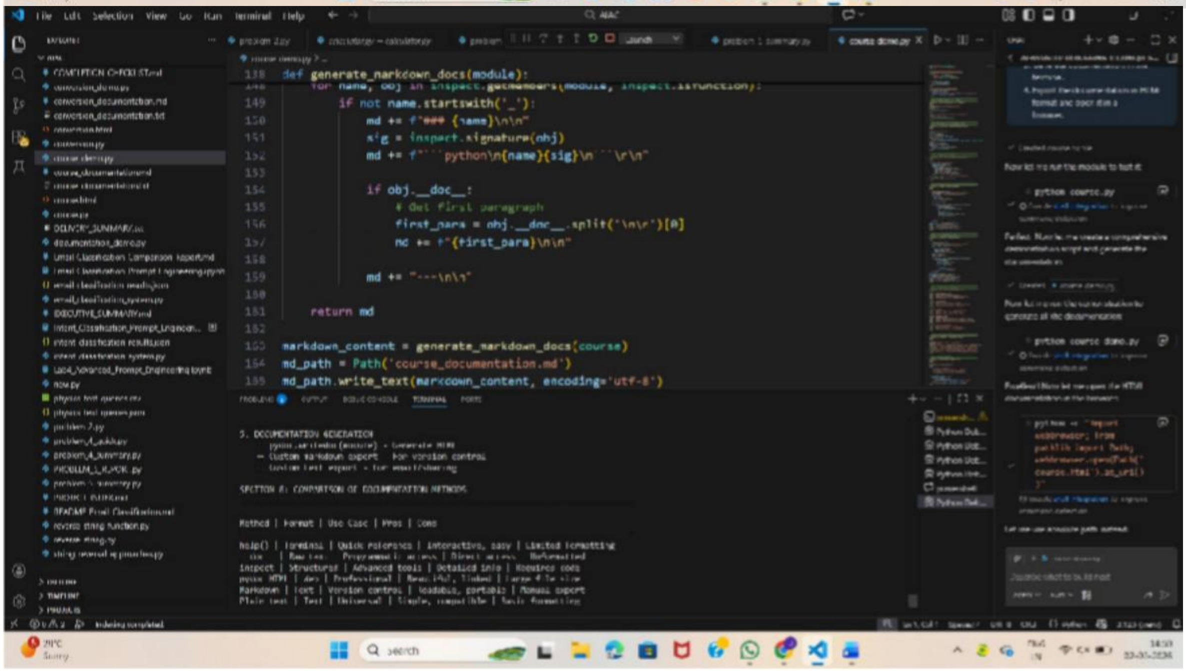
print("5. DOCUMENTATION GENERATION")
print("   ├─ pydoc.writedoc(module) - Generate HTML")
print("   ├─ Custom markdown export - For version control")
print("   └─ Custom text export - For email/sharing")
print()
# SECTION 8: COMPARISON OF DOCUMENTATION METHODS
print("SECTION 8: COMPARISON OF DOCUMENTATION METHODS")
print("-" * 80)
print()
print("Method | Format | Use Case | Pros | Cons")
print("-" * 80)
print("help() | Terminal | Quick reference | Interactive, easy | Limited formatting")
print("__doc__ | Raw text | Programmatic access | Direct access | Unformatted")
print("inspect | Structured | Advanced tools | Detailed info | Requires code")
print("pydoc HTML | Web | Professional | Beautiful, linked | Large file size")
print("Markdown | Text | Version control | Readable, portable | Manual export")
print("Plain text | Text | Universal | Simple, compatible | Basic formatting")
print()
# SECTION 9: VERIFYING GENERATED FILES
print("SECTION 9: VERIFICATION OF GENERATED FILES")
print("-" * 80)
print()

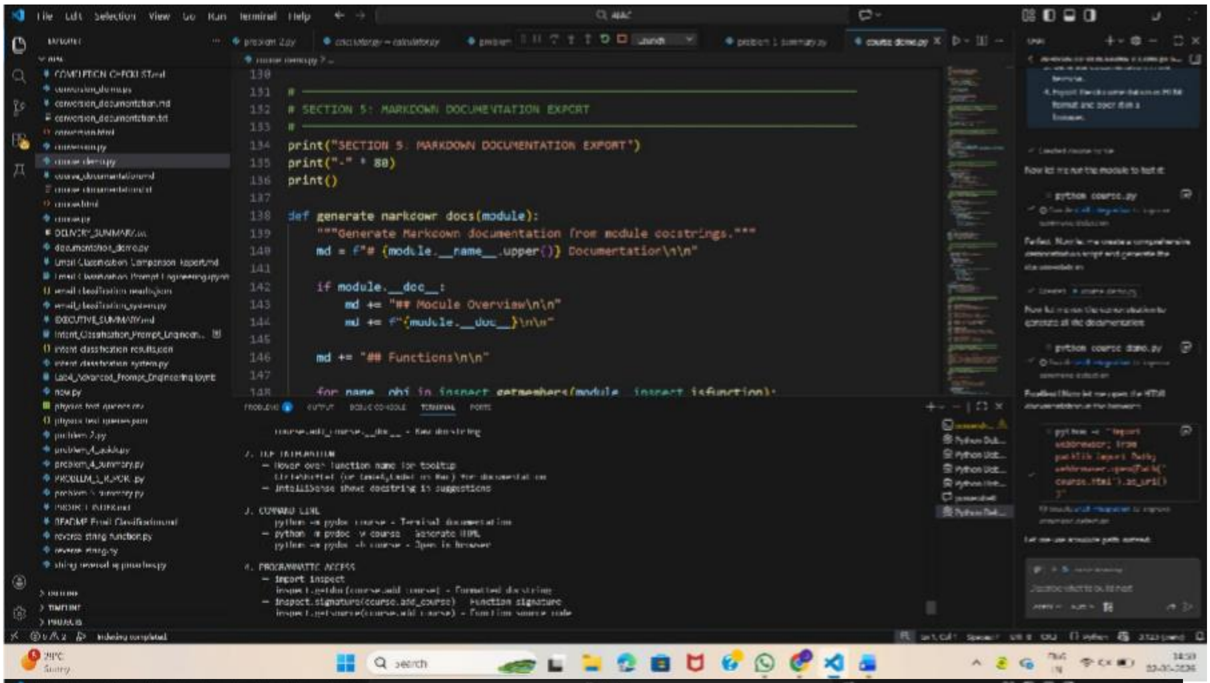


# SECTION 7: BEST PRACTICES FOR DOCUMENTATION
print("SECTION 7: BEST PRACTICES FOR DOCUMENTATION USAGE")
print("-" * 80)
print()
print("1. INTERACTIVE PYTHON SHELL")
print("   ├─ help(course) - Module documentation")
print("   ├─ help(course.add_course) - Function documentation")
print("   └─ course.add_course.__doc__ - Raw docstring")
print()
print("2. IDE INTEGRATION")
print("   ├─ Hover over function name for tooltip")
print("   ├─ Ctrl+Shift+I (or Cmd+K,Cmd+I on Mac) for documentation")
print("   └─ IntelliSense shows docstring in suggestions")
print()
print("3. COMMAND LINE")
print("   ├─ python -m pydoc course - Terminal documentation")
print("   ├─ python -m pydoc -w course - Generate HTML")
print("   └─ python -m pydoc -b course - Open in browser")
print()
print("4. PROGRAMMATIC ACCESS")
print("   ├─ import inspect")
print("   ├─ inspect.getdoc(course.add_course) - Formatted docstring")
print("   ├─ inspect.signature(course.add_course) - Function signature")
print("   └─ inspect.getsource(course.add_course) - Function source code")
print()

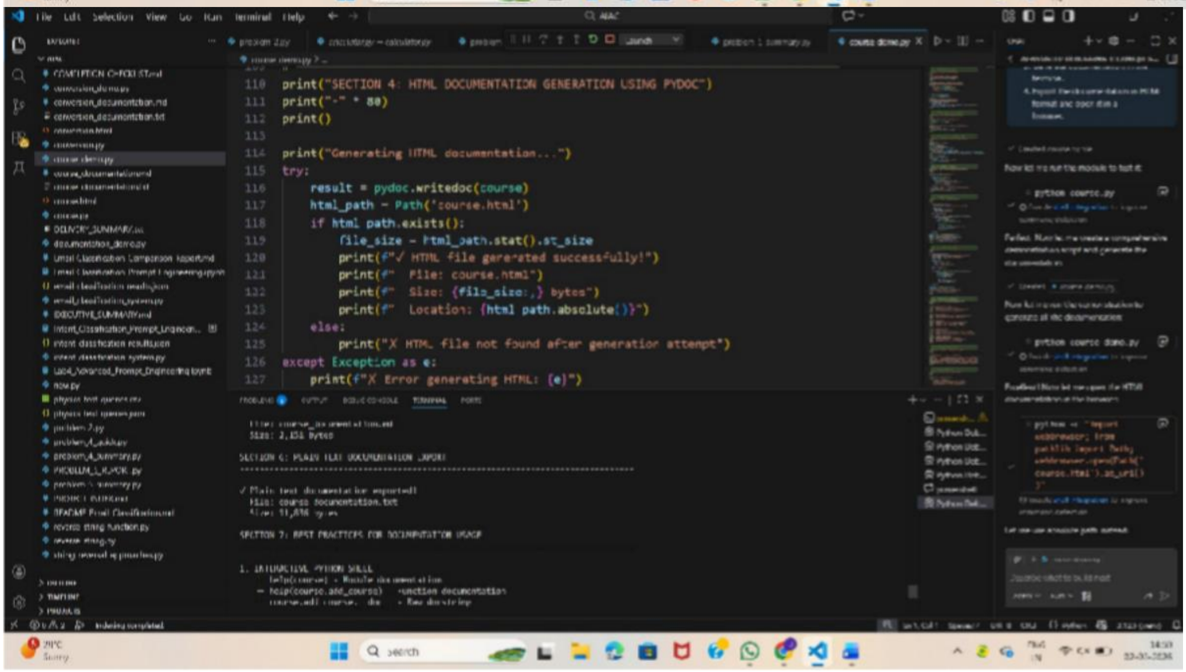

def generate_text_docs(module):
return text
text_content = generate_text_docs(course)
txt_path = Path('course_documentation.txt')
txt_path.write_text(text_content, encoding='utf-8')
print(f"✓ Plain text documentation exported!")
print(f" File: course_documentation.txt")
print(f" Size: {txt_path.stat().st_size:,} bytes")
print()
# SECTION 7: BEST PRACTICES FOR DOCUMENTATION
print("SECTION 7: BEST PRACTICES FOR DOCUMENTATION USAGE")
print("-" * 80)
NEXT STEPS:


def generate_text_docs(module):
if module.__doc__:
text += "MODULE OVERVIEW\n"
text += "-" * 80 + "\n\n"
text += f"{module.__doc__}\n\n\n"
text += "FUNCTIONS\n"
text += "=" * 80 + "\n\n"
for name, obj in inspect.getmembers(module, inspect.isfunction):
if not name.startswith('_'):
text += f"{name}\n"
sig = inspect.signature(obj)
text += f"{name}{sig}\n"
text += "-" * 80 + "\n\n"
if obj.__doc__:
text += f"{obj.__doc__}\n\n"
NEXT STEPS:

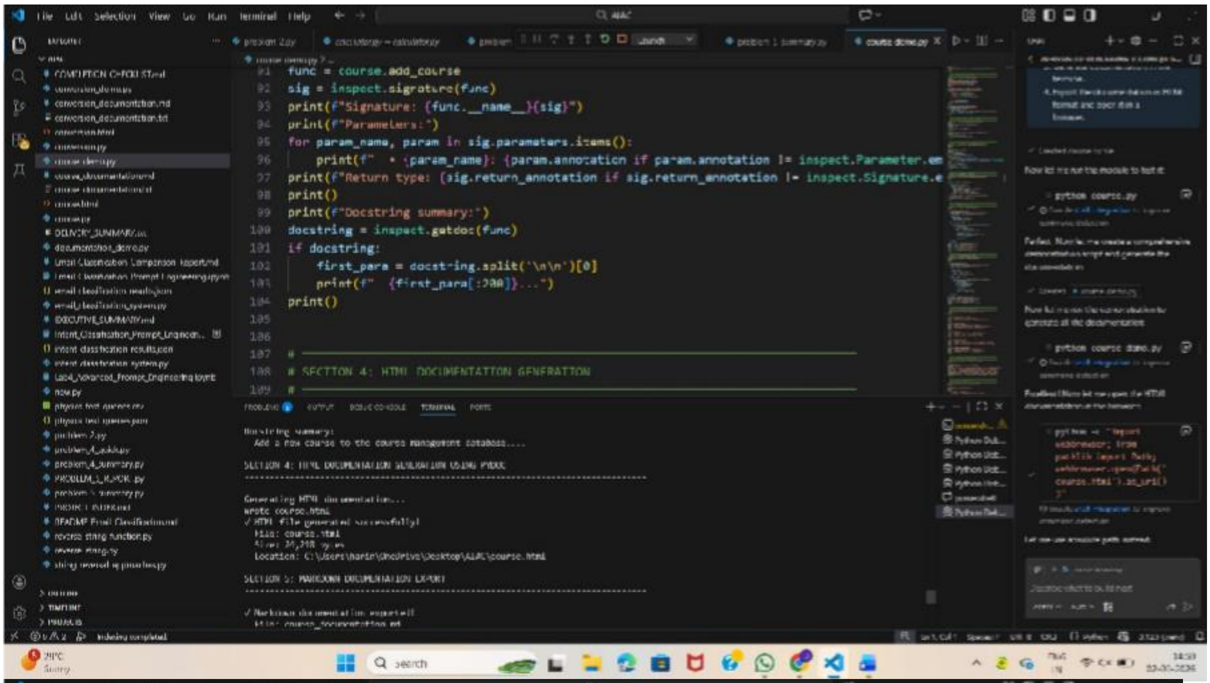

md_path.write_text(markdown_content, encoding='utf-8')
print(f"✓ Markdown documentation exported!")
print(f" File: course_documentation.md")
print(f" Size: {md_path.stat().st_size:,} bytes")
print()
# SECTION 6: PLAIN TEXT DOCUMENTATION EXPORT
print("SECTION 6: PLAIN TEXT DOCUMENTATION EXPORT")
print("=" * 80)
print()
def generate_text_docs(module):
"""Generate plain text documentation from module docstrings."""
SECTION 9: VERIFICATION OF GENERATED FILES
SECTION 10: SUMMARY AND NEXT STEPS

def generate_markdown_docs(module):
if not name.startswith('_'):
md += f"### {name}\n\n"
sig = inspect.signature(obj)
if obj.__doc__:
first_para = obj.__doc__.split('\n\n')[0]
md += f"{first_para}\n\n"
md += "---\n\n"
return md
markdown_content = generate_markdown_docs(course)
md_path = Path('course_documentation.md')
md_path.write_text(markdown_content, encoding='utf-8')
SECTION 8: COMPARISON OF DOCUMENTATION METHODS

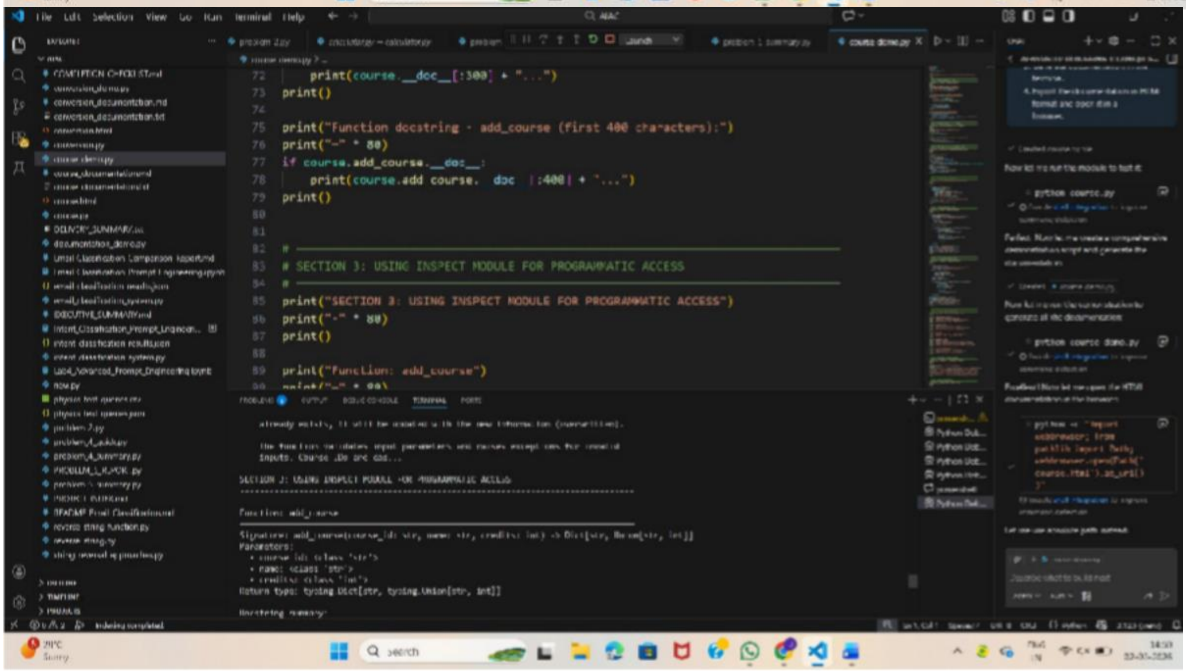


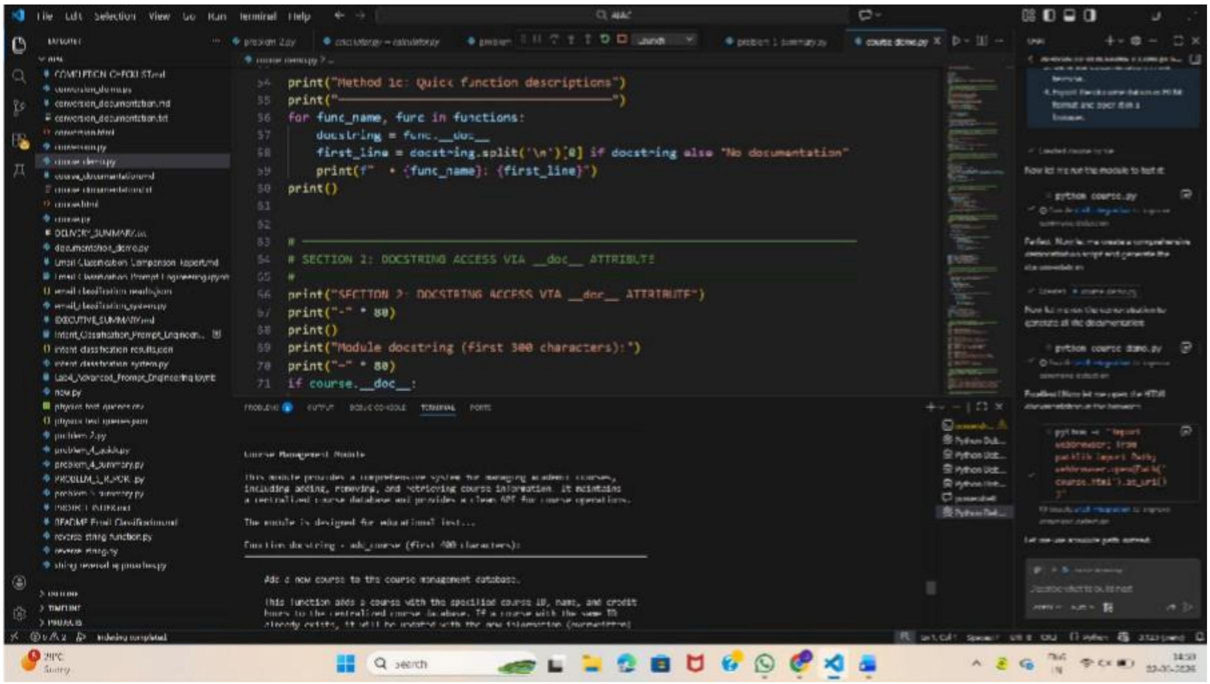
# SECTION 5: MARKDOWN DOCUMENTATION EXPORT
print("SECTION 5: MARKDOWN DOCUMENTATION EXPORT")
print("-" * 80)
print()
def generate_markdown_docs(module):
"""Generate Markdown documentation from module docstrings."""
md = f"# {module.__name__.upper()} Documentation\n\n"
if module.__doc__:
md += "## Module Overview\n\n"
md += f"{module.__doc__}\n\n"
md += "## Functions\n\n"
3. COMMAND LINE
4. PROGRAMMATIC ACCESS
import inspect



print("SECTION 4: HTML DOCUMENTATION GENERATION USING PYDOC")
print("-" * 80)
print()
print("Generating HTML documentation...")
try:
result = pydoc.writedoc(course)
html_path = Path('course.html')
if html_path.exists():
file_size = html_path.stat().st_size
print(f"✓ HTML file generated successfully!")
print(f" File: course.html")
print(f" Size: {file_size:,} bytes")
print(f" Location: {html_path.absolute()}")
else:
print("X HTML file not found after generation attempt")
except Exception as e:
print(f"X Error generating HTML: {e}")
SECTION 6: PLAIN TEXT DOCUMENTATION EXPORT
✓ Plain text documentation exported!
File: course_documentation.txt
SECTION 7: BEST PRACTICES FOR DOCUMENTATION USAGE
1. INTERACTIVE PYTHON SHELL


func = course.add_course
sig = inspect.signature(func)
print(f"Signature: {func.__name__}{sig}")
print(f"Parameters:")
for param_name, param in sig.parameters.items():
print(f" • {param_name}: {param.annotation if param.annotation != inspect.Parameter.em
print(f"Return type: {sig.return_annotation if sig.return_annotation != inspect.Signature.e
print()
print(f"Docstring summary:")
docstring = inspect.getdoc(func)
if docstring:
first_para = docstring.split('\n\n')[0]
print(f" {first_para[:200]}...")
print()
# SECTION 4: HTML DOCUMENTATION GENERATION
SECTION 4: HTML DOCUMENTATION GENERATION USING PYDOC
Generating HTML documentation...
wrote course.html
✓ HTML file generated successfully!
SECTION 5: MARKDOWN DOCUMENTATION EXPORT


print(course.__doc__[:300] + "...")
print()
print("Function docstring - add_course (first 400 characters):")
print("-" * 80)
if course.add_course.__doc__:
print(course.add_course.__doc__[:400] + "...")
print()
# SECTION 3: USING INSPECT MODULE FOR PROGRAMMATIC ACCESS
print("SECTION 3: USING INSPECT MODULE FOR PROGRAMMATIC ACCESS")
print("-" * 80)
print()
print("Function: add_course")
SECTION 3: USING INSPECT MODULE FOR PROGRAMMATIC ACCESS
Function: add_course
Parameters:
Return type: typing.Dict[str, typing.Union[str, int]]



print("Method 1c: Quick function descriptions")
for func_name, func in functions:
docstring = func.__doc__
first_line = docstring.split('\n')[0] if docstring else "No documentation"
print(f" • {func_name}: {first_line}")
print()
# SECTION 2: DOCSTRING ACCESS VIA __doc__ ATTRIBUTE
print("SECTION 2: DOCSTRING ACCESS VIA __doc__ ATTRIBUTE")
print("-" * 80)
print()
print("Module docstring (first 300 characters):")
print("-" * 80)
if course.__doc__:
Course Management Module
Add a new course to the course management database.
python course_demo.py