

# **ASSIGNMENT-9.1**

## **AI ASSISTED CODING**

**Name: Bethi Mounika**

**Roll No: 2303A52196**

**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.

This screenshot shows a Jupyter Notebook cell with code analysis results. The code prints various messages about documentation, advantages, disadvantages, and use cases. It also includes a section on approach 1: inline comments. The analysis sidebar on the right provides critical analysis, recommendations for mathematical utility, and justification.

```
576 print("Documentation accessed via help():")
577 print("  ✓ Readable")
578 print("  ✓ Quick and concise")
579 help(Find_max_simple_docstring)
580
581 print("\nADVANTAGES:")
582 print("  ✓ Quick and concise")
583 print("  ✓ Easy to write")
584 print("  ✓ Suitable for simple functions")
585 print("  ✓ Less maintenance overhead")
586
587 print("\nDISADVANTAGES:")
588 print("  X Lacks detail about parameters and return values")
589 print("  X No examples for users")
590 print("  X No error documentation")
591 print("  X Insufficient for library documentation")
592 print("  X IDE tooltips show minimal information")
593
594 print("\nUSE CASES:")
595 print("  • Simple utility functions")
596
597 # The first operand (that or int),
598 # the second operand (float or int).
599 # operation: A string representing the operation ('+', '-', '*', '/')
600
601 Returns:
602 The result of the operation as a float, or None if division by zero.
603
604 Raises:
605     ValueError: If operation is not one of '+', '-', '*', '/'.
606     TypeError: If x or y are not numeric types.
607
608 Examples:
609 >>> calculate_with_inline_comments(10, 5, '+')
610 15.0
611 >>> calculate_with_inline_comments(10, 5, '-')
612 5.0
```

Critical Analysis

- Look this is obfuscated and misleading
- Key insight: There are NO actually available - could be all three

Recommendation for Mathematical Utility

BEST APPROACH: Google-Style Docstrings

I believe Comments

Justification:

- 1. Used only in professional, academic, government and scientific
- 2. Longer algorithmic
- Implementation details visible
- Mathematical formulas are clearly explained (F(x), O(N))
- 4. Most document numerical stability and FTFI (FTFI) behavior
- 5. Comprehensive, using type hints, comprehensive docstrings

Optimal Example: find\_mean\_pythontidy()

Documentation protocol for mathematical utility calculations with:

- Implement longer algorithmic
- Use type consistency warning
- CCC TSD floating point constants
- Edge cases (float, Inf, etc.)
- Robotic examples
- Related function references

If this is ready to run and demonstrate all comments with associated code.

Cell 1 of 1 | Run Cell | Run All | Stop | Refresh | Help | 14:08 32-05-2026

This screenshot shows a Jupyter Notebook cell with code analysis results for approach 2: inline comments. The code defines a function to find the maximum value in a list. The analysis sidebar on the right provides critical analysis, recommendations for mathematical utility, and justification.

```
593
594 print("\nUSE CASES:")
595 print("  • Simple utility functions")
596 print("  • Internal helper functions")
597 print("  • Code already clear from type hints")
598 print("  • Quick prototypes")
599
600 # ===== APPROACH 2: INLINE COMMENTS =====
601
602 # ===== RUMPTY TUMPTY ENDINGS =====
603
604 print("  ✓ Readable")
605 print("  ✓ Approach 2: Inline comments")
606 print("  ✓ No need for separate docstring")
607
608 def find_max_with_inline_comments(numbers: List[Float]) -> float:
609     # Ensure we have at least one number to avoid ValueError
610     if not numbers:
611         raise ValueError("Cannot find max of empty list")
```

Critical Analysis

- Look this is obfuscated and misleading
- Key insight: There are NO actually available - could be all three

Recommendation for Mathematical Utility

BEST APPROACH: Google-Style Docstrings

I believe Comments

Justification:

- 1. Used only in professional, academic, government and scientific
- 2. Longer algorithmic
- Implementation details visible
- Mathematical formulas are clearly explained (F(x), O(N))
- 4. Most document numerical stability and FTFI (FTFI) behavior
- 5. Comprehensive, using type hints, comprehensive docstrings

Optimal Example: find\_mean\_pythontidy()

Documentation protocol for mathematical utility calculations with:

- Implement longer algorithmic
- Use type consistency warning
- CCC TSD floating point constants
- Edge cases (float, Inf, etc.)
- Robotic examples
- Related function references

If this is ready to run and demonstrate all comments with associated code.

Cell 1 of 1 | Run Cell | Run All | Stop | Refresh | Help | 14:08 32-05-2026

The screenshot displays a terminal window with two panes. The left pane shows a Python script named 'sum.py'. The right pane shows the output of the script, which is a detailed mathematical analysis of the sum operation. The output includes a 'Critical Analysis' section with bullet points and a 'Recommendation for Mathematical Utilities Library' section. The terminal interface includes tabs for 'PROBLEMS', 'CURRENT', 'BUILD-LOGS', 'TERMINAL', and 'PORTS'. A status bar at the bottom indicates '29PC' and 'Sunday'.



The screenshot shows a terminal window with two identical sessions of Python code. The code defines a function `find\_max\_optimized` that calculates the maximum value in a list. It uses the `statistics.mean` function to find the minimum value and compares it with each element in the list. It also handles empty lists by raising a `ValueError`. The code then uses `numpy.max` to calculate the maximum value again, noting that it ignores NaN values. A 'Critical Analysis' sidebar on the right provides a detailed breakdown of the code's strengths and weaknesses, including its readability, performance implications, and adherence to Google-style conventions.



The screenshot shows a terminal window with two panes. The left pane contains Python code for a 'math utilities' library, featuring docstrings and inline comments. The right pane is a sidebar with analysis and recommendations. The analysis section includes a 'Critical Analysis' box with bullet points about readability and maintainability, and a 'Recommendation for Mathematical Utilities Library' section suggesting Google-style docstrings. The 'Best Approach' section is titled 'Google-style Docstrings'. The 'Justification' section provides a detailed list of pros and cons for each approach. The terminal also displays system metrics like CPU usage and memory.





The screenshot shows a terminal window at the top and a Jupyter Notebook cell below it. The terminal window has tabs for 'ipython' and 'noway'. The Jupyter cell contains Python code for a 'find\_max\_google\_style' function and its docstring. It also includes sections for 'PEP 8 APPROACH 1: GOOGLE-STYLE DOCSTRING (SIMPLIFIED)' and 'PEP 8 APPROACH 3: GOOGLE-STYLE DOCSTRING (COMPREHENSIVE)'. A warning message 'PART 1: POSSIBLE LAYER INCONSISTENCIES' is displayed above the code. The bottom part of the screen shows a Jupyter notebook interface with a warning icon and the text 'PART 1: POSSIBLE LAYER INCONSISTENCIES'.

```

    # Test the function
    test_list = [3, 1, 4, 1, 5, 9, 2, 6]
    result = find_max_with_inline_comments(test_list)
    print(f"Example: find_max_with_inline_comments({test_list}) = {result}\n")

    print("ADVANTAGES:")
    print(" ✓ Explains implementation decisions (WHY, not WHAT)")
    print(" ✓ Helps future developers understand logic")
    print(" ✓ Documents edge cases handled")
    print(" ✓ Great for complex algorithms")
    print(" ✓ Visible directly in source code")

    print("\nDISADVANTAGES:")
    print(" X Not accessible via help() or TOC tooltips")
    print(" X Easy to become outdated during refactoring")
    print(" X Can clutter code if overdone")
    print(" X Not standardized format")
    print(" X Requires reading source code to understand API")
    print(" X Poor for API documentation generation")

```

**Critical Analysis**

- Locate this code block and redesign.
- See Insights: There are 102 static code analysis violations in this code block.

**Recommendation for Mathematical Utility Library**

**BEST PRACTICE** Google-Style Docstrings

**Inline Comments**

**Justification:**

- Used only in professional code generation and documentation.
- Comments appear in code.
- Documentation is visible when running analysis tools like complexity analysis (Pylint, McCabe).
- Most document generated statically and PEP 8 follows.
- Comments clearly explain logic or code behavior for your audience.

**Optimal Example: find\_max\_with\_inline()**

Demonstrates professional mathematical library documentation with:

- Longer longer-style docstring.
- Issue/piece moderately warning.
- PEP 8 following painless documentation.
- Page: www.Pydoc.info/
- Realistic example.
- Related functions references.

This block is ready to run and demonstrates all elements with associated code.

## 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.

The screenshot shows a code editor with a sidebar titled "Google-Style Docstring Guidance". The main pane displays a Python file with annotations for a specific function:

```
def onboarding_outcome():
    """Onboarding Outcome

    This function provides a detailed overview of the onboarding process for new developers. It includes examples, best practices, and troubleshooting tips.

    Args:
        None

    Returns:
        str: A multi-line string containing the onboarding outcome.

    Examples:
        >>> onboarding_outcome()
        'Welcome to our team! This function provides a detailed overview of the onboarding process for new developers. It includes examples, best practices, and troubleshooting tips.'
```

The sidebar contains sections like "Key Definitions", "Best-Practice Guidelines", "Web-Google-style", and "Production-Ready Example".

The screenshot shows a code editor with a sidebar titled "K8Y-Style Docstring Guidance". The main pane displays a Python file with annotations for a specific function:

```
def onboarding_outcome():
    """K8Y-Style Docstring

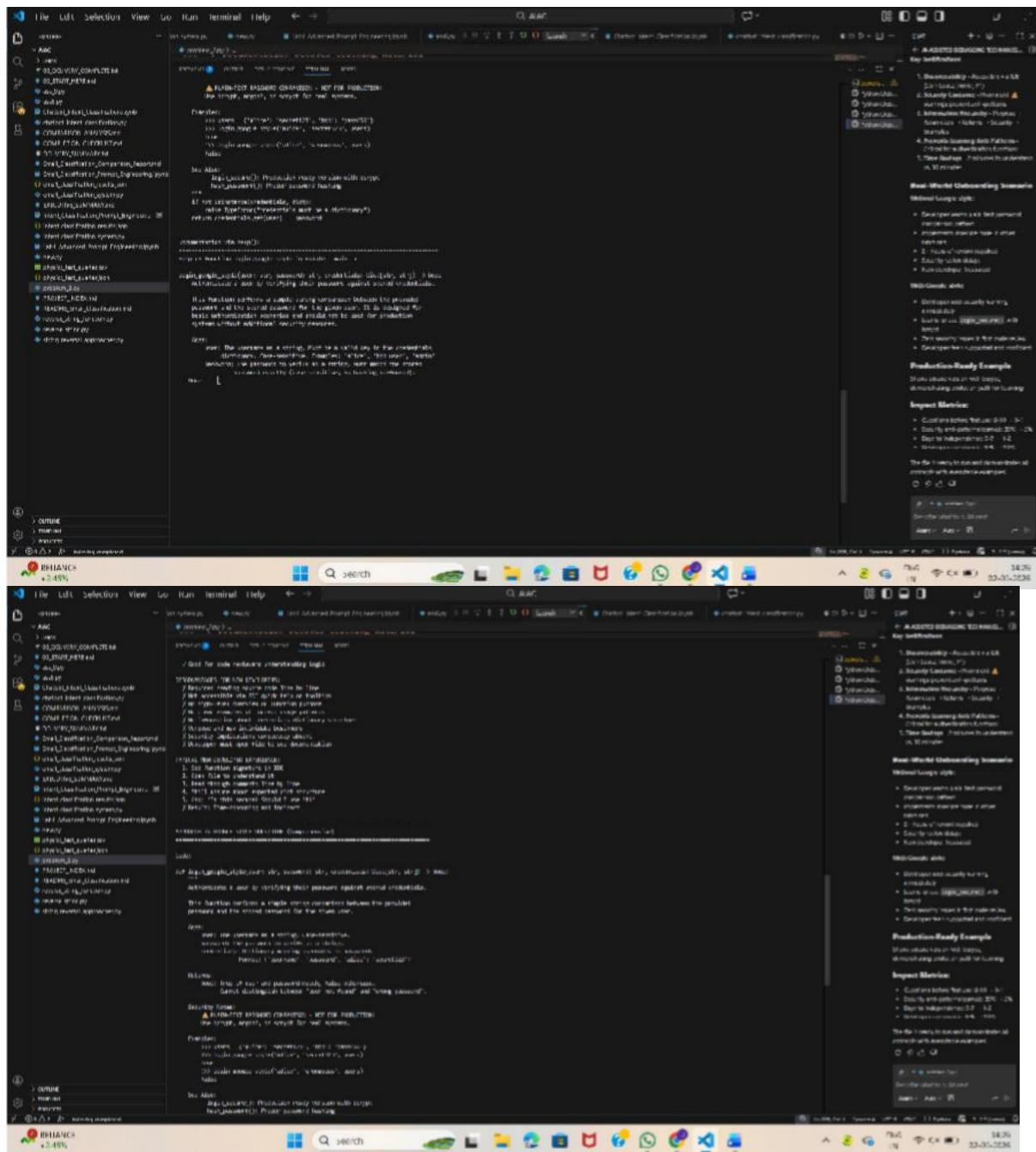
    This function provides a detailed overview of the onboarding process for new developers. It includes examples, best practices, and troubleshooting tips.

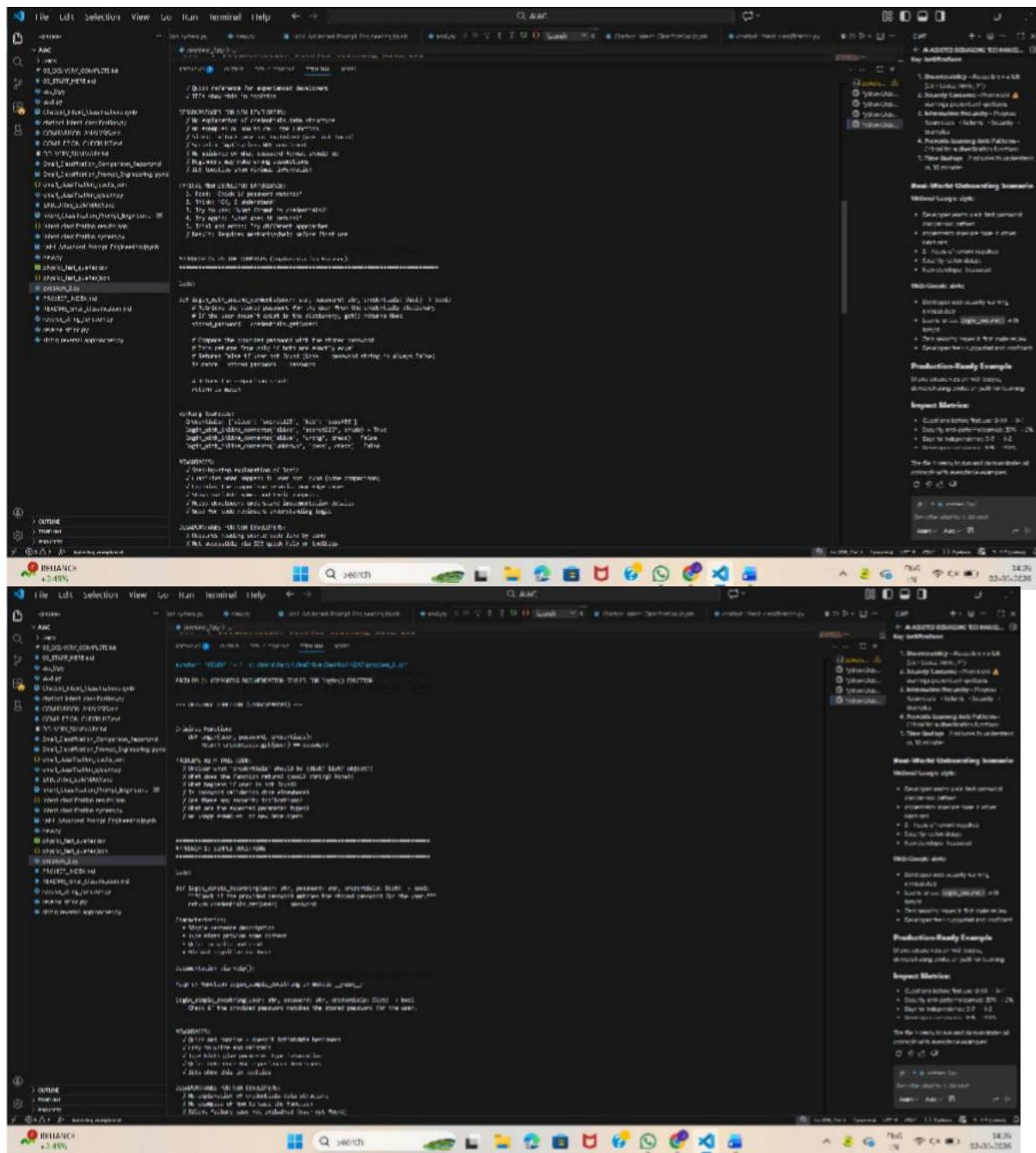
    Args:
        None

    Returns:
        str: A multi-line string containing the onboarding outcome.

    Examples:
        >>> onboarding_outcome()
        'Welcome to our team! This function provides a detailed overview of the onboarding process for new developers. It includes examples, best practices, and troubleshooting tips.'
```

The sidebar contains sections like "Key Definitions", "Best-Practice Guidelines", "Web-Google-style", and "Production-Ready Example".





The screenshot displays a software interface for reviewing code, likely a pull request or merge request tool. It features two main panes showing code snippets with inline annotations. The top pane shows a snippet of Python code with annotations like 'Developers feel supported and guided' and 'Developers feel supported and guided'. The bottom pane shows a snippet of Java code with annotations such as 'Questions before first use', 'Time to first correct usage', 'Security anti-patterns found', 'Correct usage without asking', 'Code review security issues', 'How to develop independently', and 'Developer confidence level'. Both panes include sections for 'Review Summary', 'Review Details', and 'Review History'. On the right side, there is a sidebar titled 'Key Information' and 'Recent Activity' which lists various pull requests and merge requests. The bottom right corner shows a 'Metrics' section with performance data.





**UWE:** 00:00:00 - Wasted 2+ hours of developer time  
X security review delayed  
X how does logic function and continue  
X add first impression of codebase

**Mentor:** 00:00:00 - Hey Uwe! I'm sorry for the delay.  
I've been working on some other tasks and got pulled away from this project.  
I'll make sure to catch up with you as soon as possible.

**UWE:** 00:00:00 - No problem at all! I understand.  
I just need to get this done by Friday.  
X security review delayed  
X how does logic function and continue  
X add first impression of codebase

**Mentor:** 00:00:00 - Understood. I'll prioritize this task and make sure to finish it by Friday.  
I'll keep you updated on my progress.

**UWE:** 00:00:00 - Great, thanks!

**Mentor:** 00:00:00 - You're welcome! If you have any questions or need further assistance, feel free to ask.

**UWE:** 00:00:00 - All right, thanks again!

**Mentor:** 00:00:00 - You're welcome! Safe travels!

**UWE:** 00:00:00 - See you!

**Mentor:** 00:00:00 - See you!

**UWE:** 00:00:00 - Bye!

**Mentor:** 00:00:00 - Bye!





### Problem 3: Calculator (Automatic Documentation Generation)

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

### Instructions:

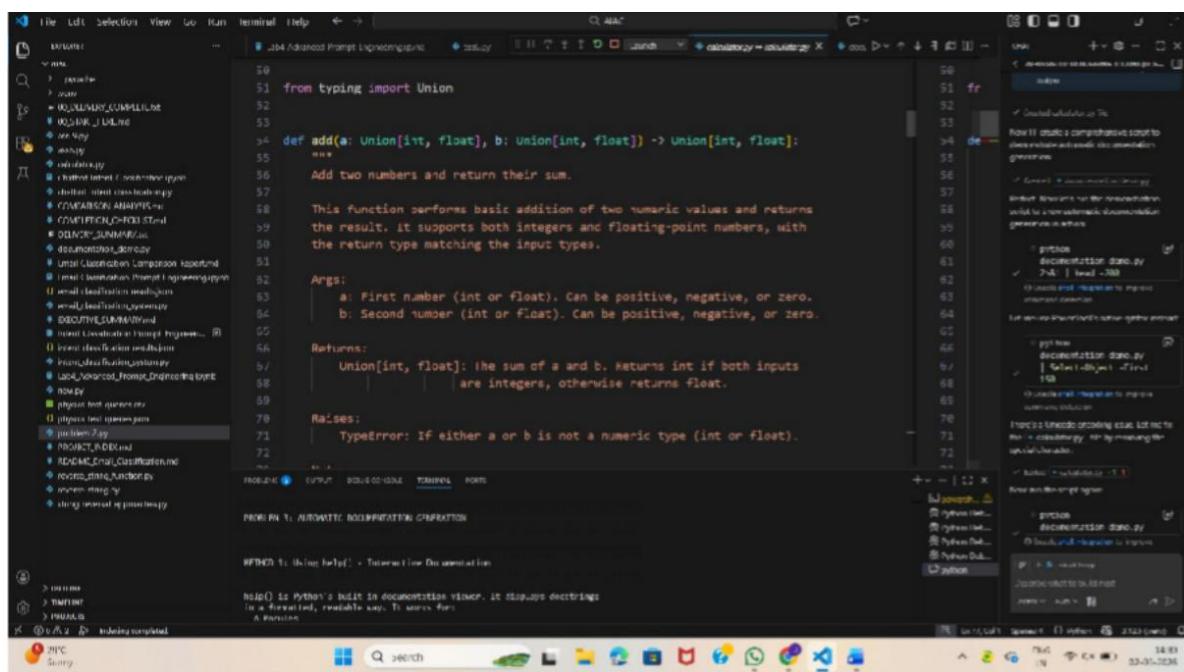
1. Create a Python module calculator.py that includes the following functions, each written with appropriate docstrings:

- 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



```

$ pydoc add
Help on function add in module __main__:

add(a, b)
    Add two numbers and return their sum.

    This function performs basic addition of two numeric values and returns
    the result. It supports both integers and floating-point numbers, with
    the return type matching the input types.

    Args:
        a: First number (int or float). Can be positive, negative, or zero.
        b: Second number (int or float). Can be positive, negative, or zero.

    Returns:
        Union[int, float]: The sum 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).

```

The screenshot shows a Jupyter Notebook environment with several tabs open. The main code cell contains Python code for a function named `add`. A tooltip from the IDE's code completion feature is displayed over the `+ operator` in the docstring, providing information about its behavior with different input types (integers vs. floats) and its time and space complexity.

**Code Cell:**

```
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:  
    """
```

**Help() Output:**

```
Help on function add in module __main__:  
  
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:  
    """
```

**Bottom Status Bar:**

Jupyter Notebook interface status bar showing file paths, line numbers, and other metadata.

A screenshot of a terminal window titled "Python 3.8.5 (v3.8.5:580f8d8, Jul 29 2020, 15:53:45) [MSC v.1916 64 bit (AMD64)]" running on Windows 10. The user has run the command "help()". The output shows the help function's docstring, which describes it as a built-in function that prints documentation for modules, classes, methods, or other objects. It includes examples of help(module), help(object), and help(function). The terminal also displays the Python documentation page at "https://docs.python.org/3/library/functions.html#help". A status bar at the bottom shows "2020-08-01 15:53:45" and "2020-08-01 15:53:45".

The screenshot shows a Jupyter Notebook environment with the following details:

- File**, **Edit**, **Selection**, **View**, **Go**, **Run**, **Terminal**, **Help**, **→**, **Cell**, **Run**.
- Untitled** notebook.
- Cell** 312: `if __name__ == "__main__":`
- Cell** 344:  `result = add("5", 3)`
- Cell** 345: `except TypeError as e:`
- Cell** 346:  `print(f"add('5', 3) raised {e.__class__.__name__}: {e}")`
- Cell** 347: `348`
- Cell** 349: `349`
- Variables** pane: `result` (int, value: 8).
- Locals** pane:
  - Adding two positive integers: `349 add(5, 3)`
  - Adding two negative numbers: `349 add(-5, -3)`
  - Adding floating-point numbers: `349 add(5.1, 2.0)`
  - Adding mixed types: integers & float: `349 add(5, 3.0)`
  - Adding zero: `349 add(0, 3)`
- Suggestions**:
  - add(): Add two numbers
  - multiply(): Multiply two numbers
  - divide(): Divide two numbers
- Documentation**:
  - divide(`b1`, `b2`) → float  
Divides the first number by the second and return the quotient.
  - This function performs division (`a / b`), returning a floating point result.  
Note that `a` doesn't have to be an int and will be an `float` as well.
  - Facebook (most recent call last):  
File "C:\Users\user\PycharmProjects\ATM\Calculator\calculator.py", line 31, in divide  
 print(f"\n{a} / {b} = {result}\n")  
File "C:\Users\user\PycharmProjects\ATM\Calculator\calculator.py", line 317, in <module>  
 return calc.divide(calc.add(calc.int\_to\_string(result), calc.multiply(result, 1000)))
- Intelligent completion**: Shows `divide()` with parameters: `int, float, float`.
- Help** pane: `divide` documentation.
- Output** pane: `349 add(5, 3)`.
- Bottom status bar**: `Untitled notebook`, `divide()`, `349 add(5, 3)`.
- System tray**: `29°C`, `Sunny`.



The image displays a dual-monitor setup for Python development. The left monitor shows a terminal window with Python code for generating documentation, specifically for a 'calculator' module. The code uses the `help()` function to generate docstrings and then prints them to the console. Below the terminal is a file browser showing various Python files like 'calculator.py', 'calculator\_error\_handling.py', and 'calculator\_math.py'. The right monitor shows a documentation viewer with the generated API documentation for the 'calculator' module, displaying sections for basic functionality, floating-point numbers, edge cases, and error handling.

The screenshot displays a dual-monitor setup for Python development. Both monitors show a Jupyter Notebook interface with code snippets and output.

**Monitor 1 (Left):**

- Code cell 1:

```
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
```

- Code cell 2:

```
# DEMONSTRATION CODE (When run as main)  
# ======  
#  
if __name__ == "__main__":  
    print("= " * 80)  
    print("CALCULATOR MODULE - DEMONSTRATION")  
    print("= " * 80)
```

PROBLEMS: 0 OUTPUT: 0 DOCS: 0 TERMINAL: 0 NOTEBOOKS: 0

**Monitor 2 (Right):**

  - Code cell 1:

```
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
```

  - Code cell 2:

```
# DEMONSTRATION CODE (When run as main)  
# ======  
#  
if __name__ == "__main__":  
    print("= " * 80)  
    print("CALCULATOR MODULE - DEMONSTRATION")  
    print("= " * 80)
```

PROBLEMS: 0 OUTPUT: 0 DOCS: 0 TERMINAL: 0 NOTEBOOKS: 0

The screenshot illustrates a dual-monitor setup. The left monitor displays a Jupyter notebook cell containing the following text:

```
def divide(a: Union[int, float], b: Union[int, float]) -> float:
    Args:
        a: First number (dividend). Must be a numeric type (int or float).
        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)
```

The right monitor displays the corresponding Python code for the 'divide' function:

```
def divide(a: Union[int, float], b: Union[int, float]) -> 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.
```

The screenshot displays a dual-monitor setup for Python development. The left monitor features a terminal window with code for the `multiply` and `subtract` functions. The `multiply` function is annotated with type hints and detailed docstrings, including examples of multiplication by zero and commutativity. The `subtract` function also includes a `See Also` section pointing to related methods like `add()`, `multiply()`, and `divide()`. The right monitor shows a file browser with various Python files and a terminal window displaying code for the `divide` function and its documentation. A tooltip from the documentation viewer on the left monitor provides information about the `help()` function.

The screenshot displays two side-by-side views of a Python development environment, likely PyCharm, illustrating the use of type hints and automatic documentation generation.

**Left Monitor (Jupyter Notebook View):**

- Code:**

```
113 def subtract(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:  
114     """  
115         - Subtraction is not commutative: a - b != b - a (except for zero).  
116         - Result type follows same rules as add().  
117         - Floating-point precision may apply.  
118         - Time complexity: O(1)  
119         - Space complexity: O(1)  
120  
121 Examples:  
122     Basic subtraction:  
123     >>> subtract(10, 3)  
124     7  
125  
126     Subtraction resulting in negative:  
127     >>> subtract(3, 10)  
128     -7  
129  
130     Subtracting negative numbers (adds):  
131     >>> subtract(3, -3)  
132     6  
133  
134     Subtracting from zero:  
135     >>> subtract(0, 5)  
136     -5  
137  
138 PROBLEM 3: AUTOMATIC DOCUMENTATION GENERATION
```
- Help Text:**

Help() is Python's built-in documentation viewer. It displays docstrings in a formatted, readable way. To work for a function:

**Right Monitor (Standard Python Script View):**

- Code:**

```
34 def add(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:  
35     """  
36         Adding zero:  
37         >>> add(0, 5)  
38         5  
39  
40     See Also:  
41         - subtract(): Subtract two numbers  
42         - multiply(): Multiply two numbers  
43         - divide(): Divide two numbers  
44     """  
45  
46     if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):  
47         raise TypeError("Both a and b must be numeric types (int or float)")  
48  
49     return a + b  
50  
51  
52 def subtract(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:  
53     """  
54         Subtract the second number from the first and return the difference.  
55  
56         This function performs basic subtraction, computing a - b. It supports both integers and floating-point numbers. The order of operands matters:  
57         subtraction(1, 2) returns 2, but subtraction(2, 1) returns -1  
58  
59 PROBLEM 3: AUTOMATIC DOCUMENTATION GENERATION
```
- Help Text:**

Help() is Python's built-in documentation viewer. It displays docstrings in a formatted, readable way. To work for a function:

## Problem 4: Conversion Utilities Module

## Task:

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

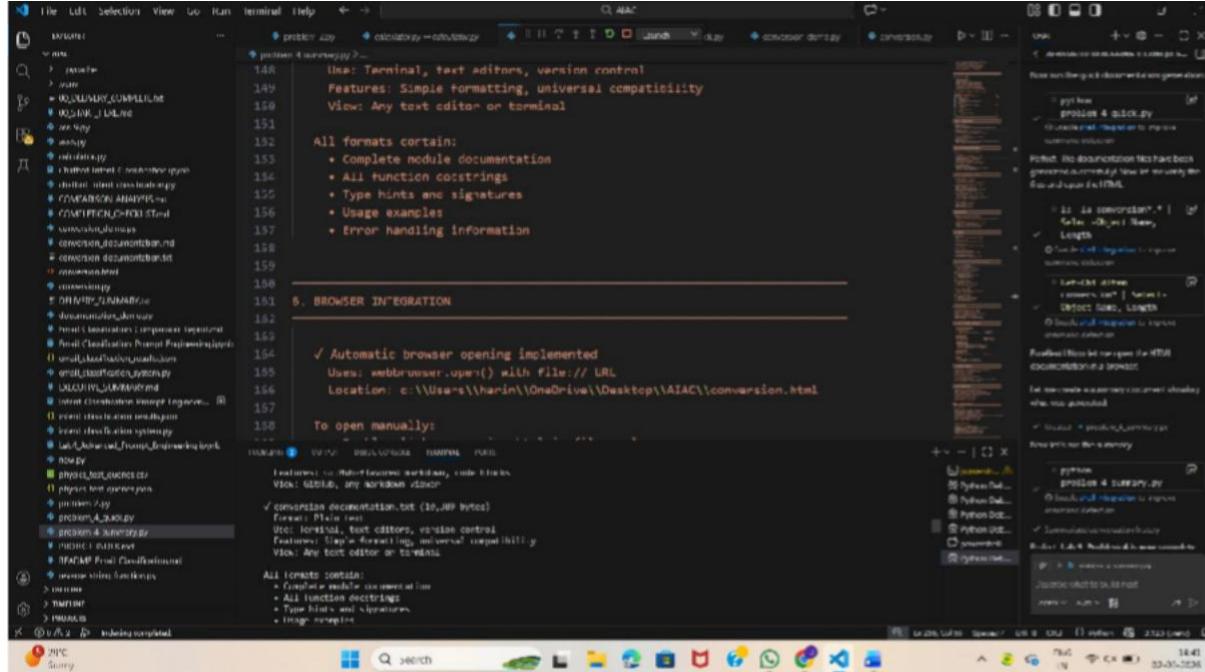
- o decimal\_to\_binary(n)
  - o binary\_to\_decimal(b)
  - o decimal\_to\_hexadecimal(n)

## 2. Use Copilot for auto-generating docstrings.

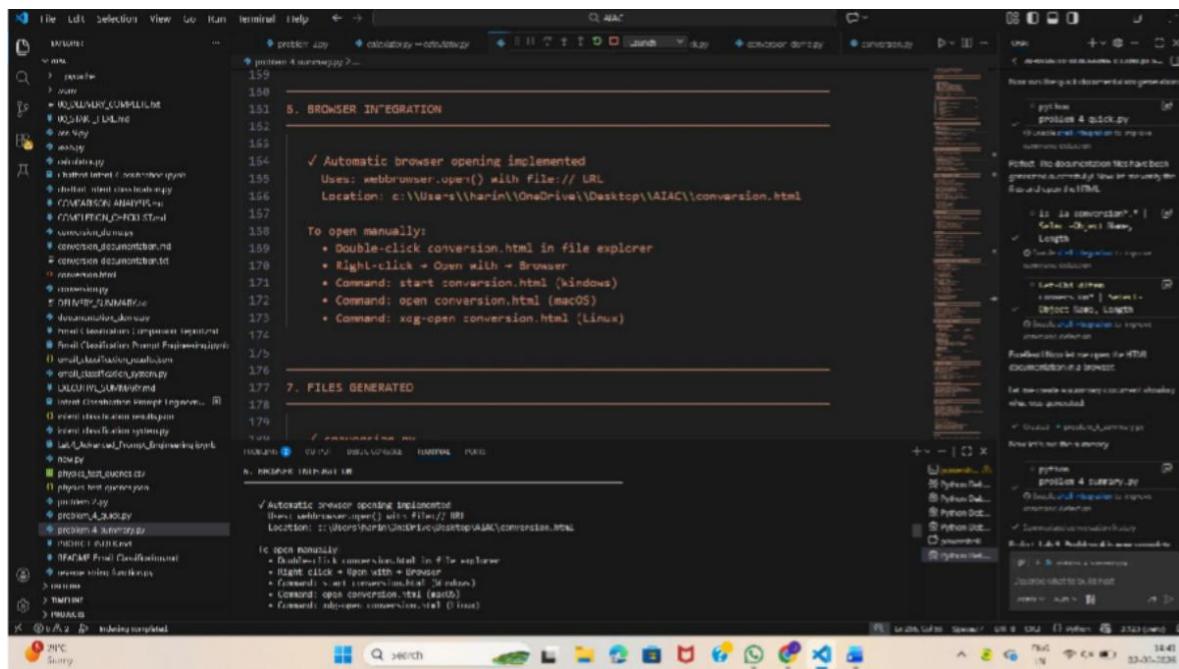
### 3. Generate documentation in the terminal.

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

#### Browser



The screenshot shows a terminal window titled "CL\_AIACT" with the command "copilot 4 summary.py" running. The output displays the generated documentation for "conversion". It includes sections for "BROWSER INTEGRATION" and "FILES GENERATED", detailing how to open the documentation manually or automatically via a browser. The terminal interface includes tabs for "problem 4", "copilot 4 - conversion.py", "copilot 4 summary.py", "conversion", and "conversion.html". A status bar at the bottom shows "1440" and "22-03-2026".



This screenshot is identical to the one above, showing the same terminal session and AIAC documentation generation process. The output from "copilot 4 summary.py" is identical, providing instructions for opening the generated HTML file ("conversion.html") in a browser. The terminal tabs and status bar are also the same.

```
File Edit Selection View Go Run Terminal Help < > C:\Users\user\PycharmProjects\conversion> python problem_4_summary.py
379     'problem_4_quick.py': 'Quick generation script',
380 }
381
382 for filename, description in conversion_files.items():
383     if os.path.exists(filename):
384         size = os.path.getsize(filename)
385         print(f" {filename:<35} {size:>8,} bytes - {description}")
386
387 print("\n" + "=" * 80)
388 print("PROBLEM 4 COMPLETE - ALL TASKS ACCOMPLISHED")
389 print("=" * 80)

Key Files:
+ conversion.py Main module (to use in your projects)
+ conversion.html Automatic documentation (for sharing with others)
+ conversion.pod.rst Automatic documentation (for GitHub)
+ problem_4_quick.py - Quick generation script
```

The terminal output shows the execution of the script, which prints a table of file names, sizes, and descriptions. It then prints a separator line, followed by a message indicating that all tasks are accomplished.

```
File Edit Selection View Go Run terminal Help > problem_4_summary.py calculator -> calculator.py launcher.py conversor_demo.py conversor.py

# problem_4_summary.py
379     'problem_4_quick.py': 'Quick generation script',
380 }
381
382 for filename, description in conversion_files.items():
383     if os.path.exists(filename):
384         size = os.path.getsize(filename)
385     print(f"\n{filename:<35} {size:>8} bytes - {description}")
386
387 print("\n" + "=" * 80)
388 print("PROBLEM 4 COMPLETE - ALL TASKS ACCOMPLISHED")
389 print("=" * 80)
390

CONCLUSION:
Problem 4 has been successfully completed. The conversion utilities module demonstrates professional-grade documentation practices:
+ source code documentation that automatically generates all outputs
+ multi-file documentation (HTML, Markdown, text)
+ Professional HTML output suitable for websites
+ Best practices in string writing
+ Automatic documentation generation using Python's built-in tools
```

```
File Edit Selection View Go Run terminal Help > problem_4_summary.py calculator -> calculator.py launcher.py conversor_demo.py conversor.py

# problem_4_summary.py
353     print(summary)
354
355     # Show file listings
356     print("\n" + "=" * 80)
357     print("GENERATED FILES IN CURRENT DIRECTORY")
358     print("=" * 80)
359
360     print("\n\n Conversion-related files:\n")
361
362     conversion_files = [
363         'conversion.py': 'Main module with conversion functions',
364         'conversion.html': 'HTML documentation (open in browser)',
365         'conversion_documentation.md': 'Markdown documentation',
366         'conversion_documentation.txt': 'Plain text documentation',
367         'conversion_demo.py': 'Full demonstration script',
368         'problem_4_quick.py': 'Quick generation script',
369     ]
370
371     for filename, description in conversion_files.items():
372         if os.path.exists(filename):
373             print(f"\n{filename:<35} {os.path.getsize(filename)} bytes - {description}")

CONCLUSION:
Problem 4 has been successfully completed. The conversion utilities module demonstrates professional-grade documentation practices:
+ source code documentation that automatically generates all outputs
+ multi-file documentation (HTML, Markdown, text)
+ Professional HTML output suitable for websites
+ Best practices in string writing
+ Automatic documentation generation using Python's built-in tools
```

```
File Edit Selection View Go Run Terminal Help + - ○ CLEAR
```

```
problem_4.py documentation --outputdir=.
```

```
CONCLUSION:
```

```
341 Problem 4 has been successfully completed. The conversion utilities module demonstrates professional-grade documentation practices:
```

```
342
343 * Source code documentation that automatically generates all formats
344 * Multiple documentation delivery methods (terminal, HTML, Markdown, text)
345 * Professional HTML output suitable for websites
346 * Best practices in docstring writing
347 * Automatic documentation extraction using Python's built-in tools
348
349 The module is production-ready and demonstrates how proper documentation enables automatic generation of multiple output formats from a single source.
```

```
350
351 Key Files:
352     - conversion.py - Main module (to use in your projects)
353     - conversion.html - HTML documentation (to share with others)
354     - conversion_demo.py - Complete demonstration script
355     - problem_4_quick.py - Quick generation script
```

```
356
357 CONCLUSION:
358 Problem 4 has been successfully completed. The conversion utilities module demonstrates professional-grade documentation practices:
359
360 * Source code documentation that automatically generates all formats
361 * Multiple documentation delivery methods (terminal, HTML, Markdown, text)
362 * Professional HTML output suitable for websites
363 * Best practices in docstring writing
364 * Automatic documentation extraction using Python's built-in tools
```

```
365
366 Additional Achievements:
367     - Generated binary documentation
368     - Generated text-based documentation
369     - Implemented error handling with meaningful messages
370     - Added type hints to all functions
371     - Created comprehensive demonstration scripts
372     - Showed multiple documentation access methods
373     - Provided usage examples and best practices
```

```
374
375 CONCLUSION:
```

```
376 Problem 4 has been successfully completed. The conversion utilities module demonstrates professional-grade documentation practices:
```

```
377
378 * Source code documentation that automatically generates all formats
379 * Multiple documentation delivery methods (terminal, HTML, Markdown, text)
380 * Professional HTML output suitable for websites
381 * Best practices in docstring writing
382 * Automatic documentation extraction using Python's built-in tools
383
384 The module is production-ready and demonstrates how proper documentation enables automatic generation of multiple output formats from a single source.
```

```
385
386 Additional Achievements:
387     - Generated binary documentation
388     - Generated text-based documentation
389     - Implemented error handling with meaningful messages
390     - Added type hints to all functions
391     - Created comprehensive demonstration scripts
392     - Showed multiple documentation access methods
393     - Provided usage examples and best practices
```

```
394
395 CONCLUSION:
```

```
396 Problem 4 has been successfully completed. The conversion utilities module demonstrates professional-grade documentation practices:
```

```
397
398 * Source code documentation that automatically generates all formats
399 * Multiple documentation delivery methods (terminal, HTML, Markdown, text)
400 * Professional HTML output suitable for websites
401 * Best practices in docstring writing
402 * Automatic documentation extraction using Python's built-in tools
403
404 The module is production-ready and demonstrates how proper documentation enables automatic generation of multiple output formats from a single source.
```

```
405
406 Additional Achievements:
407     - Generated binary documentation
408     - Generated text-based documentation
409     - Implemented error handling with meaningful messages
410     - Added type hints to all functions
411     - Created comprehensive demonstration scripts
412     - Showed multiple documentation access methods
413     - Provided usage examples and best practices
```

**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

---

**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

---

**12. TESTING AND VERIFICATION**

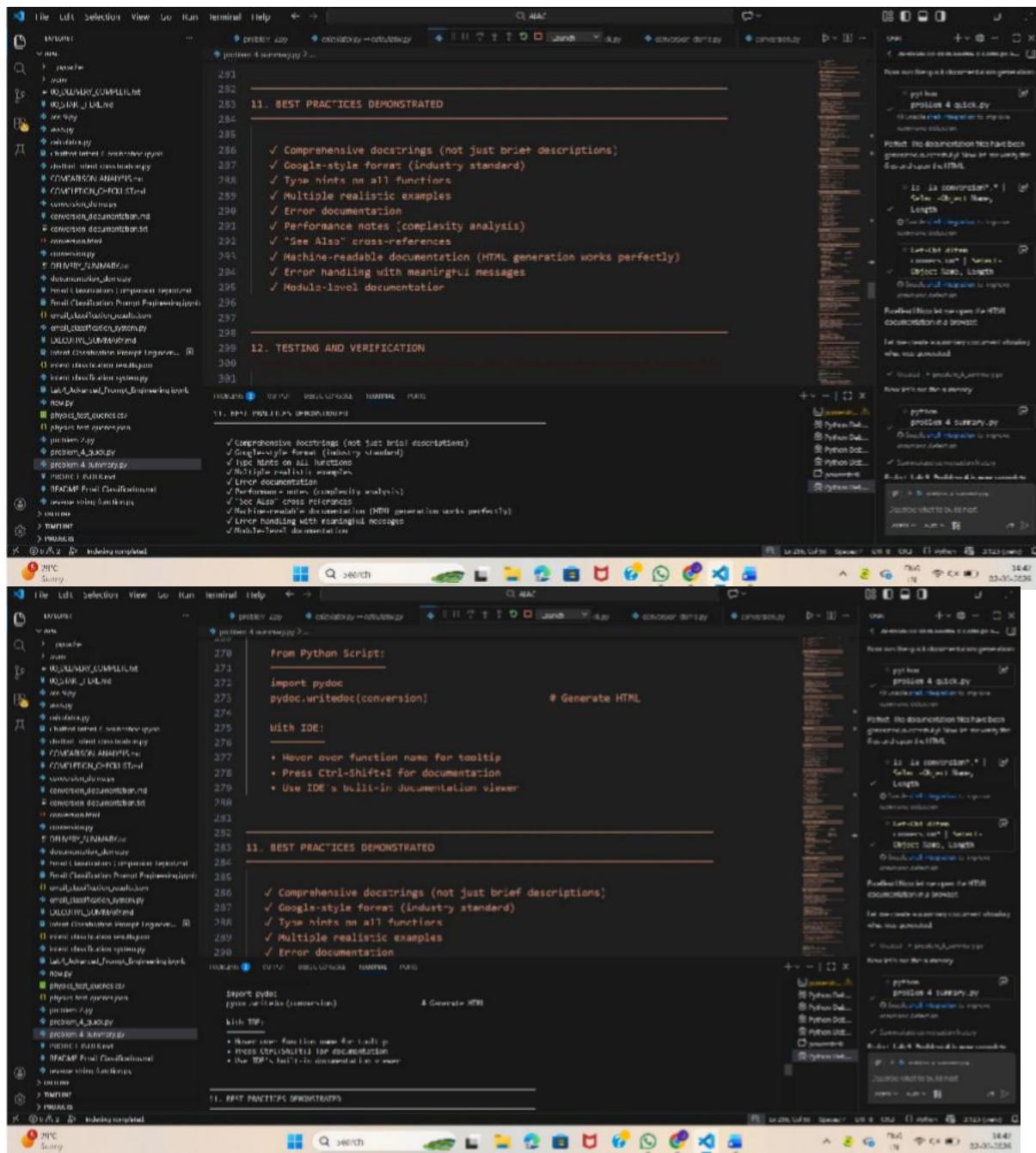
- All functions tested with valid inputs
- Edge cases tested (0, power of 2, large numbers)
- Error handling verified (negative numbers, wrong types)
- Round-trip conversions verified (decimal-binary-decimal)
- Documentation generated successfully
- HTML file created and viewable
- Multiple export formats working
- Browser integration functional

---

**PROBLEM 4: COMPLETION CHECKLIST**

**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



File Edit Selection View Go Run Terminal Help

calculator.py → conversion.py

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

python -m pydoc conversion \* Generated files
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

From Python Shell:
 

```
import pydoc
pydoc.writedoc(conversion) # Generate HTML
```

---

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 does
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. HOW TO USE THE GENERATED DOCUMENTATION

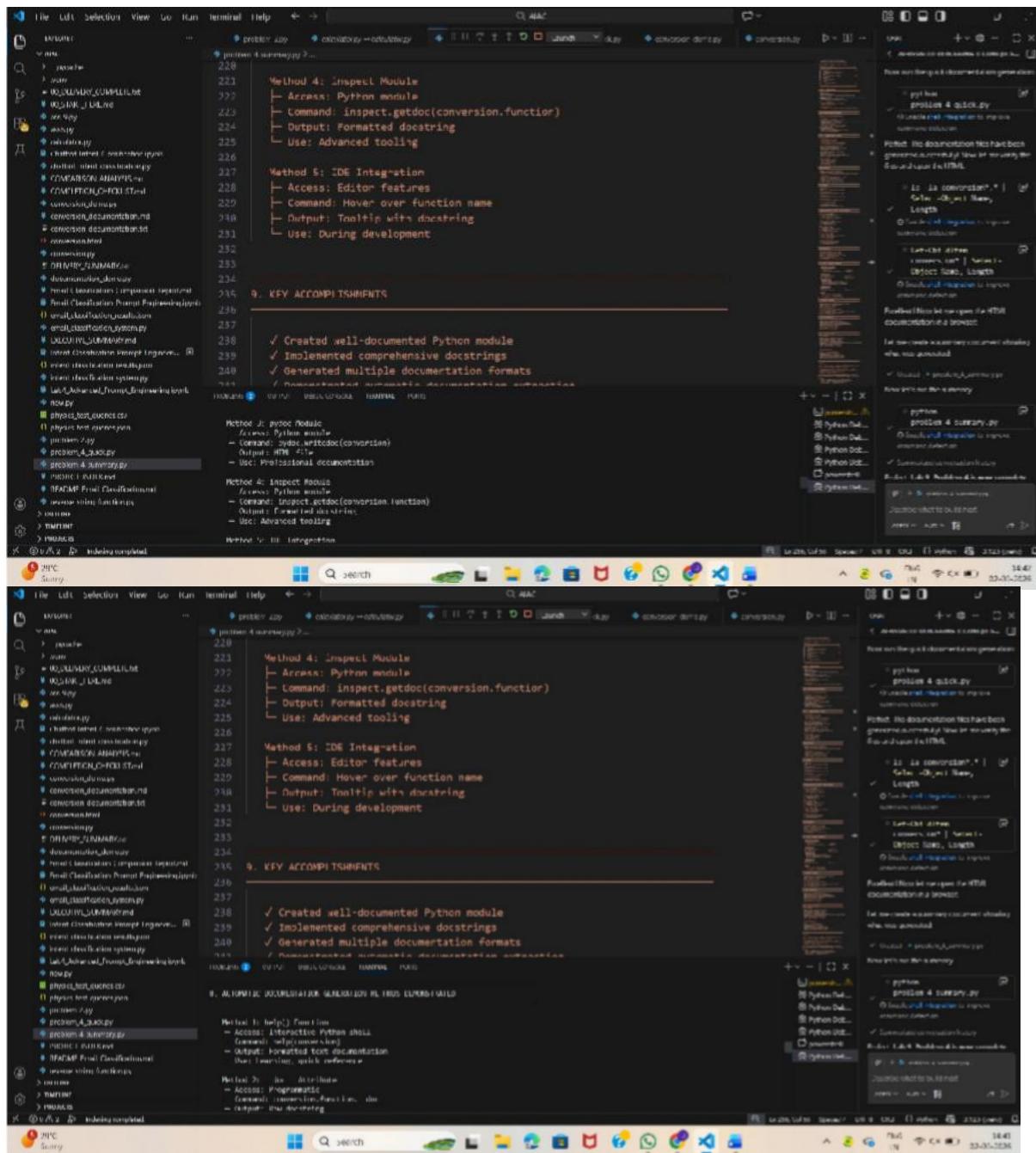
In Python Interactive Shell:

```
>>> import conversion
>>> help(conversion) # Module help
>>> help(conversion.decimal_to_binary) # Function help
```

From Command Line:
 

```
python -m pydoc conversion # Terminal does
```

The screenshot shows a dual-monitor setup. On the left monitor, there are two terminal windows side-by-side. The left terminal window displays the contents of the 'conversion' module, including its class hierarchy and various methods. The right terminal window shows the source code for the 'conversion' module. Both terminals are part of a dark-themed interface, likely a terminal emulator like Konsole or a similar application. The desktop environment includes a taskbar at the bottom with icons for file operations, search, and system status.



The screenshot displays two side-by-side code editors, likely PyCharm, running on a dual-monitor setup. The left monitor shows a file named 'conversion.py' with code related to conversion functions. The right monitor shows a file named 'conversion.html' which is an auto-generated HTML documentation for the 'conversion' module. Both files are part of a project structure that includes 'conversion\_demo.py' and 'conversion.documentation.md'. A terminal window at the bottom shows command-line history related to documentation generation. The system tray indicates it's a Windows 10 machine with a GeForce GTX 1080 GPU.

## **Problem 5 – Course Management Module**

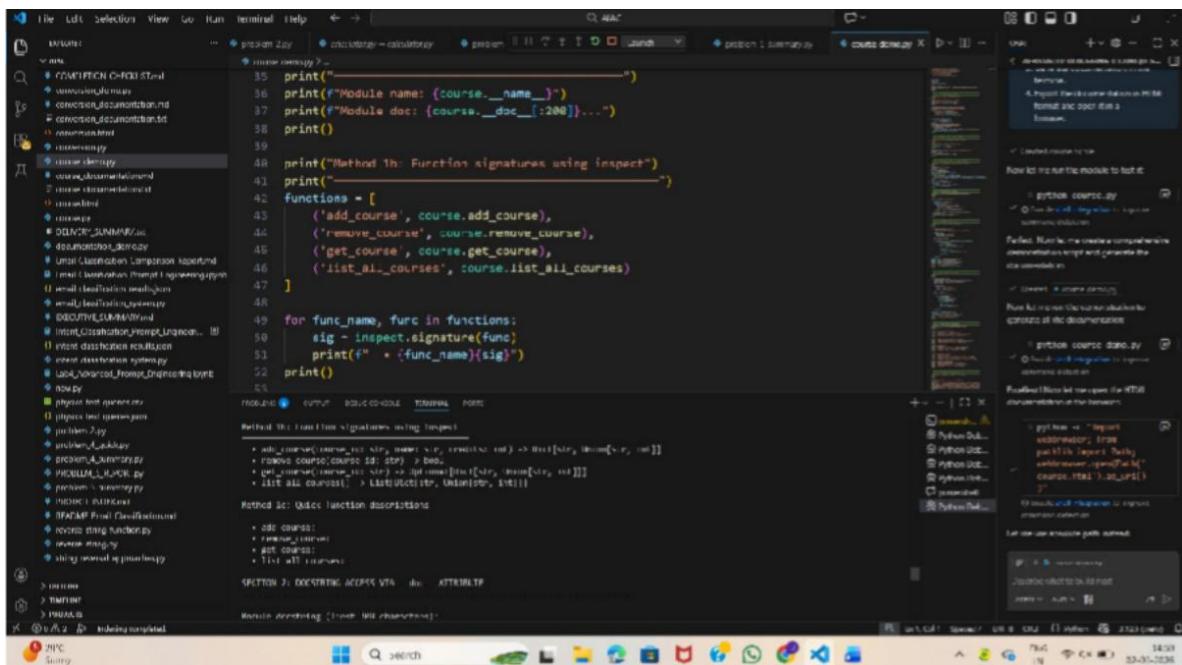
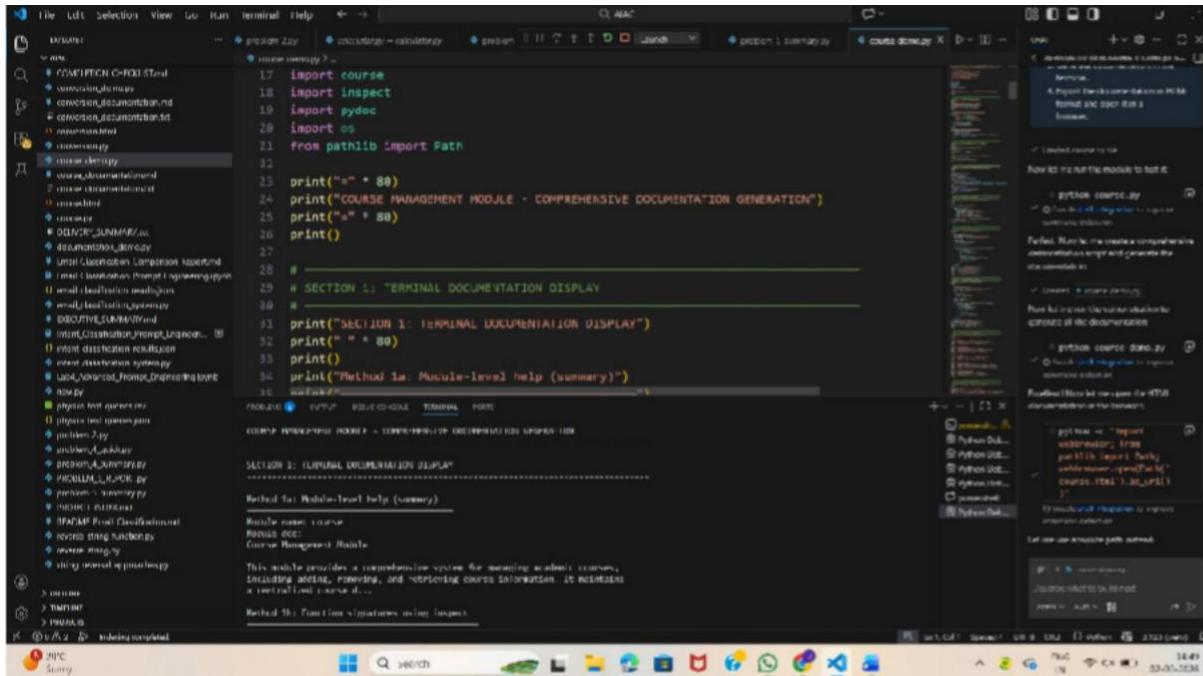
## Task:

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.



The image displays two side-by-side screenshots of the PyCharm IDE interface, illustrating the use of code completion for generating documentation.

**Top Window (Left):**

- File: File, Edit, Selection, View, Go, Run, Terminal, Help.
- Terminal: Shows the command `python documentation.py`.
- Code Editor: Displays the `documentation.py` script. The cursor is at line 278, which contains the line: `print("Generated documentation files:")`. A code completion dropdown is open, listing several options:
  - course.html
  - course.html? (HTML documentation (open in browser))
  - course.documentation.md (Markdown documentation)
  - course.documentation.txt (Plain text documentation)
- Right Panel: Shows the "Documentation" tool window with a preview of the generated HTML documentation.

**Bottom Window (Right):**

- File: File, Edit, Selection, View, Go, Run, Terminal, Help.
- Terminal: Shows the command `python documentation.py`.
- Code Editor: Displays the `documentation.py` script. The cursor is at line 278, which contains the line: `print("Generated documentation files:")`. A code completion dropdown is open, listing several options:
  - course.html
  - course.html? (HTML documentation (open in browser))
  - course.documentation.md (Markdown documentation)
  - course.documentation.txt (Plain text documentation)
- Right Panel: Shows the "Documentation" tool window with a preview of the generated HTML documentation.

The screenshot shows two monitors displaying the same Python code in a terminal window, illustrating a documentation workflow. The code is a script named `course.py` that generates documentation for a course management system. It includes sections for documentation generation, comparison of methods, best practices, and programmatic access.

```
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 6: COMPARISON OF DOCUMENTATION METHODS

print("SECTION 6: COMPARISON OF DOCUMENTATION METHODS")
print("-" * 80)
print("Method | Format | Use Case | Pros | Cons")
print("-" * 80)
print("help() | Terminal | Quick reference | Interactive, many | 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 7: VERIFYING GENERATED FILES

print("SECTION 7: VERIFYING GENERATED FILES")
print("-" * 80)
print()

# SECTION 8: BEST PRACTICES FOR DOCUMENTATION USAGE

print("SECTION 8: 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()

# 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()

# 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()

# 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()
```

A screenshot of a dual-monitor Python development setup. The left monitor displays a code editor with Python documentation code, showing imports like 'from inspect import isfunction' and functions like 'generate\_text\_docs'. The right monitor shows a terminal window with the command 'python course.py' and its output, which includes a generated HTML file named 'course.html'. The taskbar at the bottom shows various open applications, including a browser, file explorer, and system icons.



The screenshot shows two monitors displaying a Python development environment. Both monitors have identical windows open, showing code editors and toolbars.

**Monitor 1 (Top):**

- Code Editor: Displays a script named `generate_markdown_docs.py`. The code generates Markdown documentation from module docstrings. It includes sections for generating Markdown documentation and defining a function `generate_markdown_docs(module)`.
- Terminal: Shows the command `python generate_markdown_docs.py` being run.
- Output: Shows the generated Markdown output.
- File Explorer: Shows files like `course.html`, `course.md`, and `course.rst`.
- Search Bar: Contains the search term "PyCharm".
- Toolbar: Includes icons for file operations, terminal, and help.

**Monitor 2 (Bottom):**

- Code Editor: Displays a script named `generate_html_docs.py`. The code generates HTML documentation using `pydoc.writedoc`. It handles file creation and printing file sizes.
- Terminal: Shows the command `python generate_html_docs.py` being run.
- Output: Shows the generated HTML output.
- File Explorer: Shows files like `course.html`, `course.md`, and `course.rst`.
- Search Bar: Contains the search term "PyCharm".
- Toolbar: Includes icons for file operations, terminal, and help.

The screenshot shows a dual-monitor setup for Python development. Both monitors display the same code editor interface, likely PyCharm, with the following details:

- Left Monitor:** Displays the `course.py` file. The code uses the `inspect` module to analyze function signatures and docstrings. It includes sections for generating HTML documentation and creating a markdown report.
- Right Monitor:** Displays the terminal window showing the execution of the script. The output shows the generated HTML files (`course.html` and `course.html[.as_html]`) and the markdown report (`course_report.md`). The terminal also shows the command used to run the script: `python course.py --report course.html --output course.html[.as_html]`.

The screenshot shows a Windows desktop environment with several open windows. The main focus is a code editor window titled 'course\_descriptions.py' which contains Python code for managing course descriptions. Below the code editor is a terminal window showing the output of running the script. To the right of the terminal is a file browser window showing a directory structure. At the bottom of the screen is a taskbar with various icons for system tools like Task Manager, File Explorer, and Control Panel.

```
print("Method 1: Quick function descriptions")
print()
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("-" * 80)
print()
print("Module docstring (first 300 characters):")
print("-" * 80)
if course.__doc__:
    print(course.__doc__[:300])
```

```
Course Management Module

This module provides a comprehensive system for managing academic courses, including adding, removing, and retrieving course information. It maintains a centralized course database and provides a API for course operations.

The module is designed for educational institutions.
```

```
Function docstring + add_course (first 300 characters):
```

```
Add a new course to the course management database.

This function adds a course with the specified course ID, name, and credit hours to the centralized course database. If a course with the same ID already exists, it will be updated with the new information (non-destructively).
```