

**Name : G.Likhitha**

**Ht No: 2303A52487**

**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.

The screenshot shows the PyCharm IDE interface with the following details:

- File Path:** `find_max_simple.py`
- Code Editor Content:**

```
576 print("Documentation accessed via help():")
577 print("+" * 80)
578 help(Find_max_simple_docstring)
579
580 print("\nADVANTAGES:")
581 print(" ✓ Quick and concise")
582 print(" ✓ Easy to write")
583 print(" ✓ Suitable for simple functions")
584 print(" ✓ Less maintenance overhead")
585
586 print("\nDISADVANTAGES:")
587 print(" X Lacks detail about parameters and return values")
588 print(" X No examples for users")
589 print(" X No error documentation")
590 print(" X Insufficient for library documentation")
591 print(" X IDE tooltips show minimal information")
592
593 print("\nUSE CASES:")
594 print(" + Simple utility functions")
595
596 # The first operand (that or self),
597 # the second operand (float or int).
598 # operation A string representing the operation (+, -, *, /).
599
600 Returns:
601     The result of the operation as a float, or None if division by zero.
602 Raises:
603     ValueError: If operation is not one of "+", "-", "*", "/".
604     TypeError: If x or y are not numeric types.
605 Examples:
606     >>> calculate_with_good_style(10, 5, "+")
607     15.0
608     >>> calculate_with_good_style(10, 5, "/")
609     None
```
- Right Panel - Critical Analysis:**
  - Look this code up and bookmark.
  - See insight from the NCFI community.
  - Available - code available online.
- Right Panel - Recommendation for Mathematical Utilities Library:**
  - BEST APPROACH: Google-style Docstrings
  - I Before Comments
- Justification:**
  - Used only in professional code generation environments.
  - Common algorithmic code.
  - Implementation details are not complex enough (O(1), O(N)).
  - Most document numerical stability and IEEE 754 behavior.
  - Comprehensive, using symbolic use.
  - Comprehensive, generic coverage.
- Optimal Example: find\_max\_simple()**

Mathematical utilities provide mathematical library documentation with:

  - In-depth longer explanations
  - Issue-prone consistency warnings
  - IEEE 754 floating point consistency
  - Edge cases (NaN, infinity)
  - Robotic examples
  - Related function references
- Bottom Status Bar:** The file is ready to run and documents all imports with associated docstrings.

The screenshot shows the PyCharm IDE interface with the following details:

- File Path:** `find_max_in_line_comments.py`
- Code Editor Content:**

```
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 # =====#
601 # APPROACH 2: INLINE COMMENTS
602 #
603
604 print("+" * 80)
605 print("APPROACH 2: INLINE COMMENTS")
606 print("+" * 80 + "\n")
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")
```
- Right Panel - Critical Analysis:**
  - Look this code up and bookmark.
  - See insight from the NCFI community.
  - Available - code available online.
- Right Panel - Recommendation for Mathematical Utilities Library:**
  - BEST APPROACH: Google-style Docstrings
  - I Before Comments
- Justification:**
  - Used only in professional code generation environments.
  - Common algorithmic code.
  - Implementation details are not complex enough (O(1), O(N)).
  - Most document numerical stability and IEEE 754 behavior.
  - Comprehensive, using symbolic use.
  - Comprehensive, generic coverage.
- Optimal Example: find\_max\_in\_line\_comments()**

Mathematical utilities provide mathematical library documentation with:

  - In-depth longer explanations
  - Issue-prone consistency warnings
  - IEEE 754 floating point consistency
  - Edge cases (NaN, infinity)
  - Robotic examples
  - Related function references
- Bottom Status Bar:** The file is ready to run and documents all imports with associated docstrings.



calculator.py

```
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  
Help on function calculate_with_google_style or module __main__:  
calculate_with_google_style(x: float, y: float, operation: str) -> float  
    Performs a basic arithmetic operation on the numbers.  
  
    This function supports addition, subtraction, multiplication, and division.  
    Division by zero returns None to indicate an error condition.  
  
Args:  
    x: The first operand (float or int).  
    y: The second operand (float or int).  
    operation: A string representing the operation ('+', '-', '*', '/').  
  
Returns:  
    The result of the operation as a float, or None if division by zero.  
None
```

Critical Analysis

- Local style is consistent and well-chosen.
- See [Insight: There are PEP 257 maturity levels](#) → [Code style](#)

Recommendations for Mathematical Utilities Library

BEST APPROACH: Google-style Docstrings

In-line Comments

Justification:

- Used often in professional, collaborative environments.
- Comments are brief and implemented directly where complexity analysis (CIA) is high.
- Most documentational details are included in the code (IDE).
- PEP 257 follows.
- Implementation details are included in the code.

Optimal Example: `calculator.py`

Documentation for mathematical utilities library.

- Implements integer division.
- Includes comprehensive examples.
- PEP 257 following point-by-point.
- Full documentation included.
- Includes examples with associated code.

File ready to run and demonstrates all concepts with associated code.

29°C

Search

3439

The screenshot shows a Jupyter Notebook interface with several tabs at the top: File, Edit, Selection, View, Go, Run, terminal, Help, and a search bar. The main area contains Python code for numerical stability analysis:

```
1119 print(" - Numerical Stability Considerations")
1120 print(" - IEEE 754 Floating-Point Behavior")
1121 print(" - Edge Cases (infinity, NaN, empty)")
1122 print(" - Realistic Mathematical Examples")
1123 print(" - References to Related Functions")
1124 print(" - Performance Characteristics")
1125 print(" - Comparison with Alternative Implementations")
1126
1127 # *****
1128 # FINAL SUMMARY
1129 #
1130
1131 print("\n" + "=" * 80)
1132 print("FINAL SUMMARY AND RECOMMENDATIONS")
1133 print("-" * 80)
1134
1135 SUMMARY = """
1136 WHEN TO USE EACH STYLE:
1137
1138 1. Simple Duetstring (1 line):

```

Below the code, there is a help block for the `calculate_math_style()` function:

```
Help on function calculate_math_style in module __main__:
calculate_math_style(x, y) => string_operations (str) <- options[final]
    Performs a basic arithmetic operation on the numbers.

    This function supports addition, subtraction, multiplication, and division.
    Division by zero returns None to indicate an error condition.

    Arg:
        x: The first operand (float or int).
        y: The second operand (float or int).
    operation: A string representing the operation ('+', '-', '*', '/').

    Returns:
        The result of the operation as a float, or None if division by zero.

    See also:

```

The status bar at the bottom indicates "In [12] Out [13]" and the date "2023-01-25 14:53:28".

The screenshot shows a Jupyter Notebook interface with several cells of Python code. The code defines a function `find_max_optimized` that finds the maximum value in a list. It includes comments explaining different approaches like using `min()`, `mean()`, `max()`, and `numpy.max()`. The notebook also contains sections for 'Optimal Examples' and 'Optimal Solution' with detailed annotations.

```
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 Numpy Library Documentation:")
print("*" * 88)
help(#Ind_max_optimized)

print("\nKEY ADDITIONS FOR MATHEMATICAL FUNCTIONS:")
print(" + Time/Space Complexity Analysis")
print(" + Numerical Stability Considerations")
print(" + IEEE /34 Floating-Point Behavior")

# Help for the function calculate_with_numpy_style or module _math_
calculate_with_numpy = np.add(x, y, trim=operator.add) -> np.core._add
    Performs a basic arithmetic operation on the numbers.

    This function supports addition, subtraction, multiplication, and division.
    Division by zero returns None to indicate an error condition.

    Args:
        x: the first operand (float or int).
        y: the second operand (float or int).
        operator: A string representing the operation ('+', '+', '*', '/').
    Returns:
        The result of the operation as a float, or None if division by zero.

    Note:
```

The screenshot shows the PyCharm IDE interface with a Python file open. The code defines a function `find_max_optimized` that finds the maximum value in a list of floats. The code editor highlights several parts of the code with blue and red annotations, indicating potential issues or improvements. A vertical bar on the right side of the editor pane shows the current line of code being analyzed. To the right of the editor, there is a detailed 'Critical Analysis' report with sections for 'Notes', 'Examples', and 'Details'. The 'Notes' section lists several points about the use of `max()`, IEEE 754 floating-point comparisons, and NaN values. The 'Examples' section shows how the function is used with integers and floating-point numbers. The 'Details' section provides a breakdown of the code's complexity and performance characteristics. Below the analysis, there are sections for 'Recommendation for Mathematical Utilities Library', 'Best Approach: Google-style Docstrings', and 'Justification'. At the bottom of the analysis pane, there is a 'Run' button and a note about running the code. The status bar at the bottom of the screen shows the file path as 'C:\Users\...\.pycharm\myproject\src\math\math.py' and the current line as '1336'. The bottom right corner of the window has a timestamp '23.05.2019 14:43:39'.

The screenshot shows a Jupyter Notebook interface with several tabs open. The main code cell contains the following Python function:

```
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.
```

Below the code cell, there is a detailed docstring for the calculate\_math\_operation function:

```
calculate_math_operation(x: float, y: float, operation: str) -> Optional[Float]
    Performs a basic arithmetic operation on the numbers.

    This function supports addition, subtraction, multiplication, and division.
    Division by zero returns None to indicate an error condition.

    Args:
        x: The first operand (float or int).
        y: The second operand (float or int).
        operation: A string representing the operation ('+', '-', '*', '/').

    Returns:
        The result of the operation as a float, or None if division by zero.
```





File Edit Selection View Go Run Terminal Help

Critical Analysis

- Lack of style consistency and standardization
- Get Insight Themes are NFTs mostly include a code and all files

Recommendation for Mathematical Utilities Library

BEST APPROACH: Google-Style Documentation

Justification:

1. Users rely on professional, industry-standard documentation
2. Simplify algorithmic work
3. Standardized code makes it easier to understand and maintain
4. Multi-document automated consistency and PEP 8 validation
5. Consistent naming, using logical file organization
6. Code readability and maintainability

Optional Example: README.md

Documentation process for mathematical library documentation

- Google-Style: Most popular in Python
- Readme: Standard for GitHub repos
- Sphinx: StructuredText for large projects
- PEP 287: Standard Python style

7. TOOLS FOR DOCUMENTATION:

- Sphinx: Generates HTML/EPUB from its settings
- pygments: Helps to document automatically

File Edit Selection View Go Run Terminal Help

Critical Analysis

- Lack of style consistency and standardization
- Get Insight Themes are NFTs mostly include a code and all files

Recommendation for Mathematical Utilities Library

BEST APPROACH: Google-Style Documentation

Justification:

1. Users rely on professional, industry-standard documentation
2. Simplify algorithmic work
3. Standardized code makes it easier to understand and maintain
4. Multi-document automated consistency and PEP 8 validation
5. Consistent naming, using logical file organization
6. Code readability and maintainability

Optional Example: README.md

Documentation process for mathematical library documentation

- Google-Style: Most popular in Python
- Readme: Standard for GitHub repos
- Sphinx: StructuredText for large projects
- PEP 287: Standard Python style

7. TOOLS FOR DOCUMENTATION:

- Sphinx: Generates HTML/EPUB from its settings
- pygments: Helps to document automatically



The screenshot shows a dual-monitor setup with both monitors displaying PyCharm interface. The left monitor is focused on the 'DataProcessor' project, specifically on the 'find\_max\_with\_inline\_comments' function. The right monitor is focused on the 'Unsorted List Classification' project, specifically on the 'prime\_sieve' function.

**Left Monitor (DataProcessor Project):**

- Code:**

```
def find_max_with_inline_comments(numbers: List[float]) -> float:  
    # Use Python's built-in max() function for efficiency  
    # This is optimized in C and handles all numeric types  
    max_value = max(numbers)  
  
    print("Code:")  
    print("")  
    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')  
  
        # Use Python's built-in max() function for efficiency  
        # This is optimized in C and handles all numeric types  
        max_value = max(numbers)
```
- Toolbars and Status:** Standard PyCharm toolbars and status bar.

**Right Monitor (Unsorted List Classification Project):**

- Code:**

```
# 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 HOW)")  
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 TOF tooltip")  
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")
```
- Toolbars and Status:** Standard PyCharm toolbars and status bar.

**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 "Key Definitions" containing five items: 1. Onboarding - Accurate & Clear, 2. Security Considerations, 3. Information Security - Project, 4. Application Security Best Practices, and 5. Performance - Increases its understanding of the code. The main pane displays a Python file with the following content:

```
class UnionFind:
    def __init__(self):
        self.parent = {i: i for i in range(1000)}
        self.size = {i: 1 for i in range(1000)}
        self.rank = {i: 0 for i in range(1000)}

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        x_root = self.find(x)
        y_root = self.find(y)

        if x_root == y_root:
            return

        if self.rank[x_root] < self.rank[y_root]:
            self.parent[x_root] = y_root
            self.size[y_root] += self.size[x_root]
            self.rank[y_root] += 1
        else:
            self.parent[y_root] = x_root
            self.size[x_root] += self.size[y_root]
            self.rank[x_root] += 1

    def connected(self, x, y):
        return self.find(x) == self.find(y)
```

Below the code, a note states: "3. GOOGLE-STYLE DOCSSTRING - Complete Guidance". The code editor interface includes tabs for "INPUT", "CONSOLE", "TERMINAL", and "FILE".

The screenshot shows a code editor with a sidebar titled "Key Definitions" containing five items: 1. Onboarding - Accurate & Clear, 2. Security Considerations, 3. Information Security - Project, 4. Application Security Best Practices, and 5. Performance - Increases its understanding of the code. The main pane displays a Python file with the following content:

```
class UnionFind:
    def __init__(self):
        self.parent = {i: i for i in range(1000)}
        self.size = {i: 1 for i in range(1000)}
        self.rank = {i: 0 for i in range(1000)}

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

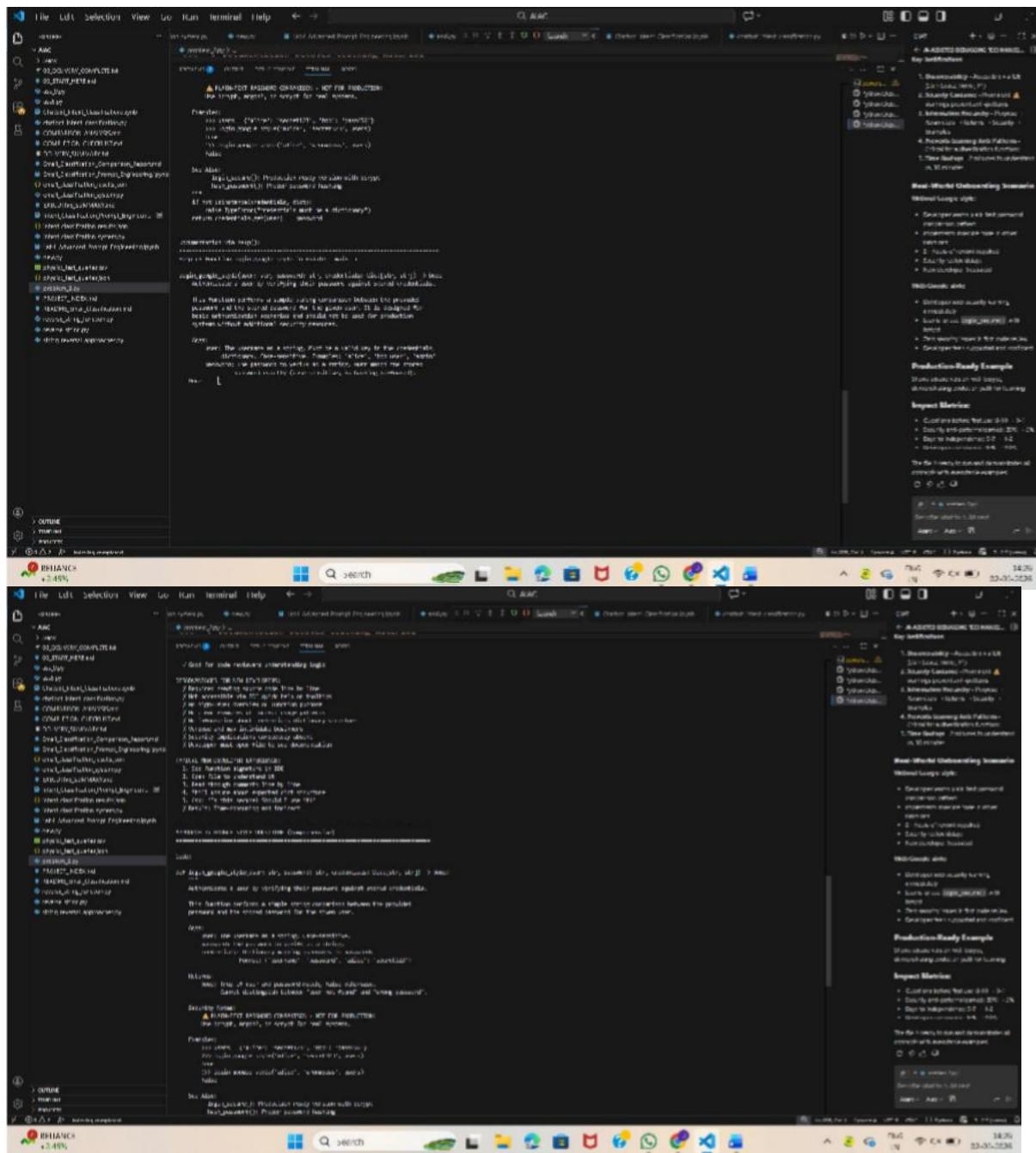
    def union(self, x, y):
        x_root = self.find(x)
        y_root = self.find(y)

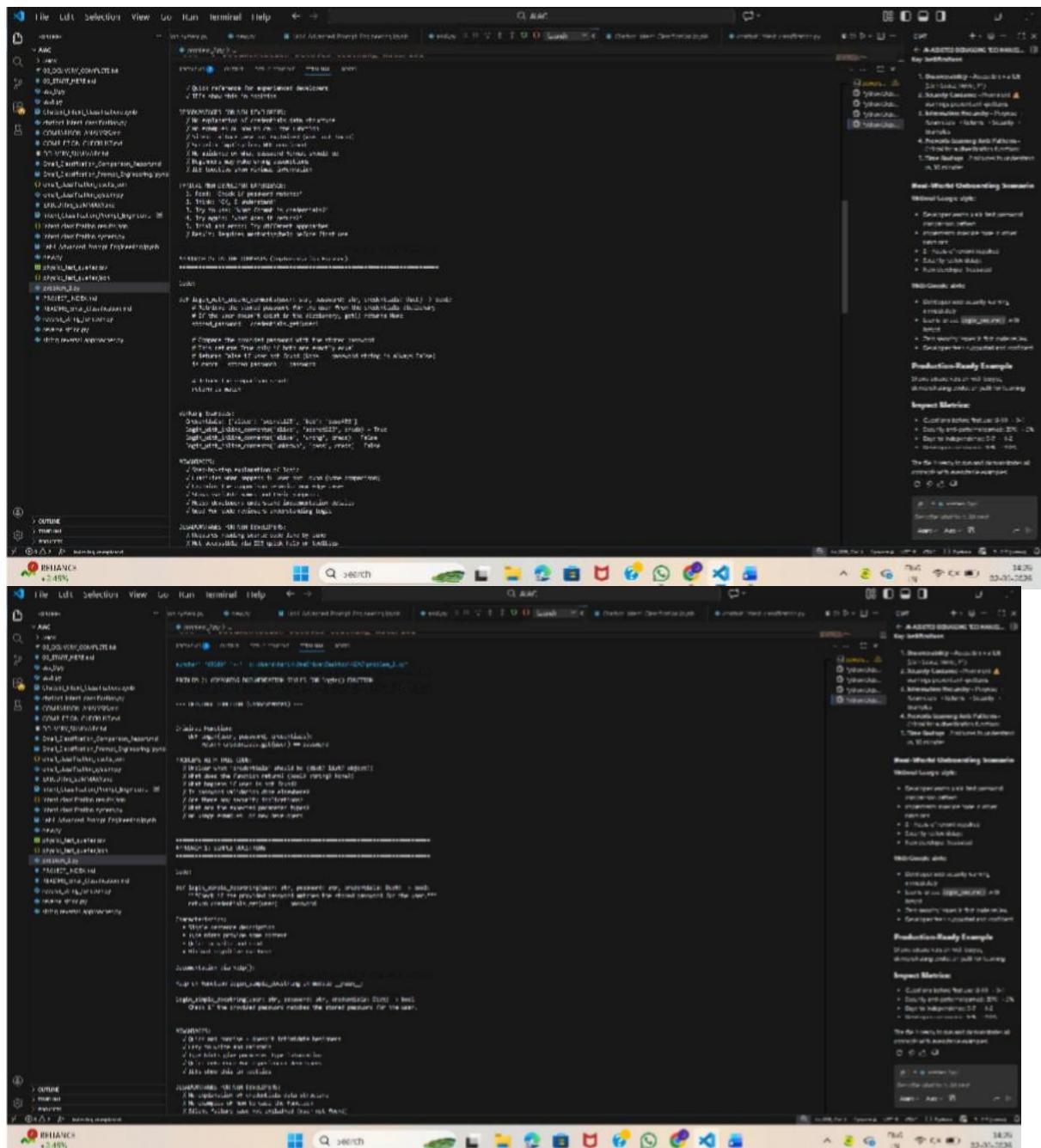
        if x_root == y_root:
            return

        if self.rank[x_root] < self.rank[y_root]:
            self.parent[x_root] = y_root
            self.size[y_root] += self.size[x_root]
            self.rank[y_root] += 1
        else:
            self.parent[y_root] = x_root
            self.size[x_root] += self.size[y_root]
            self.rank[x_root] += 1

    def connected(self, x, y):
        return self.find(x) == self.find(y)
```

Below the code, a note states: "4. PYTHON-STYLE DOCSSTRING - Minimal Guidance". The code editor interface includes tabs for "INPUT", "CONSOLE", "TERMINAL", and "FILE".

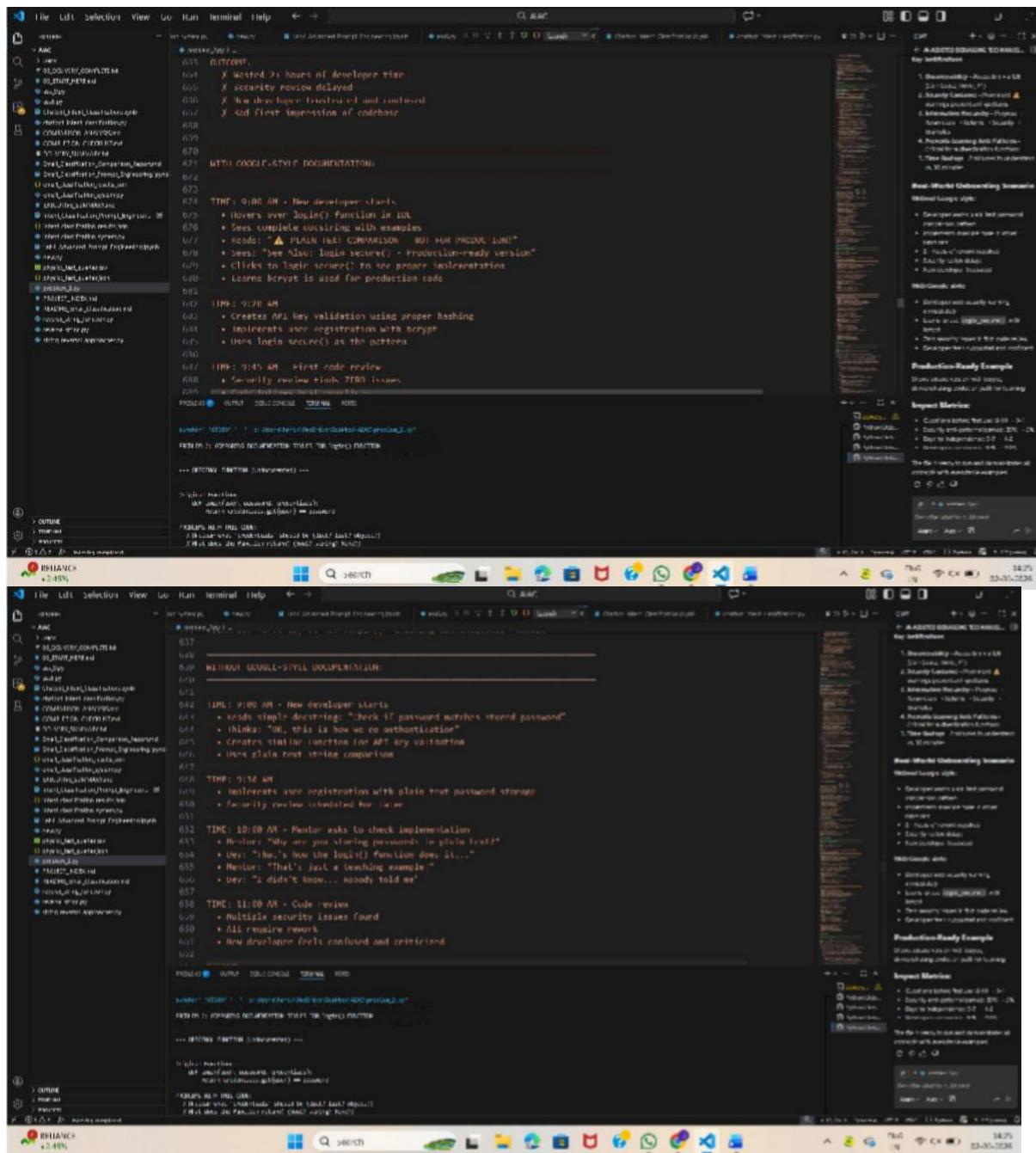








A screenshot of a Windows desktop showing a developer's integrated development environment (IDE). The top menu bar includes File, Edit, Selection, View, Go, Run, terminal, Help, and a search bar. The left sidebar shows a file tree with a .gitignore file open, containing various ignore patterns. The main code editor area displays a Java file with annotations and code snippets. A vertical margin on the right side of the code editor highlights specific lines with red and green markers, indicating code review comments. Below the code editor is a status bar showing 'PROJECT: OUTPUT: DEBUGGING: TERMINAL: READ'. The bottom status bar includes battery level (44%), signal strength, and system icons. On the far right, there is a floating 'Metrics' window displaying performance data such as CPU usage (~100%), memory (~12GB), and disk (~120GB). The taskbar at the bottom shows icons for File Explorer, Task View, Start, and several pinned applications.







The screenshot shows two instances of the PyCharm IDE running side-by-side. Both instances have the same code editor open, displaying a Python module named `calculator.py`. The code contains several functions with detailed docstrings, demonstrating automatic documentation generation. The PyCharm interface includes toolbars, menus, and a sidebar with various project-related tabs like 'File Structure', 'Tool Window', and 'Run'.

```

# File: calculator.py
# This module provides basic arithmetic operations.
# It includes functions for addition, subtraction, multiplication, division, and modulus.

def add(x, y):
    """
    Add two numbers x and y.

    :param x: First number
    :param y: Second number
    :return: Sum of x and y
    """
    return x + y

def subtract(x, y):
    """
    Subtract y from x.

    :param x: Minuend
    :param y: Subtrahend
    :return: Difference between x and y
    """
    return x - y

def multiply(x, y):
    """
    Multiply x by y.

    :param x: multiplicand
    :param y: multiplier
    :return: Product of x and y
    """
    return x * y

def divide(x, y):
    """
    Divide x by y.

    :param x: dividend
    :param y: divisor
    :return: Quotient of x divided by y
    """
    if y == 0:
        raise ValueError("Cannot divide by zero")
    return x / y

def modulus(x, y):
    """
    Calculate the modulus of x with respect to y.

    :param x: dividend
    :param y: divisor
    :return: Remainder of x divided by y
    """
    if y == 0:
        raise ValueError("Cannot calculate modulus by zero")
    return x % y

```

## Problem 3: Calculator (Automatic Documentation Generation)

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

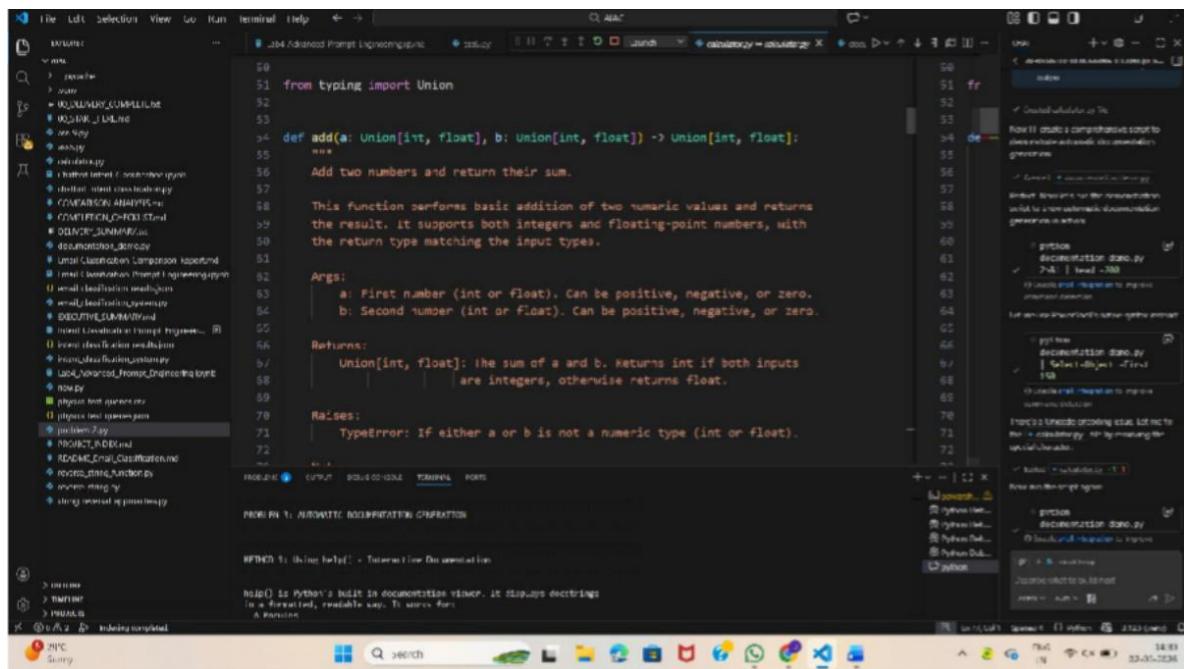
Instructions:

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: Union[int, float], b: Union[int, float]) -> Union[int, float]
    """
    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).
    """

```

A screenshot of a Windows desktop environment. In the center, a code editor window titled "C:\APC" displays a Python script. The script contains a function definition:

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

The code includes a "Notes:" section with the following comments:

- 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

Below the notes, there is a "Floating point precision may apply (e.g., 0.1 + 0.2 + 0.3)" message.

The "Examples:" section shows:

```
>>> add(5, 3)
8
```

and

```
>>> add(-5, 3)
-2
```

Under the examples, there is another note:

Adding floating point numbers:

```
>>> add(3.5, 2.3)
5.8
```

Finally, the code includes a note about mixed types:

Adding mixed types: returns float:

At the bottom of the code editor, there is a "PROBLEMS" tab.

Below the code editor, a "HELPDOC" tab is active, displaying the following text:

HELPDOC: Using help() - Interactive Documentation

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

In the bottom right corner of the screen, there is a "Windows Help" icon.





The screenshot shows a Windows desktop environment with several open windows. In the center, a PyCharm IDE window displays a Python script with syntax highlighting and code completion. Below it, a terminal window shows the command 'python -m pydoc -g' followed by the generated documentation for the 'help' module. A help command for 'help()' is also shown, listing various methods and their descriptions. The desktop taskbar at the bottom includes icons for File Explorer, Task View, and various system status indicators.

The screenshot displays a dual-monitor setup for Python development. Both monitors show the same interface, which includes:

- Left Monitor:** Shows a code editor with a file named `calculator.py`. The code implements basic arithmetic operations and handles division by zero. It also contains a `help()` docstring and a `__main__` block for testing.
- Right Monitor:** Shows a terminal window running the command `python calculator.py`, which outputs the generated documentation string.

Both monitors also display a sidebar with various Python-related icons and a bottom navigation bar with tabs like "PROBLEMS", "CURRENT", "HISTORY", "TERMINAL", and "PAGES". A status bar at the bottom right shows the date and time as 14:01 23-07-2024.

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):**

- Terminal 1:** Shows the following code in a cell:

```
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
```
- Terminal 2:** Shows the following code in a cell:

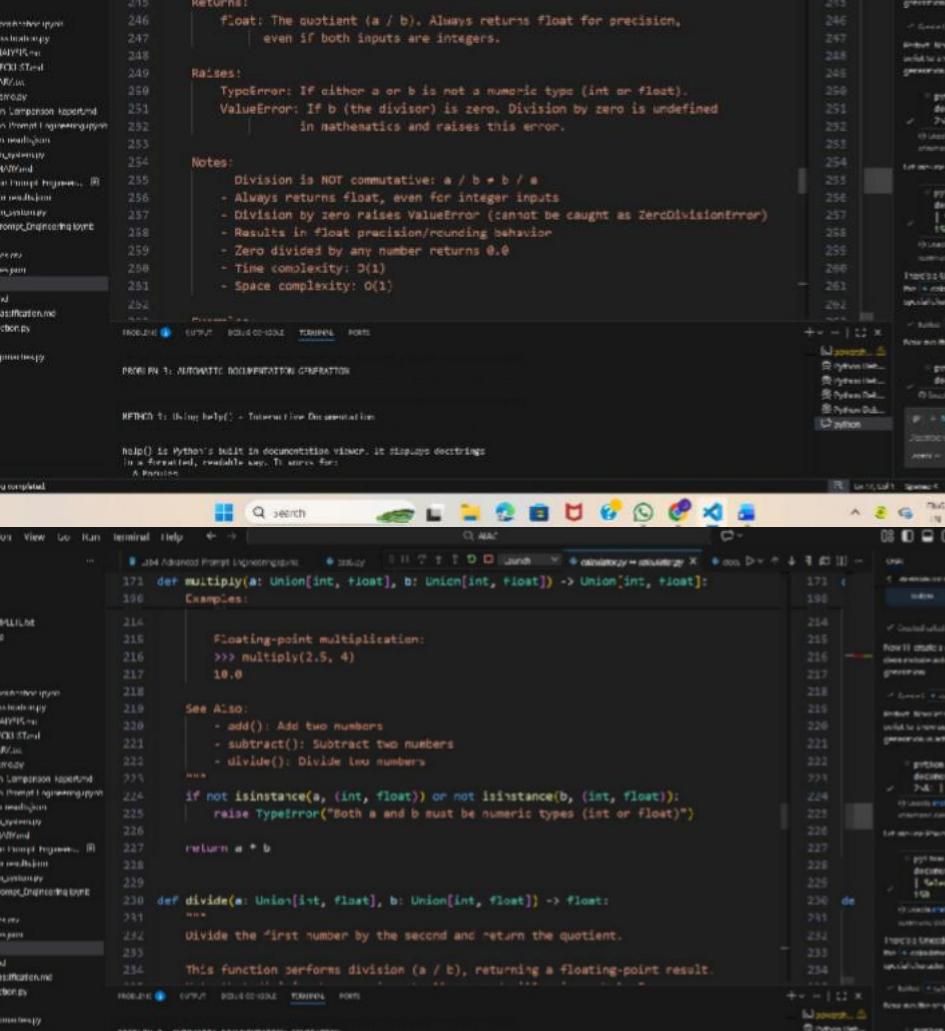
```
# DEMONSTRATION CODE (When run as main)  
# ======  
if __name__ == "__main__":  
    print("= " * 80)  
    print("CALCULATOR MODULE - DEMONSTRATION")  
    print("= " * 80)
```

**Monitor 2 (Right):**

- Terminal 1:** Shows the following code in a cell:

```
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
```
- Terminal 2:** Shows the following code in a cell:

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



```
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)

    Examples:
        >>> divide(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 1: AUTOMATIC DOCUMENTATION GENERATION

ANSWER 1: Using help() - Interactive Documentation

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

- A function

PROBLEM 2: Using help() - Interactive Documentation

ANSWER 2: Using help() - Interactive Documentation

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

- A function



```

File Edit Selection View Go Run Terminal Help < > C:\Python\operator.py 113 def subtract(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
    Notes:
        - Subtraction is not commutative: a - b != b - a (excepts when b=0)
        - Result type follows same rules as add()
        - Floating-point precision may apply
        - Time complexity: O(1)
        - Space complexity: O(1)

    Examples:
        Basic subtraction:
        >>> subtract(10, 3)
        7

        Subtraction resulting in negative:
        >>> subtract(3, 10)
        -7

        Subtracting negative numbers (adds):
        >>> subtract(5, -3)
        8

        Subtracting from zero:
        >>> subtract(0, 5)
        5

    PROBLEM 1: AUTOMATIC DOCUMENTATION CREATION

    PYTHON-1: Using help() + Interactive Documentation

    help() is Python's built-in documentation viewer. It displays docstrings
    in a formatted, readable way. It works for:
    A function

```

```

File Edit Selection View Go Run Terminal Help < > C:\Python\operator.py 54 def add(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
    Examples:
        Adding zero:
        >>> add(0, 5)
        5

    See Also:
        - subtract(): Subtract 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 subtract(a: Union[int, float], b: Union[int, float]) -> Union[int, float]:
        """
        Subtract the second number from the first and return the difference.

        This function performs basic subtraction, computing a - b. It supports
        both integers and floating-point numbers. The order of operands matters!
        """
        return a - b

    PROBLEM 1: AUTOMATIC DOCUMENTATION CREATION

    PYTHON-1: Using help() + Interactive Documentation

    help() is Python's built-in documentation viewer. It displays docstrings
    in a formatted, readable way. It works for:
    A function

```

## Problem 4: Conversion Utilities Module

Task:

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

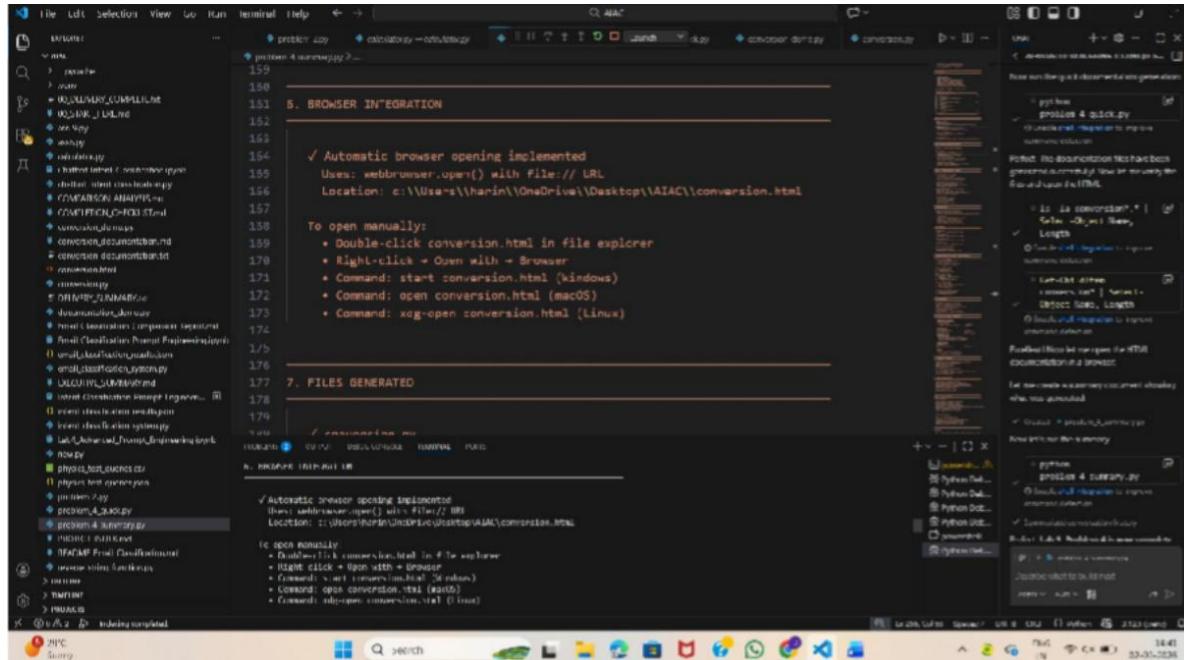
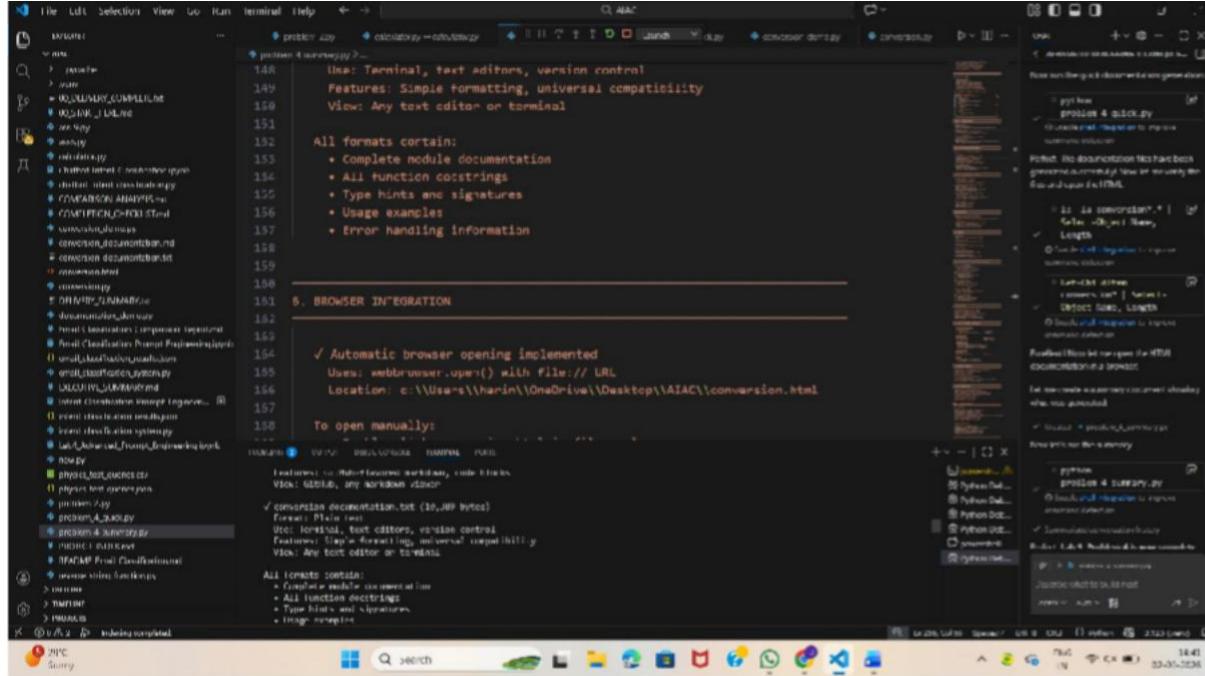
- o decimal\_to\_binary(n)
- o binary\_to\_decimal(b)
- o decimal\_to\_hexadecimal(n)

## 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.

## Browser





```
File Edit Selection View Go Run Terminal Help < > C:\Users\... problem_4_quick.py calculator>cd\output 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} {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 objects
- Multiple documentation delivery methods (Markdown, HTML, WebHelp, text)
- Professional HTML output suitable for websites
- Best practices in structuring artifacts
- Automatic documentation generation using Python's built-in tools

```
File Edit Selection View Go Run Terminal Help < > C:\Users\... problem_4_summary.py calculator>cd\output 353 354  print(summary) 355 356  # Show file listings 357  print("\n" + "-" * 80) 358  print("GENERATED FILES IN CURRENT DIRECTORY") 359  print("-" * 80) 360 361  print("\nconversion-related files:\n") 362 363  conversion_files = { 364      'conversion.py': 'Main module with converter functions', 365      'conversion.html': 'HTML documentation (open in browser)', 366      'conversion_documentation.md': 'Markdown documentation', 367      'conversion_documentation.txt': 'Plain text documentation', 368      'conversion_demo.py': 'Full demonstration script', 369      'problem_4_quick.py': 'Quick generation script', 370  } 371 372 373 374 375 376 377 378 379 380 381 382  for filename, description in conversion_files.items(): 383      if os.path.exists(filename): 384          print(f"\n {filename} {size}>8, {bytes} - {description}") 385 386 387 388 389 380
```

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

- Source code documentation that automatically generates all objects
- Multiple documentation delivery methods (Markdown, HTML, WebHelp, text)
- Professional HTML output suitable for websites
- Best practices in structuring artifacts
- Automatic documentation generation using Python's built-in tools

The screenshot shows a terminal window with a dark theme. The title bar reads "File Edit Selection View Go Run terminal Help". The main area contains several code snippets related to a "conversion" module. One snippet discusses error handling and type hints. Another snippet for "CONCLUSION:" states: "Problem 4 has been successfully completed. The conversion utilities module demonstrates professional-grade documentation practices." It lists several bullet points: "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 documentation writing", and "Automatic documentation extraction using Python's built-in tools". A third snippet for "CONCLUSION:" reiterates the successful completion and provides a summary of the module's features. At the bottom, there is an "Additional Achievements:" section with three checked items: "Generated Sphinx documentation", "Generated Travis CI documentation", and "Implemented error handling with meaningful messages". On the left, a sidebar shows the project structure with files like "conversion.py", "conversion.html", "conversion.md", and "conversion\_utils.py". On the right, a file browser shows various Python files and a "conversion" folder. The bottom of the screen shows a taskbar with icons for file operations like "New", "Open", "Save", and "Exit".

**File Edit Selection View Go Run Terminal Help**

**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

**File Edit Selection View Go Run Terminal Help**

**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

**File Edit Selection View Go Run Terminal Help**

**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 files created and viewable
- Multiple export formats working
- Browser integration functional

**File Edit Selection View Go Run Terminal Help**

**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 files created and viewable
- Multiple export formats working
- Browser integration functional

The image shows a dual-monitor system. The left monitor displays a code editor with the file 'problem 4 summary.py' open. The code defines a function 'generate\_html' that uses the 'pydoc' module to write a docstring to a file named 'conversion'. The right monitor displays the resulting HTML document, which has a header 'problem 4 summary.html' and contains two main sections: '11. BEST PRACTICES DEMONSTRATED' and '12. TESTING AND VERIFICATION'. Each section lists several best practices, such as comprehensive docstrings, Google-style format, type hints, and error documentation. A sidebar on both monitors lists other Python files like 'problem 4 quick.py', 'problem 4 summary\_by\_id.py', and 'problem 4 test\_summary.py'. At the bottom of each monitor, there is a terminal window showing the command 'python problem 4 summary.py' being run. The overall interface is a dark-themed IDE.

The screenshot illustrates a dual-monitor Linux desktop environment. The left monitor displays a terminal window with the following command:

```
python -m pydoc conversion
```

The right monitor displays a web browser showing the generated documentation. The top part of the documentation page contains the following text:

**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

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 Script:

```
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 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

---

**10. HOW TO USE THE GENERATED DOCUMENTATION**

**In Python Interactive Shell:**

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

**From Command Line:**

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

The terminal window displays the command:

```
python -m pydocx --output=calculator
```

The output shows the generation of documentation for the 'calculator' module, including sections for key accomplishments and how to use the generated documentation.

**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:

```
python -m pydocx --output=calculator
```

The terminal window also shows the command:

```
python -m pydocx --output=calculator
```

The output shows the generation of documentation for the 'calculator' module, including sections for key accomplishments and how to use the generated documentation.

**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

**9. KEY ACCOMPLISHMENTS**

- ✓ Created well-documented Python module
- ✓ Implemented comprehensive docstrings
- ✓ Generated multiple documentation formats



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

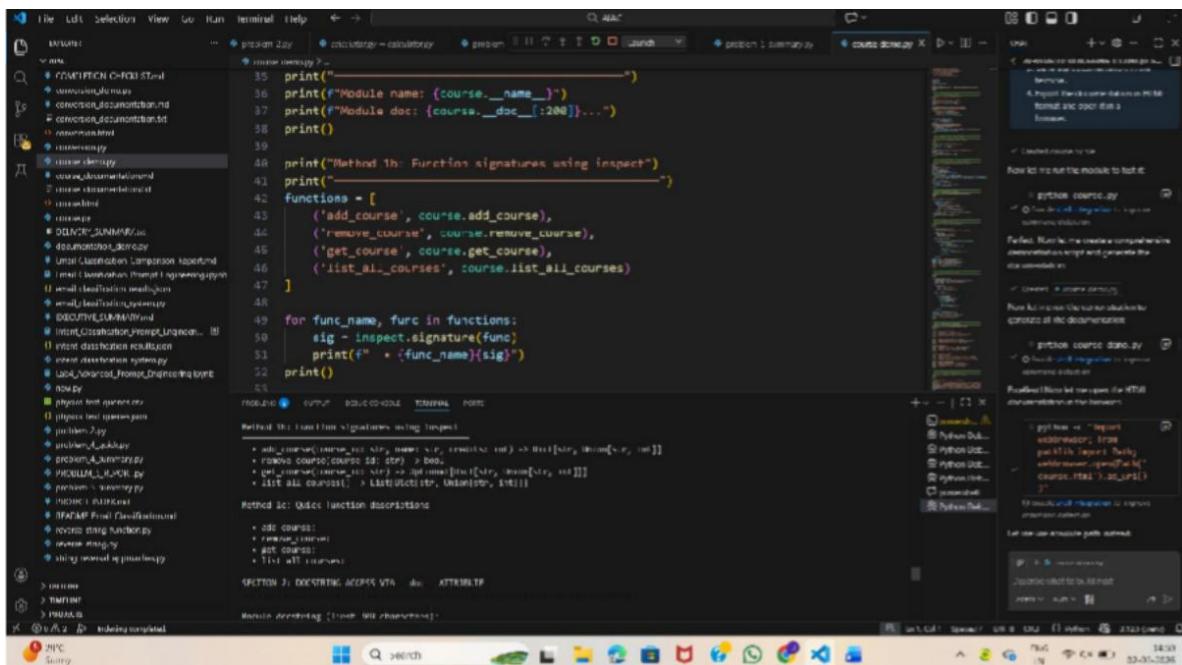
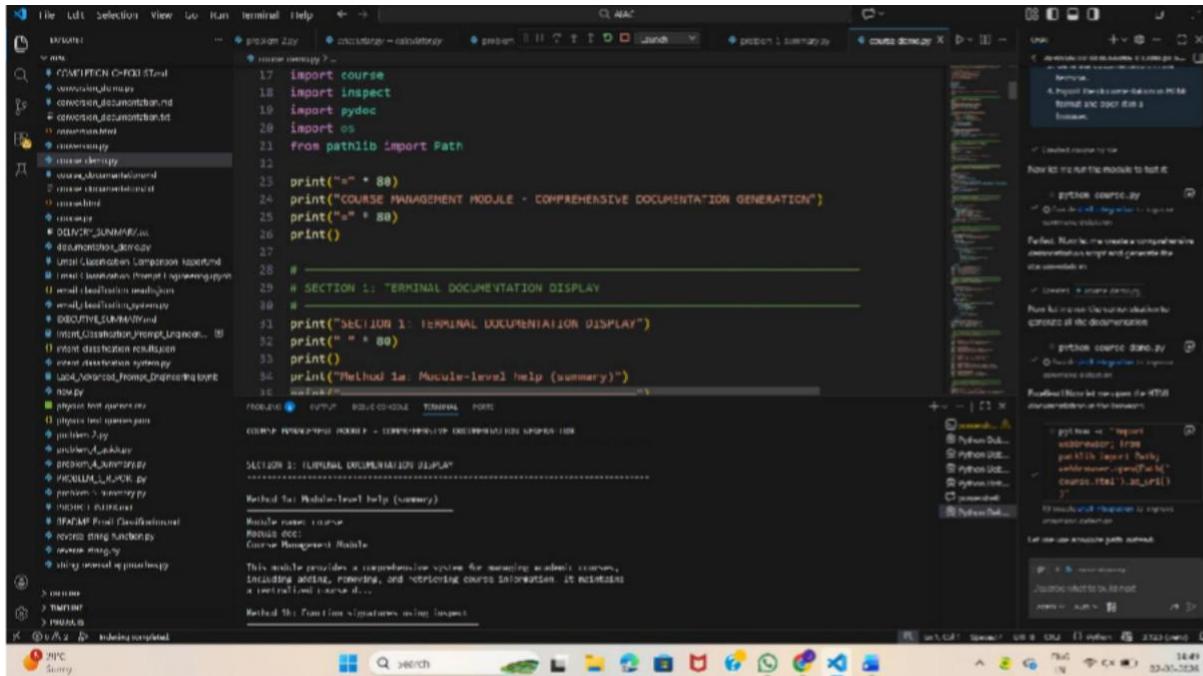
## Task:

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 Jupyter Notebook interfaces, both running on Windows 10, illustrating the process of generating documentation from Python code.

**Top Notebook:**

- Code:** The code in this notebook prints various types of documentation files (HTML, Markdown, plain text) and provides instructions for viewing them.
- Output:** The output shows the generated files: course.html, course.documentation.md, and course.documentation.txt.
- Context:** A tooltip on the right indicates that the code uses `__docstring__` to generate documentation.

**Bottom Notebook:**

- Code:** This notebook contains the same documentation generation logic as the top one, but it also includes a section for "SUMMARY AND NEXT STEPS".
- Output:** The output shows the generated files: course.html, course.documentation.md, and course.documentation.txt.
- Context:** A tooltip on the right indicates that the code uses `__docstring__` to generate documentation.

Both notebooks have a status bar at the bottom showing the date and time (2021-07-20 10:47).

```
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}: {description} ({size:,} bytes)")
    else:
        print(f"✗ {filename}: {description} (NOT FOUND)")

print()

# SECTION 10: SUMMARY AND NEXT STEPS
# -----
print("SECTION 10: SUMMARY AND NEXT STEPS")
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")
```

```
File Edit Selection View Go Run terminal Help ... process Lzy calculator - calculateby process 11 7 t search sounds process 1 summary course docstring X D+ D- terminal C:\Python\course\course_docstring> python course_docstring.py
214 #
215 # SECTION 7: BEST PRACTICES FOR DOCUMENTATION
216 #
217 print("SECTION 7: BEST PRACTICES FOR DOCUMENTATION USAGE")
218 print("-" * 80)
219 print()
220
221
222 print("1. INTERACTIVE PYTHON SHELL")
223 print("    | help(course) - Module documentation")
224 print("    | help(course.add_course) - Function documentation")
225 print("    | course.add_course.__doc__ - Raw docstring")
226 print()
227
228 print("2. IDE INTEGRATION")
229 print("    | Hover over: Function name for tooltip")
230 print("    | Ctrl+Shift+T (or Cmd+K,Cmd+I on Mac) For documentation")
231 print("    | Intellisense shows docstring in suggestions")
232 print()
233
234 print("3. COMMAND LINE")
235 print("    | python -m pydoc course - Terminal documentation")
236 print("    | python -m pydoc -w course - Generate HTML")
237 print("    | python -m pydoc -b course - Open in browser")
238 print()
239
240 print("4. PROGRAMMATIC ACCESS")
241 print("    | import inspect")
242 print("    | inspect.getdoc(course.add_course) - Formatted docstring")
243 print("    | inspect.signature(course.add_course) - Function signature")
244 print("    | inspect.getsource(course.add_course) - Function source code")
245 print()
246
247
```

The screenshot shows a Windows desktop environment with a Python development setup. The main window is a code editor displaying Python code related to generating documentation. The terminal window below shows the output of a build process, indicating successful compilation of documentation files. The sidebar on the right lists project files and provides various tools and configurations for the development environment.

The screenshot shows a terminal window with the following content:

```
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"
                text += "=" * 80 + "\n\n"

    print(text)
```

Below the code, there is a section titled "TEST STEPS:" with the following steps:

1. Open `course.html` in your web browser to view formatted docs.
2. Check `course_documentation.html` for GitHub-friendly format.
- .. Use `"python -m pydoc course"` for terminal based help.
- .. Report course module and use `help()` function for detailed live docs.

At the bottom of the terminal, there is a note: "DUPLICATION IS GRANTED BY COPY".

On the right side of the screen, there is a file browser window titled "File Explorer" showing a directory structure under "Python Doc". The "Python Doc" folder contains several subfolders and files, including "Python Data", "Python Doc", "Python Doc", "Python Doc", "Python Doc", and "Python Doc".

File Edit Selection View Go Run Terminal Help

Course Documentation Generator

```

155     md_path.write_text(markdown_content, encoding='utf-8')
156     print(f'Markdown documentation exported!')
157     print(f' File: course documentation.md')
158     print(f' Size: {md_path.stat().st_size}, bytes')
159     print()
160
161     # -----
162     # SECTION 6: PLAIN TEXT DOCUMENTATION EXPORT
163     #
164
165     print("SECTION 6: PLAIN TEXT DOCUMENTATION EXPORT")
166     print(" = 60")
167     print()
168
169     def generate_text_docs(module):
170         """Generate plain text documentation from module docstrings."""
171         text = f'{module.__name__.upper()} DOCUMENTATION\n'
172         for name, obj in inspect.getmembers(module, inspect.isfunction):
173             if not name.startswith('_'):
174                 md += f'{{{name}}}\n'
175                 sig = inspect.signature(obj)
176                 md += f'{sig}\n{obj.__doc__}\n'
177
178                 if obj.__doc__:
179                     md += f'<del>First paragraph</del>\n'
180                     first_para = obj.__doc__.split('\n\n')[0]
181                     md += f'{first_para}\n\n'
182
183                     md += "----\n"
184
185         markdown_content = generate_markdown_docs(course)
186         md_path = Path('course_documentation.md')
187         md_path.write_text(markdown_content, encoding='utf-8')

```

SECTION 5: VERIFICATION OF GENERATED FILES

Generating documentation files:

- ✓ course.py
- ✓ course.html
- ✓ course\_documentation.html
- ✓ course.documentation.html
- ✓ course.documentation.txt

SECTION 6: SUMMARY AND RUNS FILES

✓ DOCUMENTATION

- Module documentation created with comprehensive docstrings
- Terminal documentation displayed (via help and --ix)
- Plain documentation generated and saved

File Edit Selection View Go Run Terminal Help

Course Documentation Generator

```

155     md_path.write_text(markdown_content, encoding='utf-8')
156     print(f'Markdown documentation exported!')
157     print(f' File: course documentation.md')
158     print(f' Size: {md_path.stat().st_size}, bytes')
159     print()
160
161     # -----
162     # SECTION 6: PLAIN TEXT DOCUMENTATION EXPORT
163     #
164
165     print("SECTION 6: PLAIN TEXT DOCUMENTATION EXPORT")
166     print(" = 60")
167     print()
168
169     def generate_markdown_docs(module):
170         """Generate plain text documentation from module docstrings."""
171         for name, obj in inspect.getmembers(module, inspect.isfunction):
172             if not name.startswith('_'):
173                 md += f'{{{name}}}\n'
174                 sig = inspect.signature(obj)
175                 md += f'{sig}\n{obj.__doc__}\n'
176
177                 if obj.__doc__:
178                     md += f'<del>First paragraph</del>\n'
179                     first_para = obj.__doc__.split('\n\n')[0]
180                     md += f'{first_para}\n\n'
181
182                     md += "----\n"
183
184         markdown_content = generate_markdown_docs(course)
185         md_path = Path('course_documentation.md')
186         md_path.write_text(markdown_content, encoding='utf-8')

```

SECTION 5: VERIFICATION OF GENERATED FILES

Generating documentation files:

- ✓ course.py
- ✓ course.html
- ✓ course\_documentation.html
- ✓ course.documentation.html
- ✓ course.documentation.txt

SECTION 6: SUMMARY AND RUNS FILES

✓ DOCUMENTATION

- Module documentation created with comprehensive docstrings
- Terminal documentation displayed (via help and --ix)
- Plain documentation generated and saved

The screenshot displays a dual-monitor setup for Python development. The left monitor shows a code editor with a script named `generate_docs.py` that generates Markdown documentation from module docstrings. The right monitor shows a terminal window running the command `python course.py` to generate HTML documentation using the `pydoc` module.

```
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"

    for name, obj in inspect.getmembers(module, inspect.isfunction):
        if inspect.isfunction(obj):
            md += f"\n### {name}\n{obj.__doc__}\n\n"

    return md

if __name__ == "__main__":
    generate_markdown_docs(course)
```

```
print("SECTION 4: HTML DOCUMENTATION GENERATION USING PYDOC")
print("-" * 80)
print()

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()}\n')
    else:
        print("X HTML file not found after generation attempt")
except Exception as e:
    print(f"X Error generating HTML: {e}")
```

The screenshot shows a Windows desktop environment with a Python development setup. The main window is a code editor displaying a script named `course.py`. The script includes functions for adding courses and printing course details. A terminal window below the editor shows command-line interactions. To the right, there's a sidebar with documentation for the `inspect` module and other Python tools. The taskbar at the bottom has icons for a browser, file explorer, and system tray.

The screenshot shows a Python IDE interface with several tabs open. The main code editor tab contains Python code for a 'course' module, specifically sections 1 and 2 of a docstring example. Below the code editor, the terminal window shows the command 'python course\_main.py'. The bottom status bar indicates the file is 'fully analyzed'.

```
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("----- * 88")
print("Module docstring (first 300 characters):")
print("----- * 88")
if course.__doc__:
```