Name:Gaddam Sai

Hall Ticket:2303A52255

Batch:43

Assginment:9.1

Task : 01

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

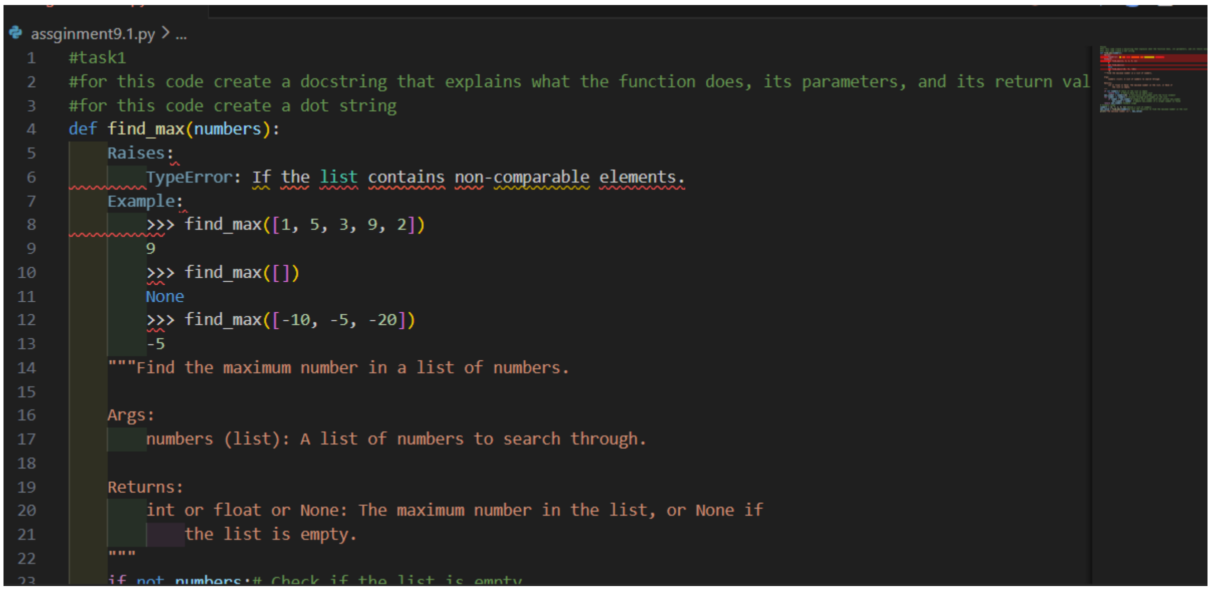
(c) Google-style documentation

• 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.

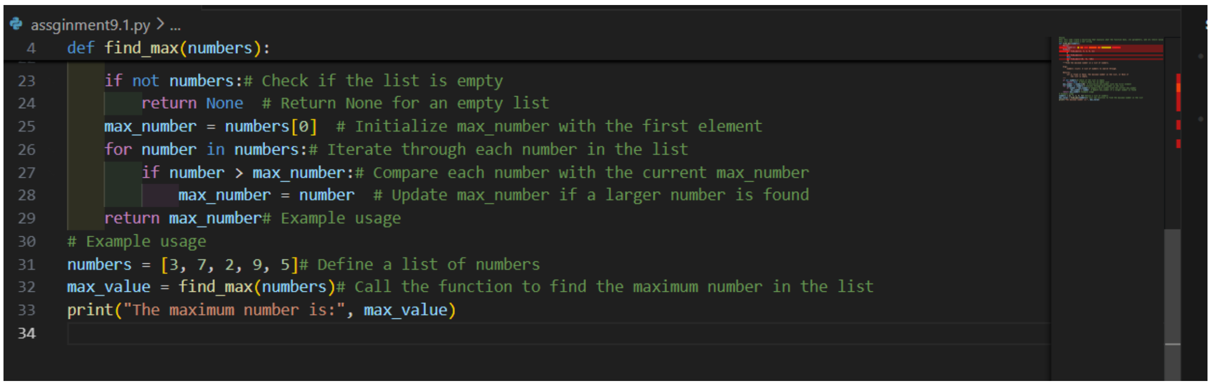
Prompt:for this code implement the documentations

Code:



assginment9.1.py > ...

```python
#task1
#for this code create a docstring that explains what the function does, its parameters, and its return val
#for this code create a dot string
def find_max(numbers):
    Raises:
        TypeError: If the list contains non-comparable elements.
    Example:
        >>> find_max([1, 5, 3, 9, 2])
        9
        >>> find_max([])
        None
        >>> find_max([-10, -5, -20])
        -5
    """Find the maximum number in a list of numbers.

    Args:
        numbers (list): A list of numbers to search through.

    Returns:
        int or float or None: The maximum number in the list, or None if
            the list is empty.
    """
    if not numbers:# Check if the list is empty
```



assginment9.1.py > ...

```python
def find_max(numbers):
    if not numbers:# Check if the list is empty
        return None  # Return None for an empty list
    max_number = numbers[0]  # Initialize max_number with the first element
    for number in numbers:# Iterate through each number in the list
        if number > max_number:# Compare each number with the current max_number
            max_number = number  # Update max_number if a larger number is found
    return max_number# Example usage
# Example usage
numbers = [3, 7, 2, 9, 5]# Define a list of numbers
max_value = find_max(numbers)# Call the function to find the maximum number in the list
print("The maximum number is:", max_value)

```

Observations:

Docstrings provide quick in-code documentation and are best for simple understanding, while inline comments explain logic step-by-step but can clutter the code.
Google-style documentation is more structured, readable, and suitable for large projects because it clearly defines parameters, returns, and behavior.
For a mathematical utilities library, Google-style documentation is most effective since it ensures consistency, clarity, and professional maintainability across many functions.

Task2:

Problem 2: Consider the following Python function:

def login(user, password, credentials):

return credentials.get(user) == password

Task:

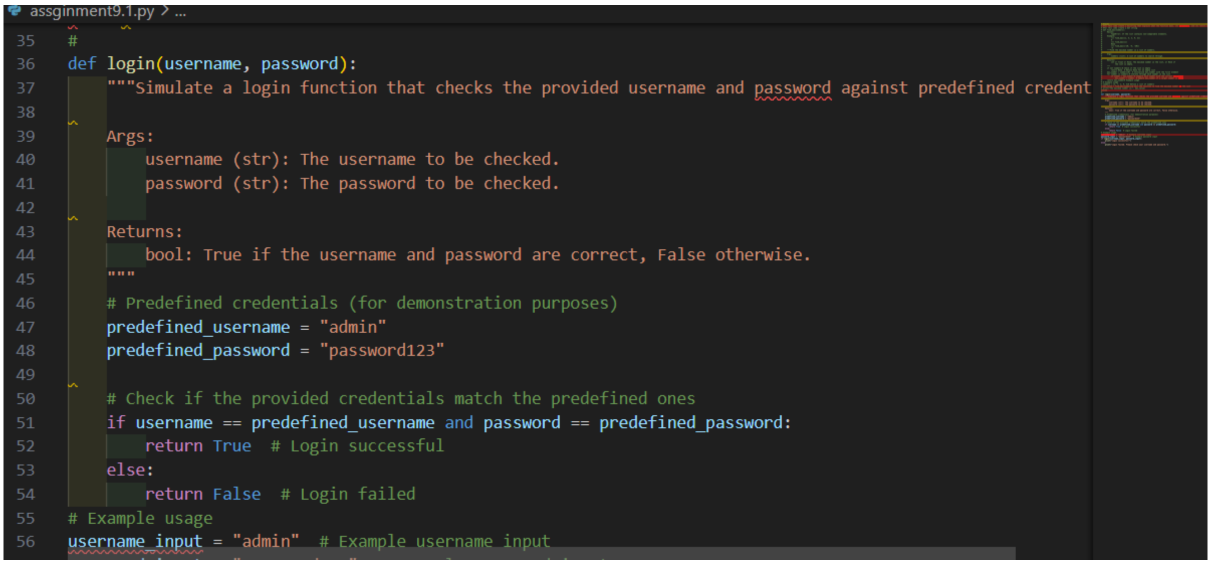
1. Write documentation in all three formats.

2. Critically compare the approaches.

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

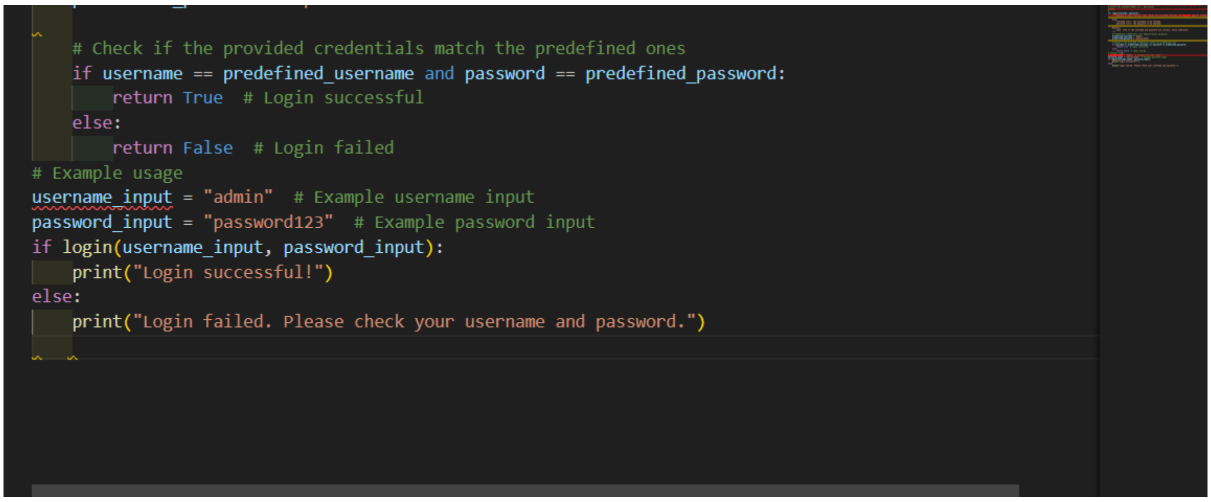
Prompt: for this code impment three documentations

Code:



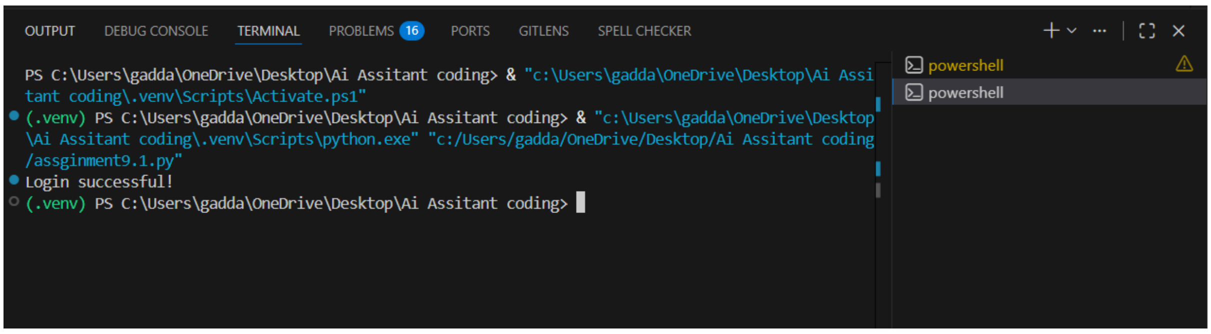
assginment9.1.py > ...

```python
#
def login(username, password):
    """Simulate a login function that checks the provided username and password against predefined credent

    Args:
        username (str): The username to be checked.
        password (str): The password to be checked.

    Returns:
        bool: True if the username and password are correct, False otherwise.
    """
    # Predefined credentials (for demonstration purposes)
    predefined_username = "admin"
    predefined_password = "password123"

    # Check if the provided credentials match the predefined ones
    if username == predefined_username and password == predefined_password:
        return True  # Login successful
    else:
        return False  # Login failed
# Example usage
username_input = "admin"  # Example username input
```



```python
    # Check if the provided credentials match the predefined ones
    if username == predefined_username and password == predefined_password:
        return True  # Login successful
    else:
        return False  # Login failed
# Example usage
username_input = "admin"  # Example username input
password_input = "password123"  # Example password input
if login(username_input, password_input):
    print("Login successful!")
else:
    print("Login failed. Please check your username and password.")
```

OUTPUT:



```
PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> & "c:\Users\gadda\OneDrive\Desktop\Ai Assi
tant coding\.venv\Scripts\Activate.ps1"
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> & "c:\Users\gadda\OneDrive\Desktop
\Ai Assitant coding\.venv\Scripts\python.exe" "c:/Users/gadda/OneDrive/Desktop/Ai Assitant coding
/assginment9.1.py"
Login successful!
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

Inline comments help beginners understand each step of the login logic but may make the code look crowded if overused. Docstrings summarize the function's purpose and usage clearly, making them useful for quick reference within the code. Google-style documentation is most helpful for onboarding new developers because it systematically explains inputs, outputs, and behavior, improving readability and maintainability

Task:03

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

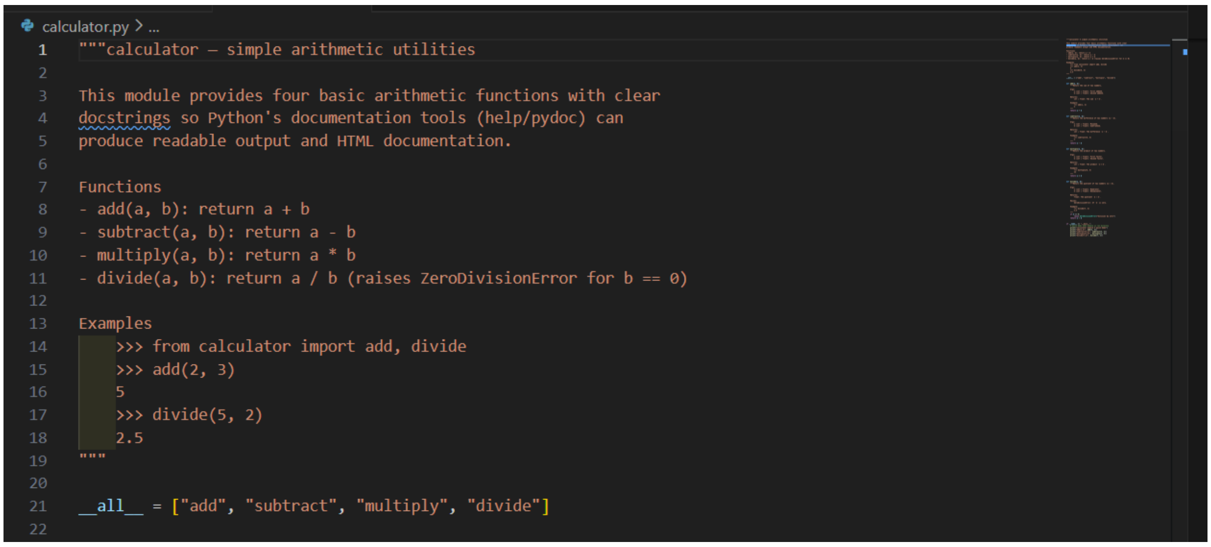
o multiply(a, b) – returns the product of two numbers

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

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

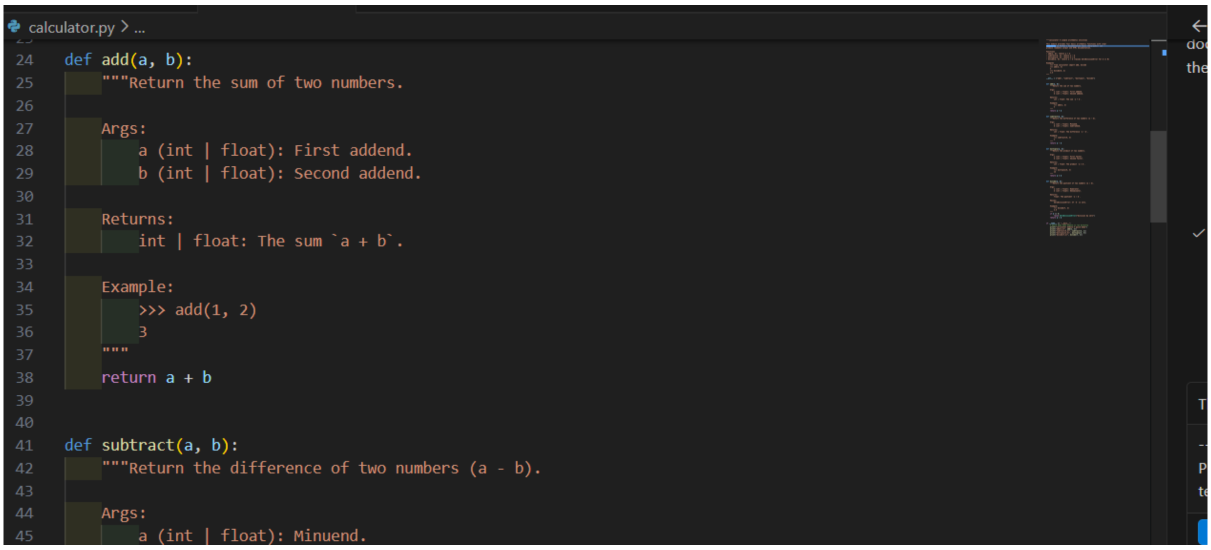
3. Generate and export the module documentation in HTML format using the pydoc utility, and open the generated HTML file in a web browser to verify the output.

Prompot: for this implement code for the calculateCode:



calculator.py > ...

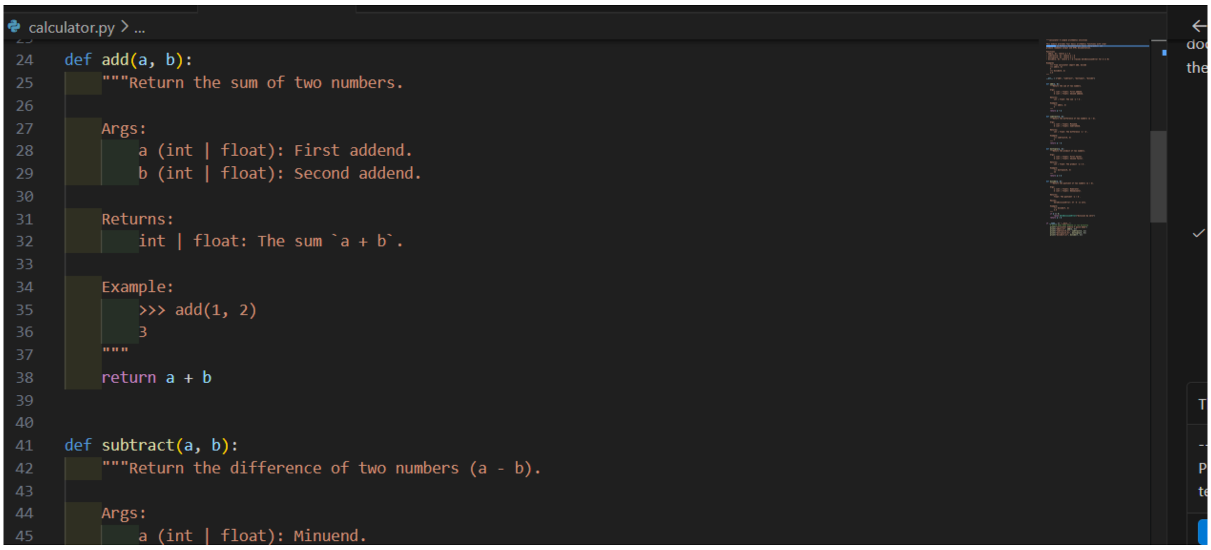
```python
"""calculator — simple arithmetic utilities

This module provides four basic arithmetic functions with clear
docstrings so Python's documentation tools (help/pydoc) can
produce readable output and HTML documentation.

Functions
- add(a, b): return a + b
- subtract(a, b): return a - b
- multiply(a, b): return a * b
- divide(a, b): return a / b (raises ZeroDivisionError for b == 0)

Examples
    >>> from calculator import add, divide
    >>> add(2, 3)
    5
    >>> divide(5, 2)
    2.5
"""

__all__ = ["add", "subtract", "multiply", "divide"]

```



calculator.py > ...

```python
def add(a, b):
    """Return the sum of two numbers.

    Args:
        a (int | float): First addend.
        b (int | float): Second addend.

    Returns:
        int | float: The sum `a + b`.

    Example:
        >>> add(1, 2)
        3
    """
    return a + b


def subtract(a, b):
    """Return the difference of two numbers (a - b).

    Args:
        a (int | float): Minuend.
```

Output:



```
calculator.py > ...
def add(a, b):
    """Return the sum of two numbers.

    Args:
        a (int | float): First addend.
        b (int | float): Second addend.

    Returns:
        int | float: The sum `a + b`.

    Example:
        >>> add(1, 2)
        3
    """
    return a + b


def subtract(a, b):
    """Return the difference of two numbers (a - b).

    Args:
        a (int | float): Minuend.
```

Observation:

Well-written docstrings enable automatic documentation generation tools like pydoc to produce clear module descriptions without extra effort.
Terminal documentation allows quick inspection during development, while HTML output provides a structured, shareable reference.
Automatic documentation improves consistency, reduces manual work, and is especially useful for maintaining reusable modules like a calculator library.

Task:04

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.

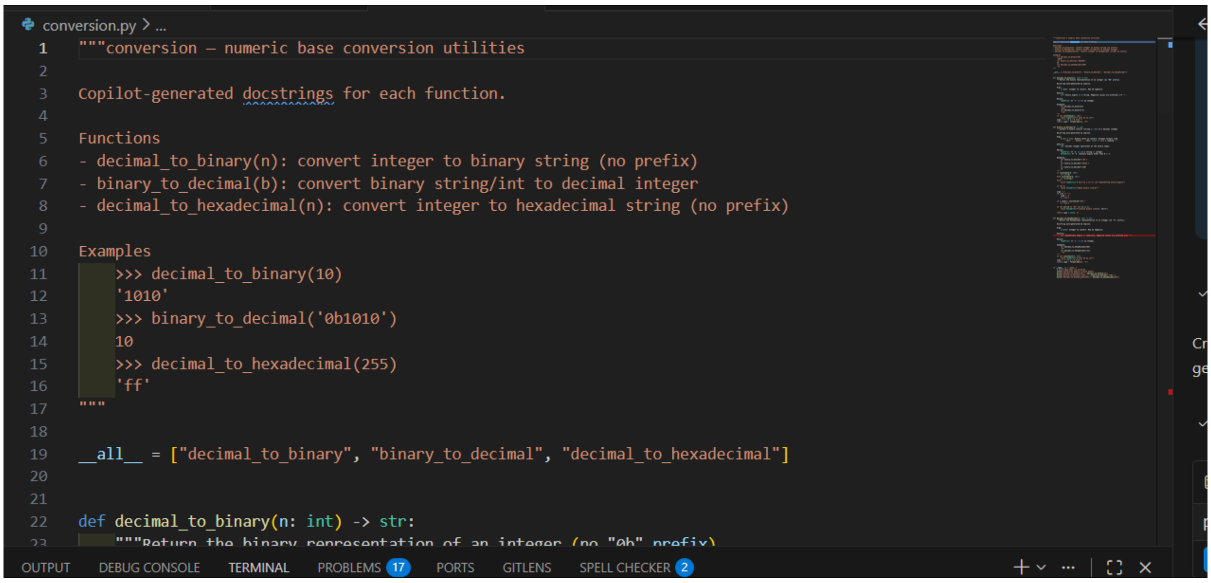
3. Generate documentation in the terminal.

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

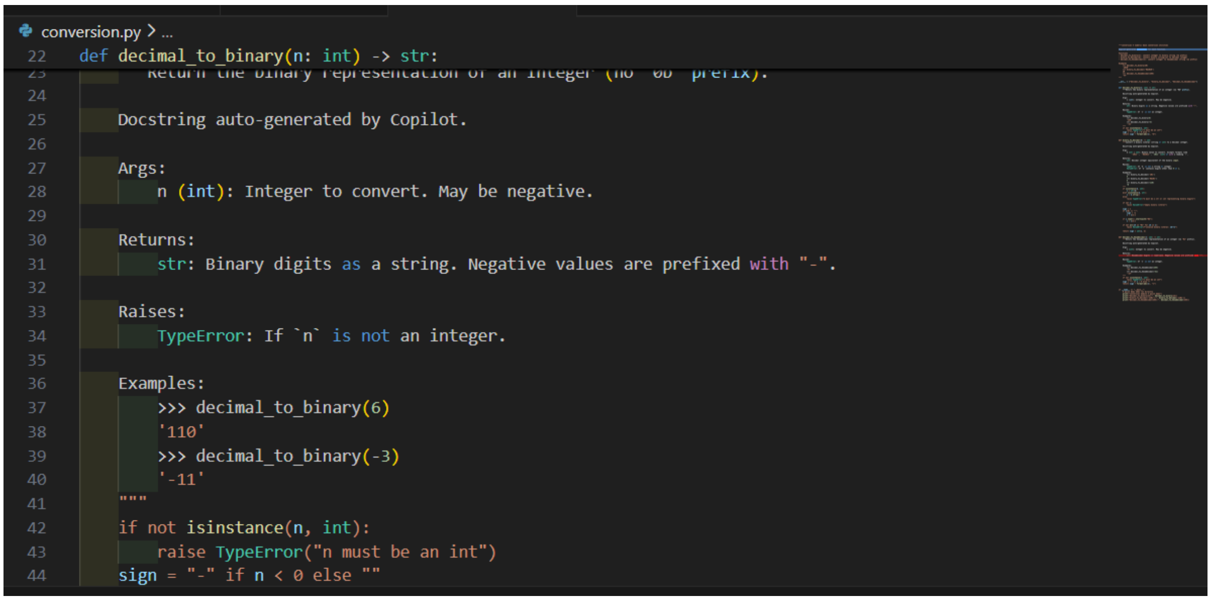
Browser

Prompt: for this code implement the user conversion

Code:

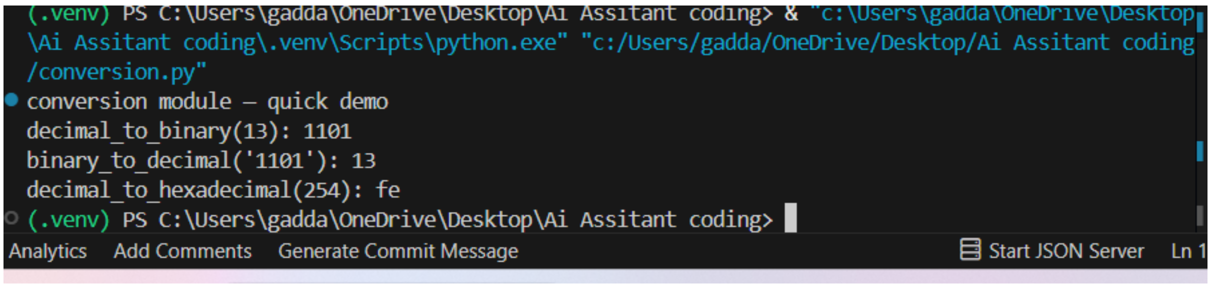


```
"""conversion — numeric base conversion utilities

Copilot-generated docstrings for each function.

Functions
- decimal_to_binary(n): convert integer to binary string (no prefix)
- binary_to_decimal(b): convert binary string/int to decimal integer
- decimal_to_hexadecimal(n): convert integer to hexadecimal string (no prefix)

Examples
    >>> decimal_to_binary(10)
    '1010'
    >>> binary_to_decimal('0b1010')
    10
    >>> decimal_to_hexadecimal(255)
    'ff'
"""


__all__ = ["decimal_to_binary", "binary_to_decimal", "decimal_to_hexadecimal"]


def decimal_to_binary(n: int) -> str:
    """Return the binary representation of an integer (no "0b" prefix)
```



```
def decimal_to_binary(n: int) -> str:
    """Return the binary representation of an integer (no "0b" prefix).

    Docstring auto-generated by Copilot.

    Args:
        n (int): Integer to convert. May be negative.

    Returns:
        str: Binary digits as a string. Negative values are prefixed with "-".

    Raises:
        TypeError: If `n` is not an integer.

    Examples:
        >>> decimal_to_binary(6)
        '110'
        >>> decimal_to_binary(-3)
        '-11'
    """
    if not isinstance(n, int):
        raise TypeError("n must be an int")
    sign = "-" if n < 0 else ""
```

Output:



```
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> & "c:\Users\gadda\OneDrive\Desktop
\Ai Assitant coding\.venv\Scripts\python.exe" "c:/Users/gadda/OneDrive/Desktop/Ai Assitant coding
/conversion.py"
conversion module — quick demo
decimal_to_binary(13): 1101
binary_to_decimal('1101'): 13
decimal_to_hexadecimal(254): fe
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observation:

Auto-generated docstrings (e.g., via Copilot) speed up development but should be reviewed for accuracy and clarity.
Terminal documentation provides quick access for developers, while HTML output offers a well-organized, user-friendly reference.
Automatic documentation ensures consistency and is highly beneficial for utility modules that may be reused across multiple projects.

Task 05:

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.

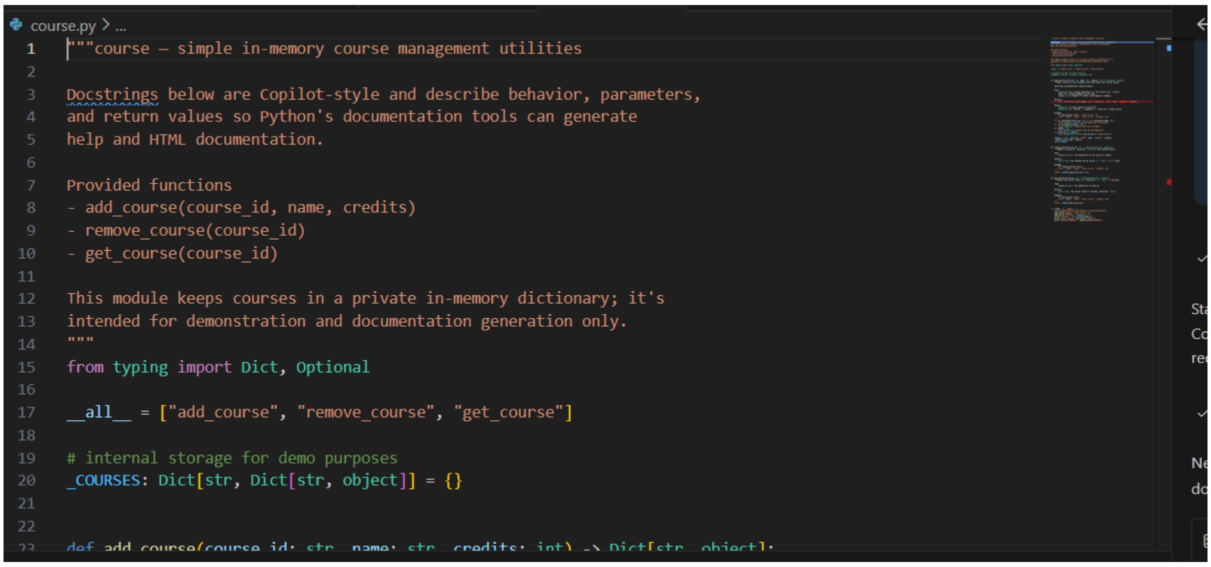
3. Generate documentation in the terminal.

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

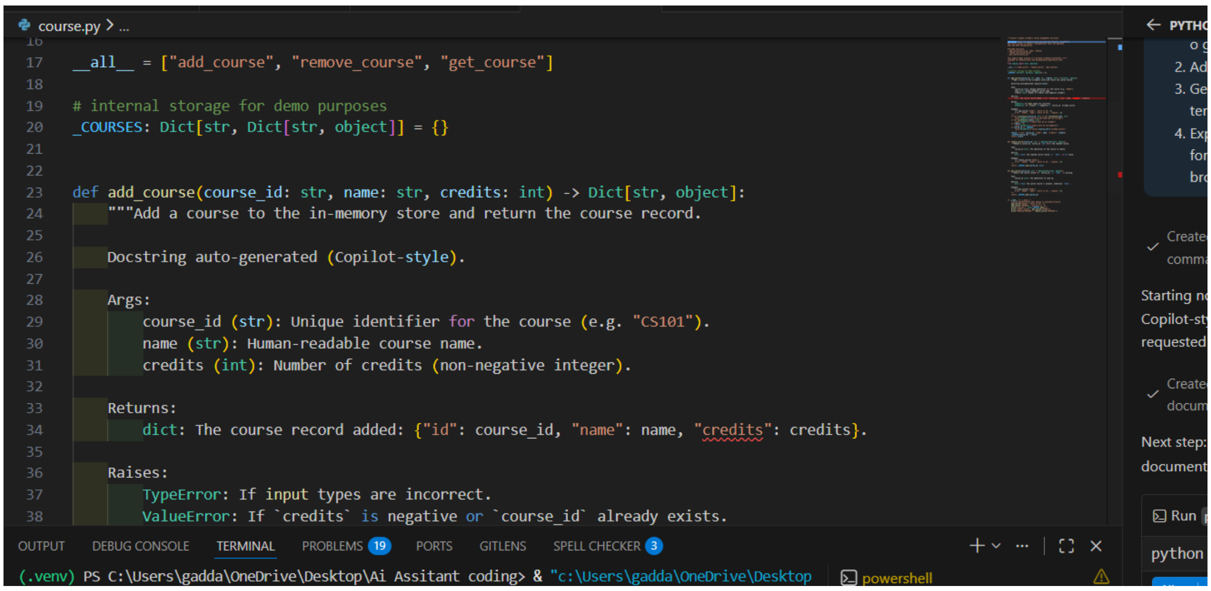
browser

prompt: for this generate the course management for all three documentation

Code


course.py > ...

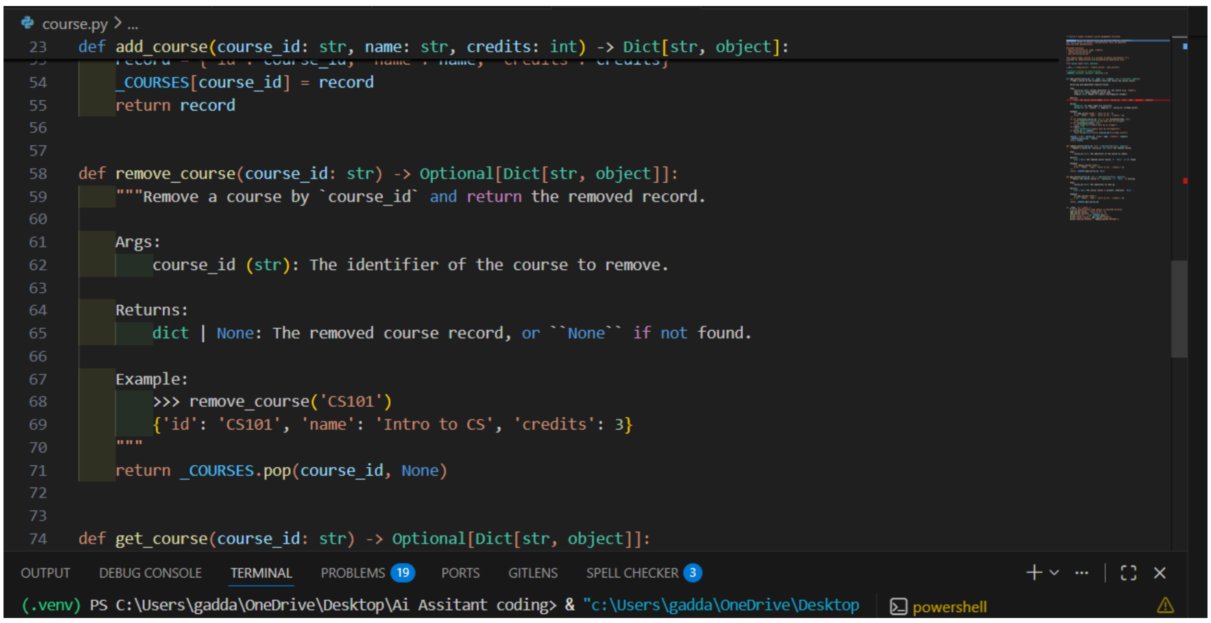
```python
"""course — simple in-memory course management utilities

Docstrings below are Copilot-style and describe behavior, parameters,
and return values so Python's documentation tools can generate
help and HTML documentation.

Provided functions
- add_course(course_id, name, credits)
- remove_course(course_id)
- get_course(course_id)

This module keeps courses in a private in-memory dictionary; it's
intended for demonstration and documentation generation only.
"""
from typing import Dict, Optional

__all__ = ["add_course", "remove_course", "get_course"]

# internal storage for demo purposes
_COURSES: Dict[str, Dict[str, object]] = {}


def add_course(course_id: str, name: str, credits: int) -> Dict[str, object]:
```


course.py > ...

```python
__all__ = ["add_course", "remove_course", "get_course"]

# internal storage for demo purposes
_COURSES: Dict[str, Dict[str, object]] = {}


def add_course(course_id: str, name: str, credits: int) -> Dict[str, object]:
    """Add a course to the in-memory store and return the course record.

    Docstring auto-generated (Copilot-style).

    Args:
        course_id (str): Unique identifier for the course (e.g. "CS101").
        name (str): Human-readable course name.
        credits (int): Number of credits (non-negative integer).

    Returns:
        dict: The course record added: {"id": course_id, "name": name, "credits": credits}.

    Raises:
        TypeError: If input types are incorrect.
        ValueError: If `credits` is negative or `course_id` already exists.
```

OUTPUT DEBUG CONSOLE TERMINAL PROBLEMS 19 PORTS GITLENS SPELL CHECKER 3

```
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> & "c:\Users\gadda\OneDrive\Desktop
```

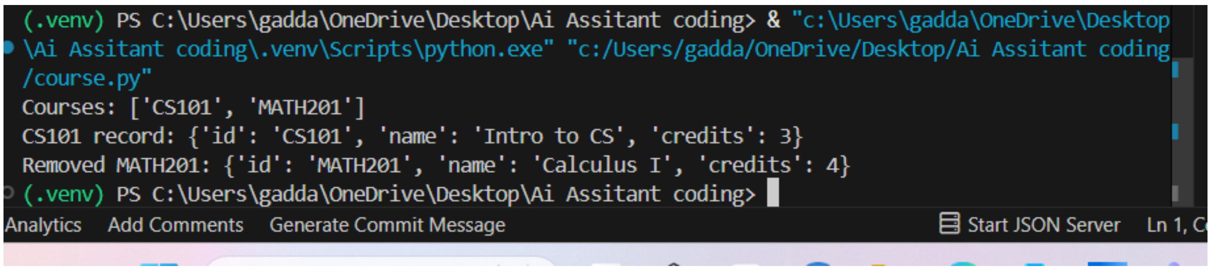

course.py > ...

```python
def add_course(course_id: str, name: str, credits: int) -> Dict[str, object]:
    record = {"id": course_id, "name": name, "credits": credits}
    _COURSES[course_id] = record
    return record


def remove_course(course_id: str) -> Optional[Dict[str, object]]:
    """Remove a course by `course_id` and return the removed record.

    Args:
        course_id (str): The identifier of the course to remove.

    Returns:
        dict | None: The removed course record, or ``None`` if not found.

    Example:
        >>> remove_course('CS101')
        {'id': 'CS101', 'name': 'Intro to CS', 'credits': 3}
    """
    return _COURSES.pop(course_id, None)


def get_course(course_id: str) -> Optional[Dict[str, object]]:
```

OUTPUT DEBUG CONSOLE TERMINAL PROBLEMS 19 PORTS GITLENS SPELL CHECKER 3

```
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> & "c:\Users\gadda\OneDrive\Desktop
```

Output:



```
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding> & "c:\Users\gadda\OneDrive\Desktop
\Ai Assitant coding\.venv\Scripts\python.exe" "c:/Users/gadda/OneDrive/Desktop/Ai Assitant coding
/course.py"
Courses: ['CS101', 'MATH201']
CS101 record: {'id': 'CS101', 'name': 'Intro to CS', 'credits': 3}
Removed MATH201: {'id': 'MATH201', 'name': 'Calculus I', 'credits': 4}
(.venv) PS C:\Users\gadda\OneDrive\Desktop\Ai Assitant coding>
```

Observations:

Copilot-generated docstrings help quickly document course management functions but should be validated to ensure correctness and completeness.
Terminal documentation is useful for developers during coding, whereas HTML documentation provides a structured reference for broader use.
Automatic documentation improves maintainability and clarity, especially for modules that manage structured data like courses.