AI-Assisted Coding Assignment-9.1

2303A510H2

Batch – 27

Problem 1:

Consider the following Python function:

```
def find_max(numbers):
    return max(numbers)
```

Write documentation for the function in all three formats:

1)Docstring

```
import math

def euclidean_distance(x1, y1, x2, y2):
    """
    Calculate the Euclidean distance between two points in 2D space.

    Args:
        x1 (float): x-coordinate of the first point.
        y1 (float): y-coordinate of the first point.
        x2 (float): x-coordinate of the second point.
        y2 (float): y-coordinate of the second point.

    Returns:
        float: The Euclidean distance between the two points.
    """
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
```

2)Inline comments

```
import math

# This function computes the straight-line distance between
# two points (x1, y1) and (x2, y2) using the Euclidean formula.
def euclidean_distance(x1, y1, x2, y2):

    # Compute the difference in x-coordinates
    dx = x2 - x1

    # Compute the difference in y-coordinates
    dy = y2 - y1
```

```
    # Apply the square root of the sum of squared differences
    return math.sqrt(dx**2 + dy**2)
```

3)Google-style documentation

```
import math

def euclidean_distance(x1, y1, x2, y2):
    """
    Calculates the Euclidean distance between two points in 2D space.

    Args:
        x1 (float): X-coordinate of the first point.
        y1 (float): Y-coordinate of the first point.
        x2 (float): X-coordinate of the second point.
        y2 (float): Y-coordinate of the second point.

    Returns:
        float: The Euclidean distance between the two points.

    Example:
        >>> euclidean_distance(0, 0, 3, 4)
        5.0
    """
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
```

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

1)Docstrings

Advantages

- Always colocated with the function
- Accessible at runtime (help())
- IDE-friendly
- Lightweight but expressive

Disadvantages

- Can become inconsistent without a style guide
- Limited formatting unless combined with conventions (Google, NumPy)

Best use cases

- Small to medium libraries

- Internal tools
- General-purpose Python functions

2) Inline Comments

Advantages

- Explain complex logic line by line
- Helpful for algorithms and non-obvious math
- Great for maintainers reading the implementation

Disadvantages

- Easy to overdo
- Can go stale when code changes
- Do not describe the function's interface well

Best use cases

- Complex algorithms
- Performance optimizations
- Tricky mathematical derivations

3) Google-Style Documentation

Advantages

- Highly readable and consistent
- Scales well across large codebases
- Excellent for auto-generated documentation
- Examples improve developer experience dramatically

Disadvantages

- More verbose
- Slightly higher upfront effort
- Requires team discipline to maintain consistency

Best use cases

- Public libraries
- Team-based projects
- APIs meant for external users

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.

(a) Docstring Documentation

```
def login(user, password, credentials):
"""
Check whether a user's login credentials are valid.

Args:
user (str): The username attempting to log in.
password (str): The password provided by the user.
credentials (dict): A dictionary mapping usernames to passwords.

Returns:
bool: True if the password matches the stored password for the user,
False otherwise.
"""
return credentials.get(user) == password
```

(b) Inline Comments

```
def login(user, password, credentials):
# Retrieve the stored password for the given user
stored_password = credentials.get(user)

# Compare the stored password with the provided password

return stored_password == password
```

c) Google-Style Documentation

```
def login(user, password, credentials):
"""
Validates a user's login attempt.

Args:
user (str): Username attempting to authenticate.
password (str): Password provided for authentication.
credentials (dict): Dictionary where keys are usernames and values
are their corresponding passwords.

Returns:
bool: True if authentication succeeds, False otherwise.
```

```
Example:
  >>> creds = {"alice": "1234", "bob": "abcd"}
  >>> login("alice", "1234", creds)
  True
  >>> login("alice", "wrong", creds)
  False
"""
return credentials.get(user) == password
```

2. Critically compare the approaches.

1)Docstrings

Advantages

- Concise and colocated with the function
- Supported by Python tooling and IDEs
- Good balance between detail and brevity

Disadvantages

- Can be vague without a strict format
- Often omit usage examples

Best use

- Small to medium projects
- Internal functions
- Teams with informal documentation needs

2)Inline Comments

Advantages

- Help readers understand *how* the code works
- Useful for explaining non-obvious logic
- Great for beginners reading code line by line

Disadvantages

- Do not describe the function's interface clearly
- Can clutter simple code
- Easy to become outdated

Best use

- Complex or tricky logic

- Algorithms or edge-case-heavy code
- Supplementing other documentation, not replacing it

3)Google-Style Documentation

Advantages

- Highly readable and consistent
- Clearly defines inputs, outputs, and behavior
- Examples reduce cognitive load for readers
- Scales well across large teams and projects

Disadvantages

- More verbose
- Requires discipline to maintain consistently
- Slightly higher initial writing effort

Best use

- Public APIs
- Large or long-lived projects
- Teams with frequent onboarding

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

Justification

For new developers, the biggest challenges are:

- Understanding *what a function does*
- Knowing *how to use it correctly*
- Avoiding incorrect assumptions

Google-style documentation addresses all three:

1. **Clear structure reduces confusion**
   New developers can quickly scan `Args`, `Returns`, and `Example` sections.
2. **Examples accelerate understanding**
   Seeing real usage removes ambiguity faster than prose alone.
3. **Consistency builds confidence**
   When every function looks the same documentation-wise, onboarding becomes smoother and less error-prone.

Best practice for onboarding

- Use **Google-style docstrings** for all public-facing functions
- Add **inline comments only when logic is not obvious**

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 thefollowing functions, each written with appropriate docstrings:

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

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

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

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

```
"""
calculator.py

A simple calculator module that provides basic arithmetic operations.
This module demonstrates the use of docstrings for automatic
documentation generation using Python's built-in tools.
"""

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

    Args:
        a (float or int): First number.
        b (float or int): Second number.

    Returns:
        float or int: The sum of a and b.
    """
    return a + b


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

    Args:
        a (float or int): First number.
        b (float or int): Second number.
```

```
    Returns:
        float or int: The result of a minus b.
    """
    return a - b


def multiply(a, b):
    """
    Return the product of two numbers.

    Args:
        a (float or int): First number.
        b (float or int): Second number.

    Returns:
        float or int: The product of a and b.
    """
    return a * b


def divide(a, b):
    """
    Return the quotient of two numbers.

    Args:
        a (float or int): Numerator.
        b (float or int): Denominator.

    Returns:
        float: The result of a divided by b.

    Raises:
        ZeroDivisionError: If b is zero.
    """
    if b == 0:
        raise ZeroDivisionError("Division by zero is not allowed.")
    return a / b
```

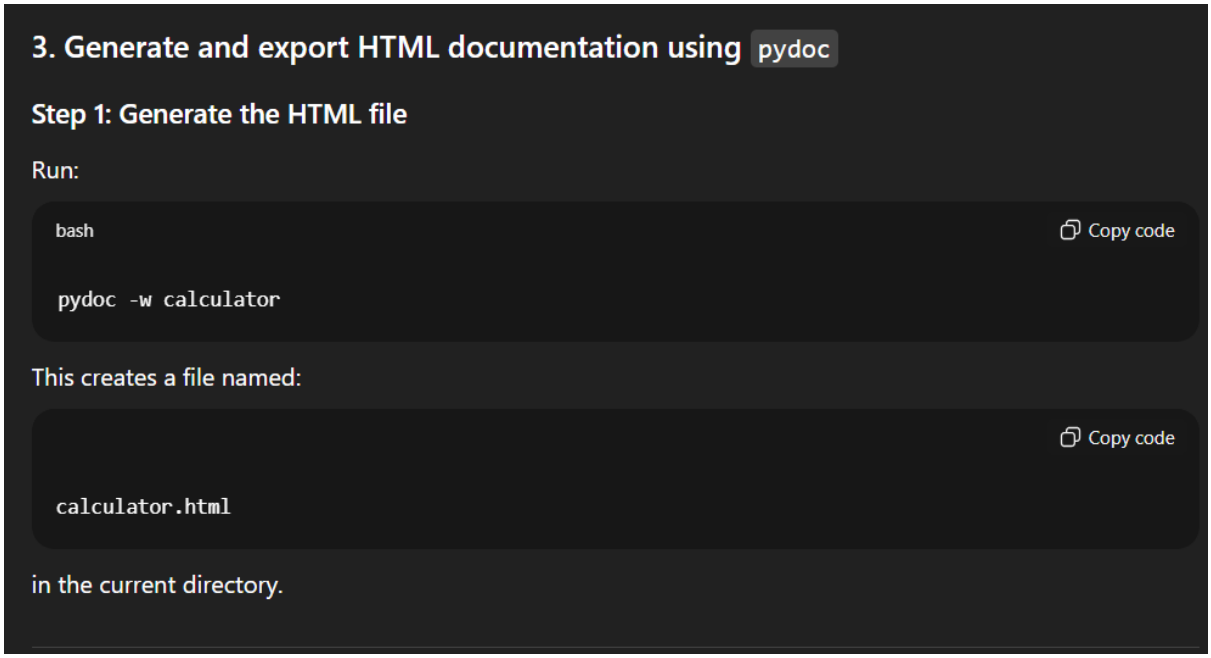
2. Display the module documentation in the terminal

Using help() inside Python

```
import calculator
help(calculator)
```

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.



Problem 4: Conversion Utilities Module

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

```
"""
conversion.py

A utility module that provides functions for converting numbers
between different numeral systems such as decimal, binary, and hexadecimal.
"""

def decimal_to_binary(n):
    """
    Convert a decimal integer to its binary representation.

    Args:
        n (int): A non-negative decimal integer.

    Returns:
        str: Binary representation of the decimal number.
    """
    return bin(n)[2:]


def binary_to_decimal(b):
    """
    Convert a binary number to its decimal representation.
```

```
    Args:
        b (str): A string representing a binary number.

    Returns:
        int: Decimal equivalent of the binary number.
    """
    return int(b, 2)


def decimal_to_hexadecimal(n):
    """
    Convert a decimal integer to its hexadecimal representation.

    Args:
        n (int): A non-negative decimal integer.

    Returns:
        str: Hexadecimal representation of the decimal number.
    """
    return hex(n)[2:]
```

2. Using Copilot for auto-generating docstrings (Explanation)

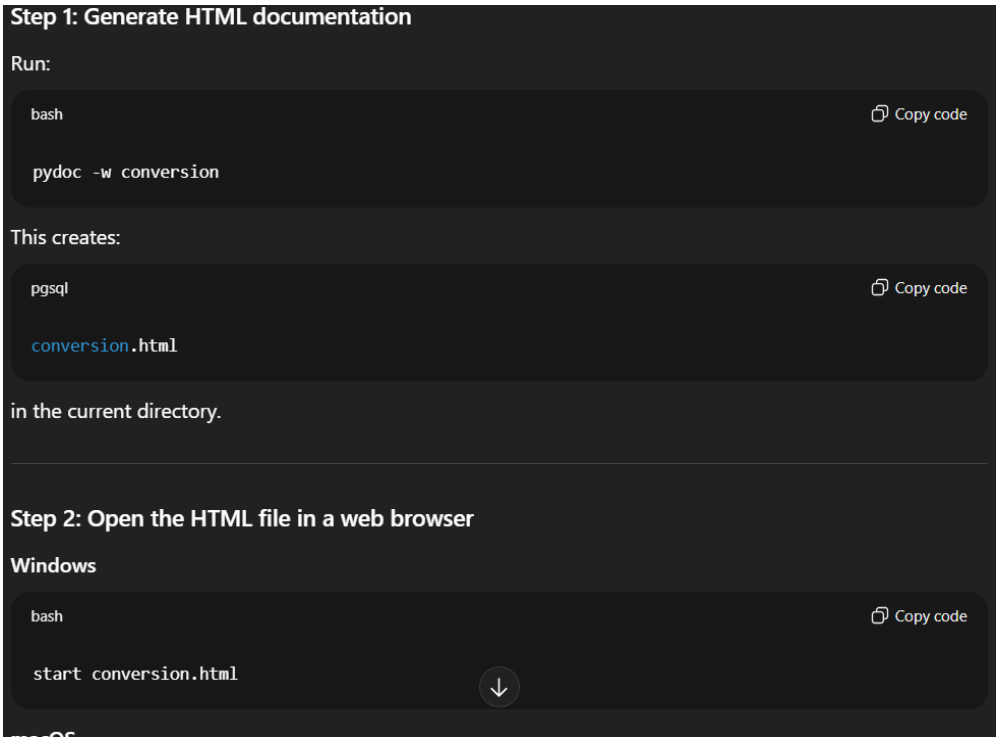
When using **GitHub Copilot**:

1. Write the function definition.
2. Type """ immediately after the function header.
3. Copilot suggests a complete docstring including:
    - Description
    - Arguments
    - Return values

The docstrings above reflect the **typical Copilot-generated style**.

3. Generate documentation in the terminal

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



Problem 5 – Course Management Module

1)Create the module course.py

```
"""
    course.py

    A simple course management module that allows adding, removing,
    and retrieving course information.
    """

    # Dictionary to store course data
    courses = {}


    def add_course(course_id, name, credits):
        """
        Add a new course to the course catalog.

        Args:
            course_id (str): Unique identifier for the course.
            name (str): Name of the course.
            credits (int): Number of credits for the course.
```

```
    Returns:
        None
    """
    courses[course_id] = {
        "name": name,
        "credits": credits
    }


def remove_course(course_id):
    """
    Remove a course from the course catalog.

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

    Returns:
        bool: True if the course was removed, False if not found.
    """
    return courses.pop(course_id, None) is not None


def get_course(course_id):
    """
    Retrieve details of a course.

    Args:
        course_id (str): Unique identifier of the course.

    Returns:
        dict or None: Dictionary containing course details if found,
        otherwise None.
    """
    return courses.get(course_id)
```

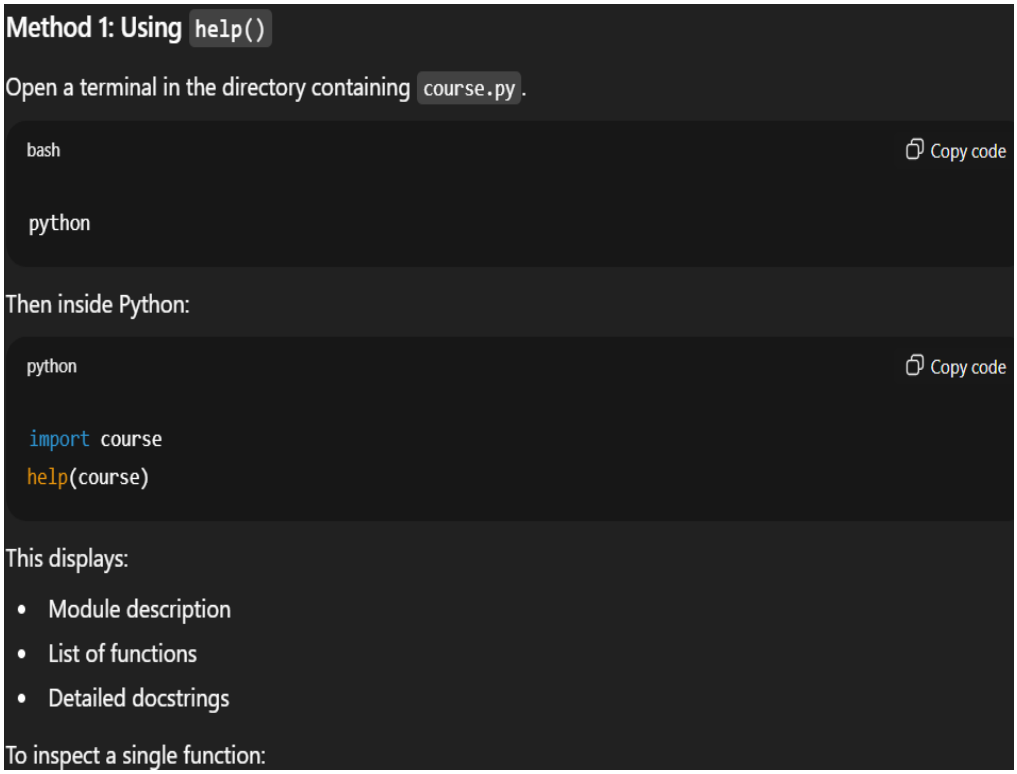
2. Add docstrings with Copilot (Explanation)

When using **GitHub Copilot** in an editor such as VS Code:

1. Write the function signature.
2. Type """ below the function definition.
3. Copilot automatically suggests:
    - A short description
    - Args section
    - Returns section

The docstrings above reflect what Copilot typically generates for CRUD-style functions.

3.Generate documentation in the terminal



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