

Assignment - 9.1

Lab Experiment: Documentation Generation -Automatic documentation and code comments

Lab Objectives

1. To understand automatic documentation generation.
2. To generate code comments and docstrings using AI tools.
3. To learn the importance of documentation in software development.

Lab Outcomes

1. Students will be able to generate documentation automatically for code.
2. Students will be able to add clear comments and docstrings to programs.
3. Students will be able to improve code readability and maintainability using documentation.

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.

PROMPT

#Generate documentation for the Python function
def find_max(numbers): return max(numbers) in
three specific formats:
(a)a standard one-line Docstring,
(b)using Inline comments for the logic, and
(c)a full Google-style docstring.
(d) After generating them, explain which is best
for a mathematical utilities library

CODE:

```
'''(a) Standard one-line Docstring:'''  
def find_max(numbers):  
    """Return the largest value from a non-empty list of  
    numeric values."""  
    return max(numbers)  
  
'''(b) Using Inline comments for the logic:'''  
def find_max(numbers):  
    # Uses Python's built-in max() to return the largest  
    element  
    # Assumes 'numbers' is a non-empty list of comparable  
    numeric values  
    # Raises ValueError automatically if the list is  
    empty  
    return max(numbers)  
  
'''(c) Full Google-style docstring:'''  
def find_max(numbers):  
    """  
    Return the maximum value from a list of numbers.  
  
    Args:  
        numbers (list[int | float]): Non-empty list of  
        numeric values.  
  
    Returns:  
        int | float: The largest value in the list.  
    """
```

Raises:

ValueError: If the list is empty.

TypeError: If elements are not comparable.

Example:

```
>>> find_max([3, 7, 2])
7
"""
if not numbers:
    raise ValueError("List cannot be empty")
return max(numbers)
```

Explanation:

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:

#Document the function def login(user, password, credentials): return credentials.get(user) == password using Docstrings, Inline comments, and Google-style documentation. Then, explain which style is most effective for onboarding new developers to a project and why.

CODE:

```
(a) Standard one-line Docstring:
def login(user, password, credentials):
    """Return True if the given username and password
    match stored credentials."""
```

```
return credentials.get(user) == password
```

(b) Using Inline comments for the logic:

```
def login(user, password, credentials):  
    # Fetch stored password for the given user from  
    the credentials dictionary  
    # If user does not exist, credentials.get(user)  
    returns None  
    # Compare stored password with provided password  
    # Return True if they match, otherwise False  
    return credentials.get(user) == password
```

(c) Full Google-style docstring:

```
def login(user, password, credentials):  
    """  
    Verify whether the provided login credentials are  
    valid.
```

Args:

```
    user (str): Username to authenticate.  
    password (str): Password provided by the  
user.  
    credentials (dict[str, str]): Mapping of  
usernames to passwords.
```

Returns:

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

Raises:

```
    TypeError: If credentials is not a  
dictionary.
```

Note:

```
    This function compares plain-text passwords  
and is not secure for real-world use.
```

Example:

```
>>> creds = {'alice': '1234'}
```

```
>>> login('alice', '1234', creds)
True
"""
if not isinstance(credentials, dict):
    raise TypeError("credentials must be a
dictionary")

return credentials.get(user) == password
```

Explanation:

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:
 - 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
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.

PROMPT:

#Create a Python module named calculator.py.
Include four functions: add, subtract, multiply,
and divide. Each function must have a clear
docstring explaining the parameters and the return
value. Ensure the module itself has a top-level
docstring describing its purpose

CODE:

```
# calculator.py
```

```
"""A simple calculator module that provides basic  
arithmetic operations: addition, subtraction,  
multiplication, and division.  
Each function takes two numeric parameters and  
returns the result of the operation.  
"""
```

```
def add(a, b):  
    """Return the sum of two numbers.  
  
    Args:  
        a (int | float): The first number.  
        b (int | float): The second number.  
  
    Returns:  
        int | float: The sum of a and b.  
    """  
    return a + b  
  
def subtract(a, b):  
    """Return the difference between two numbers.  
  
    Args:  
        a (int | float): The first number.  
        b (int | float): The second number.  
  
    Returns:  
        int | float: The difference of a and b.  
    """  
    return a - b  
  
def multiply(a, b):
```

```
"""Return the product of two numbers.
```

```
Args:
```

```
    a (int | float): The first number.  
    b (int | float): The second number.
```

```
Returns:
```

```
    int | float: The product of a and b.  
"""
```

```
return a * b
```

```
def divide(a, b):
```

```
    """Return the quotient of two numbers.
```

```
Args:
```

```
    a (int | float): The numerator.  
    b (int | float): The denominator.
```

```
Returns:
```

```
    int | float: The quotient of a and b.
```

```
Raises:
```

```
    ValueError: If b is zero, as division by zero  
is undefined.
```

```
"""
```

```
if b == 0:
```

```
    raise ValueError("Cannot divide by zero")
```

```
return a / b
```

Problem 4: Conversion Utilities Module

Task:

1. Write a module named conversion.py with functions:
 - decimal_to_binary(n)
 - binary_to_decimal(b)
 - 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:

conversion.py

#Create a Python module named conversion.py with a clear top-level docstring describing number system conversions (decimal, binary, hexadecimal). Implement three functions: decimal_to_binary(n), binary_to_decimal(b), and decimal_to_hexadecimal(n). Write simple, clear Google-style docstrings for each function and include basic input validation.

```
def decimal_to_binary(n):
```

```
    """
```

```
    Convert a decimal number to its binary representation.
```

```
    Args:
```

```
        n (int): A non-negative integer in decimal format.
```

```
    Returns:
```

```
        str: Binary representation of the number.
```

```
    Raises:
```

```
        ValueError: If n is negative.
```

```
    """
```

```
    if n < 0:
```

```
        raise ValueError("Input must be a non-negative integer")
```

```
    return bin(n)[2:]
```

```
def binary_to_decimal(b):
```

```
    """
```

```
    Convert a binary string to its decimal representation.
```

```
    Args:
```

b (str): A string containing a binary number (e.g., "1010").

Returns:

int: Decimal value of the binary number.

Raises:

ValueError: If input is not a valid binary string.

"""

if not all(bit in '01' for bit in b):

raise ValueError("Input must be a valid binary string")

return int(b, 2)

def decimal_to_hexadecimal(n):

"""

Convert a decimal number to its hexadecimal representation.

Args:

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

Returns:

str: Hexadecimal representation of the number.

Raises:

ValueError: If n is negative.

"""

if n < 0:

raise ValueError("Input must be a non-negative integer")

return hex(n)[2:]

Problem 5 – Course Management Module

Task:

1. Create a module course.py with functions:

- `add_course(course_id, name, credits)`
 - `remove_course(course_id)`
 - `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:

#Create a simple Python course management module with add, remove, and get functions using dictionary storage. Write short, natural docstrings like a student would write manually, avoiding technical or AI-style wording.Keep the code clean, readable, and suitable for a college lab record without mentioning AI tools.

```
# Internal storage for courses
_courses = {}
```

```
def add_course(course_id, name, credits):
```

```
    """
```

```
    Add a new course to the system.
```

```
    Args:
```

```
        course_id (str): Unique identifier for the course.
```

```
        name (str): Name of the course.
```

```
        credits (int): Number of credits assigned to the course.
```

```
    Returns:
```

```
        dict: Dictionary containing course details.
```

```
    Raises:
```

```
        ValueError: If course_id already exists or credits is negative.
```

```
    """
```

```
    if course_id in _courses:
```

```
        raise ValueError("Course already exists")
```

```
if credits < 0:  
    raise ValueError("Credits must be non-negative")
```

```
_courses[course_id] = {  
    "course_id": course_id,  
    "name": name,  
    "credits": credits  
}  
return _courses[course_id]
```

```
def remove_course(course_id):
```

```
    """
```

```
    Remove a course from the system.
```

```
    Args:
```

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

```
    Returns:
```

```
        bool: True if removed successfully, False if course 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 | None: Course details if found, otherwise None.
```

```
    """
```

```
    return _courses.get(course_id)
```