**HOW TO CREATE FUNCTION DOCUMENTATION**

Documentation of a function is key to helping developers understand how to use and operate the function. Good documentation not only improves work efficiency but also reduces errors during source code usage or development. In this article, we'll explore how to write effective documentation, required components, standards to follow, and important considerations.

**Purpose of Documentation**

Documentation plays a crucial role for the following reasons:

- Helps readers understand source code: For complex functions, documentation saves time in analyzing logic

- Ensures correct usage: Users clearly understand input parameters, return values, and conditions to follow

- Increases reusability: With clear documentation, functions can be easily reused in other projects without needing to re-read source code

- Easier maintenance and development: Documentation helps new programmers easily approach and fix bugs when needed

**Essential Components in Function Documentation**

Good documentation should provide complete information for readers. Basic sections include:

**a. Name and Brief Description**

The function name should be clear and self-explanatory. A brief description of the function's purpose helps readers understand immediately without diving into details.

Example:

```python

"""

Calculates the sum of two integers.

"""

```

**b. Purpose**

Clearly explain the function's purpose. For example, explain when to use this function and its benefits.

Example:

```python

"""

Purpose:

This function is used to calculate the sum of two integers. It's useful in basic

mathematical problems or arithmetic processing applications.

"""

```

**c. Syntax and Parameters**

List all input parameters:

- Parameter name: Should be clearly named, showing its function

- Data type: Specify the data type required by the function (integer, string, list, etc.)

- Explanation: State the meaning of the parameter and how it's used in the function

Example:

```python

"""

Parameters:

a (int): First integer number.

b (int): Second integer number.

"""

```

**d. Return Value**

Explain the value returned by the function. Include:

- Data type

- Meaning or purpose of the return value

Example:

```python

"""

Returns:

int: The sum of two input integers.

"""

```

**e. Usage Examples**

Provide one or more specific examples to illustrate how to use the function. This is especially useful for complex functions.

Example:

```python

"""

Example:

>>> add\_numbers(2, 3)

5

"""

```

**f. Exceptions (if any)**

List types of errors or exceptions the function might raise and conditions leading to them.

Example:

```python

"""

Raises:

ValueError: If either parameter is not an integer.

"""

```

**g. Special Notes**

Clearly state any limitations or special conditions users should be aware of when using the function.

Example:

```python

"""

Note:

This function only works with integers. If you pass float numbers,

they will be automatically rounded.

"""

```

**Documentation Standards**

There are several documentation styles in programming. Here are two popular styles:

**a. Google Style**

Google style stands out for its brevity and clarity. This is a popular choice in the Python community.

Example:

```python

def add\_numbers(a: int, b: int) -> int:

"""

Calculate the sum of two integers.

Args:

a (int): First integer number.

b (int): Second integer number.

Returns:

int: Sum of the two input integers.

Example:

>>> add\_numbers(2, 3)

5

"""

return a + b

```

**b. NumPy Style**

This style is suitable for complex functions, especially in data science.

Example:

```python

def add\_numbers(a: int, b: int) -> int:

"""

Calculate the sum of two integers.

Parameters

----------

a : int

First integer number.

b : int

Second integer number.

Returns

-------

int

Sum of the two input integers.

Examples

--------

>>> add\_numbers(2, 3)

5

"""

return a + b

```

**Documentation Best Practices**

**1. Concise and Complete:**

- No need to write too verbosely, but ensure all information is included

**2. Avoid Ambiguity:**

- Don't use ambiguous words like "maybe", "sometimes". Be specific

**3. Project Consistency:**

- Choose one style and maintain consistency

**4. Update Documentation with Code Changes:**

- Documentation must reflect the current state of the function

**5. Reader-Friendly:**

- Write documentation from the user's perspective, not just for yourself

**Example:**

def convert(n: int) -> int:

"""

Convert a decimal number to hexadecimal representation using recursion.

This function converts a decimal (base-10) number to its hexadecimal (base-16)

representation by recursively dividing by 16 and storing remainders. The result

is stored in a global list 'lst' where digits are inserted at the beginning.

Args:

n (int): A positive decimal number to convert to hexadecimal.

Must be non-negative.

Returns:

int: Always returns 0 (used for recursion control).

The actual hexadecimal result is stored in the global 'lst' list.

Notes:

- Requires a global list 'lst' to be initialized before calling

- Modifies the global 'lst' list by inserting hexadecimal digits

- Uses characters '0-9' for values 0-9 and 'A-F' for values 10-15

Examples:

>>> lst = []

>>> convert(255)

0

>>> print(''.join(lst))

'FF'

>>> lst = []

>>> convert(10)

0

>>> print(''.join(lst))

'A'

Warning:

The function relies on a global 'lst' variable which must be

initialized as an empty list before calling the function.

"""

hexchar = '0123456789ABCDEF'

lst.insert(0, hexchar[n % 16])

n = n // 16

if n == 0:

return 0

else:

return convert(n)