# 2.7 The Function Design Recipe

Often when beginners are tasked with writing a program to solve a problem, they jump immediately to writing code. Doesn't matter whether the code is correct or not, or even if they fully understand the problem: somehow the allure of filling up the screen with text is too tempting. So before we go further in our study of the Python programming language, we'll introduce the *Function Design Recipe*, a structured process for taking a problem description and designing and implementing a function in Python to solve this problem.

## *The Function Design Recipe by example*

Consider the following example problem: write a function to determine whether or not a number is even. We'll use this example to illustrate the five steps of the Function Design Recipe.

**1. Write example uses.**

Pick a name for the function (often a verb or verb phrase). Sometimes a good name is a short answer to the question "What does your function do?" Write one or two examples of calls to your function and the expected returned values. Include an example of a standard case (as opposed to a tricky case). Put the examples inside a triple-quoted string that you've indented since it will be the beginning of the docstring.

```
    """
    >>> is_even(2)
    True
    >>> is_even(17)
    False
    """
```

**2. Write the function header.**

Write the function header above the docstring (not indented). Choose a meaningful name for each parameter (often nouns). Include the type contract (the types of the parameters and return value).

```
def is_even(value: int) -> bool:
    """
    >>> is_even(2)
    True
    >>> is_even(17)
    False
    """
```

**3. Write the function description.**

Before the examples, add a description of what the function does and mention each parameter by name or otherwise make sure the purpose of each parameter is clear. Describe the return value.

```
def is_even(value: int) -> bool:
    """Return whether value is even

    >>> is_even(2)
    True
    >>> is_even(17)
    False
    """
```

**4. Implement the function body.**

Write the body of the function and indent it to match the docstring. To help yourself write the body, review your examples from the first step and consider how you determined the return values. You may find it helpful to write a few more example calls.

```
def is_even(value: int) -> bool:
    """Return whether value is even

    >>> is_even(2)
    True
    >>> is_even(17)
    False
    """
    return value % 2 == 0
```

**5. Test the function.**

Test your function on all your example cases including any additional cases you created in the previous step. Additionally, try it on extra tricky or corner cases.

One simple way to test your function is by calling it in the Python console. In the next section, we'll discuss more powerful ways of testing your code.

If you encounter any errors/incorrect return values, first make sure that your tests are correct, and then go back to Step 4 and try to identify and fix any possible errors in your code. This is called *debugging* your code, a process we'll discuss throughout this course.

## *The importance of documenting your functions*

The Function Design Recipe places a large emphasis on developing a precise and detailed function header and docstring before writing any code for the function body. There are two main benefits to doing this.

First, when you are given a programming task—"Write a function to do X"—you want to make sure you fully understand the goal of that function before trying to solve it. Forcing yourself to write out the function header and docstring, with examples, is an excellent way to reinforce your understanding about what you need to do.

Second, as you begin to work on larger projects and writing dozens or hundreds of functions, it is easy to lose track of what each function does. The function header and docstring serve as *documentation* for the function, communicating to others–and to your future self—what that function is supposed to to. Your choices for the function's name, its parameter names, its type contract, its docstring examples, and its description, can make the difference between code that is easy to work on and maintain, and code that is undecipherable.

So the bottom line is you should follow this process for all of the functions you'll write in this course, and beyond—trust us, it will save you lots of time and headaches!

## *References*

- CSC108 videos: Function Design Recipe (Part 1, Part 2)