2.2 Defining Our Own Functions

Python provides many built-in functions, but as we start writing more code ourselves, it will be essential that we be able to create our own functions specific to the problem we are solving. In this section, we'll learn how to define our own functions in Python.

First, let's recall how we define a function in mathematics. We first specify the function name, domain, and codomain, for example, $f : \mathbb{R} \to \mathbb{R}$. Then, we write the function header and body, usually in a single line: for example, $f(x) = x^2$. We do this so often in mathematics that we often take parts of this for granted, for example leaving out the domain/codomain specification, and usually choosing f as the function name and x as the parameter name. However, the functions we'll implement in Python are much more diverse, and so it will be important to be explicit in every part of this process.

Defining a Python function

Here is the complete definition of a "squaring" function in Python. Take a moment to read through the whole definition, and then continue reading to learn about this definition's different parts.

```python
def square(x: float) -> float:
    """Return x squared.

    >>> square(3.0)
    9.0
    >>> square(2.5)
    6.25
    """
    return x ** 2
```

This function definition is the most complex form of Python code we've seen so far, so let's break this down part by part.

Function header

The first line, `def square(x: float) -> float:` is called the **function header**. Its purpose is to convey the following pieces of information:

- The function's name (`square`).
- The number and type of arguments the function expects. A **parameter** is a variable in a function definition that refers to a argument when the function is called. In this example, the function has one parameter with name `x` and type `float`.
- The function's **return type**, which is the type written after the `->`. In this example, the function return type is `float`.[1]

[1] For `square`, its parameter and return type are the same, but this doesn't have to be the case in general.

The syntax for a function header for a unary function is:

```
def <function_name>(<parameter_name>: <parameter_type>) -> <return_type>:
```

Compared to our mathematical version, there are two main differences. First, we chose the name `square` rather than `f` as the function name; in Python, we will always pick descriptive names for our functions rather than relying on the conventional "f". And second, we use data types to specify the function domain and codomain: the code `x: float` specifies that the parameter `x` must be a `float` value, and the code `-> float` specifies that this function always returns a `float` value.

We can express this restriction in an analogous way to $\mathbb{R} \to \mathbb{R}$ by writing `float -> float`; we call `float -> float` the **type contract** of the `square` function.

Function docstring

The next seven lines, which start and end with triple-quotes (`"""`), is called the **function docstring**. This is another way of writing a comment in Python: text that is meant to be read by humans, but not executed as Python code. The goal of the function docstring is to communicate what the function does. The function docstring is indented inside the function header, as a visual indicator that it is part of the overall function definition.[2]

The first part of the docstring, `Return x squared.`, is an English description of the function. The second part might look a bit funny at first, since it seems like Python code:[3]

[2] Unlike many other programming languages, this kind of indentation in Python is **mandatory** rather than merely recommended. Python's designers felt strongly enough about indentation improving readability of Python programs that they put indentation requirements like this into the language itself.
[3] Or more precisely, it looks like the Python console!

```
>>> square(3.0)
9.0
>>> square(2.5)
6.25
```

This part of the docstring shows example uses of the function, just like the examples we showed of built-in functions in the previous section. You can read the first example literally as "evaluating `square(3.0)` in the Python console returns `9.0`" and the second as "evaluating `square(2.5)` in the Python console returns `6.25`". These examples are called **doctest examples**, for a reason we'll see in a future section. While an English description may technically be enough to specify the function's behaviour, doctest examples are invaluable for aiding understanding of the function behaviour (which is why we use them in teaching as well!).

Function body

The final line, `return x ** 2`, is called the **body** of the function, and is the code that is executed when the function is called. Like the function docstring, the function body is also indented so that it is "inside" the function definition.

This code uses another keyword, `return`, which signals a new kind of statement: the **return statement**, which has the form:

```
return <expression>
```

When the Python interpreter executes a return statement, the following happens:

1. The `<expression>` is evaluated, producing a value.
2. That value is then returned to wherever the function was called. No more code in the function body is executed after this point.

Defining functions in files

Okay, so that's a function definition. But how do we actually use such a definition in Python? While it is possible to define functions directly in the Python console, this isn't a good approach: every time we restart the Python console, we lose all our previous functions definitions and have to re-type them.

So instead, we save functions in files so that we can reuse them across multiple sessions in the Python console (and in other files, as we'll see later). For example, we can create a new file called `my_functions.py`, and write our function definition in that file:[4]

[4] We strongly encourage you to follow along with this in PyCharm!

```python
def square(x: float) -> float:
    """Return x squared.

    >>> square(3.0)
    9.0
    >>> square(2.5)
    6.25
    """
    return x ** 2
```

Then after saving the file, we can right-click on the file in PyCharm and select "Run File in Python Console". This will start the Python console and run our file, which then allows us to call our function `square` just like any built-in function:

```
>>> square(3.0)
9.0
```

Function docstrings and `help`

It turns out that function docstrings aren't just regular comments—they are are read and stored by the Python interpreter at each function definition. You might wonder why, though, if these docstrings are just text that explain what a function is supposed to do.

Here is one very cool reason: *a function's docstring is part of the documentation that is displayed when a user calls `help` on that function*. So when we used `help` to look at some of the built-in functions in the previous section, the Python interpreter was actually showing us the docstrings of those functions! To demonstrate this, let's call `help` on our `square` function in the Python console:

```
>>> help(square)
Help on function square in module my_functions:

square(x: float) -> float
    Return x squared.

    >>> square(3.0)
    9.0
    >>> square(2.5)
    6.25
```

Neat!

Defining functions with multiple parameters

Let's now look at a more complex example that will illustrate a function definition that takes in more than one parameter.

Recall the distance formula from Section 1.6 to calculate the distance between two points $(x_1, y_1), (x_2, y_2)$ in the Cartesian plane:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

We'll now write a function in Python that calculates this formula. This function will take two inputs, where each input is a `list` of two `float`s, representing the x- and y-coordinates of each point. When we define a function with multiple parameters, we write the name and type of each parameter using the same format we saw earlier, with parameters separated by commas from each other. Here is the function header and docstring:

```python
def calculate_distance(p1: list, p2: list) -> float:
    """Return the distance between points p1 and p2.

    p1 and p2 are lists of the form [x, y], where the x- a

    >>> calculate_distance([0, 0], [3.0, 4.0])
    5.0
    """
```

In order to use the above formula, we need to extract the coordinates from each point. This is a good reminder of `list` indexing, and also introduces a new concept: function bodies can consist of more than one statement.

```python
    # The start of the body of calculate_distance
    x1 = p1[0]
    y1 = p1[1]
    x2 = p2[0]
    y2 = p2[1]
```

Now that we have the four coordinates, we can apply the above formula and return the result.

```python
    # Continuing the function body
    return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
```

Putting this all together, we have:

```python
def calculate_distance(p1: list, p2: list) -> float:
    """Return the distance between points p1 and p2.

    p1 and p2 are lists of the form [x, y], where the x- and y-coordinates are points.

    >>> calculate_distance([0, 0], [3.0, 4.0])
    5.0
    """
    x1 = p1[0]
    y1 = p1[1]
    x2 = p2[0]
    y2 = p2[1]
    return ((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5
```

If we save this function in our `my_functions.py` file from earlier, we can then "Run File in Python Console" and call this function in the Python console:

```
>>> calculate_distance([0, 0], [3.0, 4.0])
5.0
```

Calling one function from another

Our above function body is perfectly correct, but you might notice that the `** 2` expressions exactly mimic the body of the first function we defined in this section: `square`. And so we can call our `square` function inside the body of `calculate_distance`:

```python
def calculate_distance(p1: list, p2: list) -> float:
    """Return the distance between points p1 and p2.

    p1 and p2 are lists of the form [x, y], where the x- and y-coordinates are points.

    >>> calculate_distance([0, 0], [3.0, 4.0])
    5.0
    """
    x1 = p1[0]
    y1 = p1[1]
    x2 = p2[0]
    y2 = p2[1]
    return (square(x2 - x1) + square(y2 - y1)) ** 0.5
```

This example is still relatively small, so you might wonder why it's useful to replace the `(x2 - x1) ** 2` with `square(x2 - x1)`; isn't that roughly the same amount of typing?

That's true in this case, but we wanted to use the example to illustrate a more general principle: it is possible to call a function from inside the body of another function. This is a very useful principle that comes in handy when our programs grow larger and more complex: we'll see how we can split up a complex block of code into separate functions that call each other, enabling us to better organize our code. We'll explore this idea in more detail, and other principles of good function and program design, throughout this course.

What happens when a function is called?

We'll wrap up this section with an introduction to some of the technical details of function calls in Python. In the previous section, we called built-in functions and took for granted that they worked properly, without worrying about how they work. Now that we're able to define our own functions, we are ready to fully understand what happens when a function is called.

As an example, suppose we've defined `square` as above, and then call it in the Python console:

```
>>> square(2.5)
```

When we press Enter, the Python interpreter evaluates the function call by doing the following:

1. Evaluate the argument `2.5`, and then assign its value to the function parameter `x`.[5]
2. Evaluate the body of the `square` function, by doing:
   a. First evaluate `x ** 2`, which is `6.25` (since `x` refers to the value `2.5`).
   b. Then stop executing the function body, and return the value `6.25` back to the Python console.
3. The function call `square(2.5)` evaluates to `6.25`, and this is displayed on the screen.

[5] This is analogous to the concept of *substituting* the value `2.5` for variable `x` that you use for evaluating mathematical function calls.

Multiple function calls in one expression

We saw in the previous chapter that expressions can be combined together to form larger expressions, for example `(4 * 5) + (3 * 8)`. Since function calls are a type of expression, we can also combine them in larger expressions, like we did in our `calculate_distance` example.

What does the Python interpreter do, exactly, when we call `square` twice in the same expression? For example, let's consider typing the following into the Python console and pressing Enter:

```
>>> square(2.5) + square(-1.0)
```

In the description below, notice how we've duplicated the text from before to illustrate the similarities between calling `square(2.5)` and `-square(-1.0)`.

1. The Python interpreter evaluates the operands to `+` in left-to-right order, so evaluate `square(2.5)` first.
   i. Evaluate `2.5`, and then assign `2.5` to the function parameter `x`.
   ii. Evaluate the body of the `square` function, by doing:
      a. First evaluate `x ** 2`, which is `6.25` (since `x` refers to `2.5`).
      b. Then stop executing the function body, and return the value `6.25` back to the Python console.
2. Nothing is displayed yet! There's still `square(-1.0)` to be evaluated.
   i. Evaluate `-1.0`, and then assign `-1.0` to the function parameter `x`.
   ii. Evaluate the body of the `square` function, by doing:
      a. First evaluate `x ** 2`, which is `1.0` (since `x` refers to `-1.0`).
      b. Then stop executing the function body, and return the value `1.0` back to the Python console.
3. Now the expression to evaluate has been simplified to `6.25 + 1.0`, which evaluates to `7.25`. This value is displayed on the screen.

References

- CSC108 videos: Defining Functions (Part 1, Part 2)
- CSC108 videos: Docstrings and Function `help` (Video)
- CSC108 videos: Function Reuse (Part 1, Part 2, Example)