4.1 Specifying What a Function Should Do

One of the most central questions in software development is, "How do we know that the software we write is correct?" Certainly, writing test cases will ensure that our functions produce the expected output for specific situations. But as our programs increase in complexity, how confident can we be that our test cases are sufficient?

In this chapter, we'll explore the question of function correctness through multiple approaches. First, we'll start by investigating what it means for functions to be correct in the first place—something that seems pretty straightforward, but with some subtleties that are important to get right. Then, we'll explore some software tools that we can use in this course to help us check that our functions are correct, or at least partially correct. And finally, we'll extend our study of mathematical logic into the study of mathematical *proofs*, and see how formal proofs can be used as a powerful tool to verifying the correctness of a function implementation.

Function specification and correctness

A **function specification** consists of two parts:

1. A description of what values the function takes as valid inputs. We can represent this description as a set of predicates, where a valid input to the function must satisfy *all* predicates. We call these predicates the **preconditions** of the function.
2. A description of what the function returns/does, in terms of its inputs.[1] We can represent this description as a set of predicates as well, that must all be satisfied by the return value of the function. We call these predicates the **postconditions** of the function.

[1] For now, all of our Python functions only return values, and do nothing else. Later on in the course, we'll study other kinds of function behaviour that could be included in a specification.

With these two parts, a function's specification defines what we expect the function to do. The job of an implementation of the function is to provide the Python code in the function body that meets this specification. We say that a function's implementation is **correct with respect to its specification** when the following holds: *For all inputs that satisfy the specification's preconditions, the function implementation's return value satisfies the specification's postconditions.*

A function specification acts as a contract or agreement between the person who implements the function and the person who calls the function. For the person implementing the function, their responsibility is to make sure their code correctly returns or does what the specification says. When writing this code, they do not need to worry about exactly how the function is called and *assume* that the function's input is always valid.[2] For the person writing the code that calls the function, their responsibility is to make sure they call the function with valid inputs. When they make this call, they do not need to worry about exactly how the function is implemented and *assume* that the function works correctly.

[2] So in fact, we have already seen several preconditions in this course. Every time we had a function description that said "you may ASSUME that…", that was a precondition.

The concept of a function specification is a very powerful one, as it spreads the responsibility of function correctness across two parties that can do their work separately—as long as they both know what the function specification is. As a result, these specifications must be very precise. Outside of software, lawyers are hired to draft and review contracts to make sure that they are defensible in the eyes of the law. Similarly, programmers must behave as lawyers when designing software to write ironclad contracts that leave no ambiguity in what is expected of the user or how the software will behave. In this section, we introduce some new tools and terminology that can help our functions be more explicit in their requirements and behaviour.

Simple specifications

Even though we haven't formally introduced the notion of a function specification until this section, you've been writing specifications all along simply by following the Function Design Recipe introduced in 2.7 The Function Design Recipe. Let's take a look at an early example:

```python
def is_even(n: int) -> bool:
    """Return whether n is even.

    >>> is_even(1)
    False
    >>> is_even(2)
    True
    """
    # Body omitted.
```

Here, the *type contract* and *description* actually form a complete specification of this function's behaviour:

1. The type annotation of the parameter `n` tells us that the valid inputs to `is_even` are `int` values. The type annotation `int` is itself a *precondition* of the function.
2. Similarly, the type annotation for the return value tells us that the function will always return a `bool`. In addition, the description "Return whether n is even." specifies the relationship between the function's return value and its input.[3] The function description and return type annotation specify the *postconditions* of the function.

[3] The doctest examples aid understanding, but are not strictly required to specify what this function does.

From this alone, we know what it means for this function to be implemented correctly, even if we can't see the implementation.

> `is_even` is implemented correctly when *for all* `int`*s* `n`, `is_even(n)` *returns a* `bool` *that is* `True` *when* `n` *is even, and* `False` *when* `n` *is not even.*

For example, suppose David has implemented this function. Mario loads this function implementation into the Python console and calls it:

```python
>>> is_even(4)
False
```

In this case, `4` is an `int`, so Mario held up his end of the contract when he called the function. But the `False` return value is inconsistent with the function description, and so we know there must be an error in the implementation—David is at fault, not Mario.

Suppose David fixes his implementation, and asks Mario to try another call. Mario types in:

```python
>>> is_even(4)
True
```

Okay pretty good, and now Mario tries:

```python
>>> is_even([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in is_even
TypeError: unsupported operand type(s) for %: 'list' and '
```

In this case, the function did not produce a return value but rather an error (i.e., `TypeError`). Is David at fault again? *No!* Mario violated the function's precondition by passing in a `list` rather than an `int`, and so he should have no expectation that `is_even` will meet its postcondition. Therefore, Mario (the caller of the function) caused the error.

Preconditions in general

All parameter type annotations are preconditions for a function. But often these type annotations are not precise enough to specify the exact set of valid inputs. Consider this function:

```python
def max_length(strings: set) -> int:
    """Return the maximum length of a string in the set of strings.

    >>> max_length({'Hello', 'Mario', 'David Liu'})
    9
    """
    return max({len(s) for s in strings})
```

What happens when the set is empty? Let's try it out in the Python console:

```python
>>> empty_set = set()
>>> max_length(empty_set)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 7, in max_length
ValueError: max() arg is an empty sequence
```

We've obtained an error, rather than an `int`; this makes logical sense, because it is impossible to find the maximum value in a set that contains no values at all. But from a formal function specification sense, who is to blame: the function's caller or the function's implementer?

As it stands, the implementer is at fault because the only description of "valid inputs" given is the type annotation `set`; the empty set is still a `set`. So we need to update the specification to rule out this possibility, but how? We encountered this issue in 3.3 Filtering Collections, when we wanted to restrict a statement to apply to a subset of our domain. Here we're doing the same thing: making the set of valid function inputs more specific, because we only want to guarantee our implementation works correctly on those inputs. We add a precondition to the function docstring as follows:

```python
def max_length(strings: set) -> int:
    """Return the maximum length of a string in the set of strings.

    Preconditions:
      - strings != set()
    """
    return max({len(s) for s in strings})
```

Whenever possible, we'll express these general preconditions as valid Python expressions involving the function's parameters.[4] In English, we would say that the full specification of `max_length`'s valid inputs is "`strings` is a `set`, and `strings != set()`". As functions get more complex, we can add additional preconditions by listing them under the header `Preconditions:` in the docstring. **A function input is valid when it satisfies the type annotations and *all* general precondition expressions**.

[4] Sometimes we'll encounter a precondition that is extremely complex, in which case you can write them in English.

Note that adding the precondition to the docstring does not change the behaviour of the function. If an empty set is passed into the function by the user, the function will still produce the `ValueError` we saw above. However, now that the precondition has been documented in the function specification, if we call `max_length(empty_set)`, we know that the error is entirely our fault because we violated a precondition.

Preconditions as assumptions and restrictions

Preconditions allow the implementer of a function to specify assumptions about the function's inputs, and so simplify the work of the implementer. On the other hand, preconditions place restrictions on the user of the function; the onus is on them to respect these preconditions every time the function is called. This often increases the complexity of the code that calls the function. For example, in our `max_length` function, the calling code might need an if statement to first check whether a set is empty before passing it to `max_length`.

When confronted with an "invalid input", there is another strategy other than simply ruling out the invalid input with a precondition: explicitly defining some alternate function behaviour for this input. Here is another way we could define `max_length`:

```python
def max_length(strings: set) -> int:
    """Return the maximum length of a string in the set of strings.

    Return 0 if strings is empty.
    """
    if strings == set():
        return 0
    else:
        return max({len(s) for s in strings})
```

Here, we picked a reasonable default value for `max_length` when given an empty set,[5] and then handled that as an explicit case in our implementation by using an if statement. Our function implementation is more complex than before, but now another person can call our function on an empty set without producing an error:

[5] This is very similar to how we define empty sums and products by a mathematical convention.

```python
>>> empty_set = set()
>>> max_length(empty_set)
0
```

You're probably wondering: is this version of `max_length` better or worse than our original one with the precondition? This version resulted in a longer description and function body, but it also removed a possible error we might encounter when calling the function. On the other hand, is 0 really a "reasonable" return value for this function in this case? Because this is ultimately a design decision, there is no clear "right answer"—there are always trade-offs to be made. Rather than sticking with a particular rule (i.e., "[always/never] use preconditions"), it's better to use broader principles to evaluate different choices. How much complexity is added by handling an additional input in a function implementation? Are there "reasonable" behaviours defined for a larger set of inputs than what you originally intended? The trade-offs are rarely clear cut.

That's not all!

It turns out that with either of the "precondition" or "reasonable default" strategies, our specification of `max_length` is still incomplete. Before moving onto the next section, take a moment to study our two implementations of this function and try to guess what is still missing from the specification!