4.3 Checking Function Specifications with `python_ta`

While our previous example illustrates how to document preconditions as part of a function specification, it has one drawback: it relies on whoever is calling the function to read the documentation! Of course, reading documentation is an important skill for any computer scientist, but despite our best intentions we sometimes miss things. It would be nice if we could turn our preconditions into executable Python code so that the Python interpreter checks them every time we call the function.

Checking preconditions with assertions

One way to do this is to use an `assert` statement, just like we do in unit tests. Because we've written the precondition as a Python expression, we can convert this to an assertion by copy-and-pasting it at the top of the function body.

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

    Preconditions:
      - strings != set()
    """
    assert strings != set()  # Check the precondition
    return max({len(s) for s in strings})
```

Now, the precondition is checked every time the function is called, with a meaningful error message when the precondition is violated:

```
>>> 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
AssertionError
```

We can even improve the error message we get by using an extended syntax for `assert` statements, where we include a string message to display after the boolean expression being checked:

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

    Preconditions:
      - strings != set()
    """
    assert strings != set(), 'Precondition violated: max_length called on an empty set.'
    return max({len(s) for s in strings})
```

Calling `max_length` on an empty set raises the same `AssertionError` as before, but now displays a more informative error message:

```
>>> 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
AssertionError: Precondition violated: max_length called on an empty set.
```

However, this approach of copy-and-pasting preconditions into assertions is tedious and error-prone. First, we have to duplicate the precondition in two places. And second, we have increased the size of the function body with extra code. And worst of all, both of these problems increase with the number of preconditions! *There must be a better way.*

Enter `python_ta`

The `python_ta` library we use in this course has a way to automatically check preconditions for all functions in a given file. Here is an example (using the new import-from statement we saw in the previous section):

```
from python_ta.contracts import check_contracts


@check_contracts
def max_length(strings: set[str]) -> int:
    """Return the maximum length of a string in the set of strings.

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

Notice that we've kept the function docstring the same, but removed the assertion. Instead, we are importing a new module (`python_ta.contracts`), and then using the `check_contracts` from that module as a… what? 🤔

The syntax `@check_contracts` is called a **decorator**, and is technically a form of syntax that is an *optional part of a function definition* that goes immediately above the function header. We say that the line `@check_contracts` *decorates* the function `max_length`, which means that it adds additional behaviour to the function beyond what is written the function body.

So what is this "additional behaviour" added by `check_contracts`? As you might imagine, it reads the function's type contract and the preconditions written in the function docstring, and causes the function to check these preconditions every time `max_length` is called. Let's see what happens when we run this file in the Python console, and attempt to call `max_length` on an empty set:

```
>>> max_length(set())
Traceback (most recent call last):
  ...  # File location details omitted
AssertionError: max_length precondition "len(strings) > 0" was violated for arguments {strings:
```

Pretty cool! And moreover, because all parameter type annotations are preconditions, `python_ta` will also raise an error if an argument does not match a type annotation. Here's an example of that:

```
>>> max_length(110)
Traceback (most recent call last):
  ...  # File location details omitted
AssertionError: max_length argument 110 did not match type annotation for parameter strings: se
```

We'll be using `check_contracts` for the rest of this course to help us make sure we're sticking to the specifications we've written in our function header and docstrings when we call our functions. Moreover, `check_contracts` checks the return type of each function, so it'll also work as a check when we're implementing our functions to make sure the return value is of the correct type.