4.2 Type Annotations Revisited

Recall our definition of `max_length` from the previous section:

```
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})
```

Let us introduce another issue:

```
>>> max_length({1, 2, 3})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <setcomp>
TypeError: object of type 'int' has no len()
```

Once again, our specification of valid inputs has failed us. The parameter type for `max_length` is `set`, and in Python `set`s can contain the values of many different types. It is not until the function description that we see that the parameter is not just *any* set, but specifically a set of strings. We could make this requirement more explicit by introducing another precondition, but there is a better approach.

The types in a collection

There are three collection types that have seen so far: `set`, `list`, and `dict`. For *homogeneous* collections (where every element has the same data type), the Python programming language gives us a way of specifying the type of the contained values using a familiar syntax. The table below shows these types and some examples; note that `T`, `T1`, etc. are variables that could be replaced with any data type.

Type	Description
`set[T]`	A set whose elements all have type `T`. For example, `{'hi', 'bye'}` has type `set[str]`.
`list[T]`	A list whose elements all have type `T`. For example, `[1, 2, 3]` has type `list[int]`.
`dict[T1, T2]`	A dictionary whose keys all have type `T1` and whose associated values all have type `T2`. For example, `{'a': 1, 'b': 2, 'c': 3}` has type `dict[str, int]`.

Using this more specific type annotation syntax, here is how we can improve the type contract for `max_length`:

```
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})
```

Arbitrary and heterogeneous collections

Though indicating the type of the values inside a collection is useful, it is not always necessary. Sometimes we *want* to be flexible and say that a value must be a specific type of collection, but we don't care what's in the collection, and allow for both homogeneous and heterogeneous collections. With list concatenation, for example, the expression `list1 + list2` will work regardless of what the actual types of the elements in `list1` and `list2` are. In such cases, we will continue using the built-in types `set`, `list`, and `dict`, for these types annotations *without* including square brackets for their contained values.

Applying what we've learned

Let us revisit a function we designed when discussing if statements back in 3.4 If Statements:

```
def get_status_v3(scheduled: int, estimated: int) -> str:
    """Return the flight status for the given scheduled and estimated departure times.

    The times are given as integers between 0 and 23 inclusive, representing
    the hour of the day.

    The status is 'On time', 'Delayed', or 'Cancelled'.
    """
```

How can we improve the specification of this function? Looking at the type annotations we see that, since none are collection types, we cannot make them any more specific than they already are. Next, looking at the docstring we see that there is the potential for some preconditions:[1]

[1] We kept the English description of what the times represent, but moved the Python-checkable part into formal preconditions.

```
def get_status_v3(scheduled: int, estimated: int) -> str:
    """Return the flight status for the given scheduled and estimated departure times.

    The times given represent the hour of the day.

    Preconditions:
      - 0 <= scheduled <= 23
      - 0 <= estimated <= 23
    """
```

Next let us revisit the `count_cancelled` function we designed:

```
def count_cancelled(flights: dict) -> int:
    """Return the number of cancelled flights for the given flight data.

    flights is a dictionary where each key is a flight ID,
    and whose corresponding value is a list of two numbers, where the first is
    the scheduled departure time and the second is the estimated departure time.

    >>> count_cancelled({'AC110': [10, 12], 'AC321': [12, 19], 'AC999': [1, 1]})
    1
    """
    cancelled_flights = {id for id in flights
                         if get_status2(flights[id][0], flights[id][1]) == 'Cancelled'}
    return len(cancelled_flights)
```

Here we can improve the type annotations. The first parameter is not just a `dict`, but a `dict[str, list[int]]`—that is, its keys are strings (the flight IDs), and the corresponding value is a list of integers. Does this type annotation mean that now the documentation describing the dictionary is irrelevant? *No!* While the type annotation gives some insight on the structure of the data, it does not provide domain-specific context, like the fact that the `str` keys represent flight IDs, or that the list values represent scheduled and estimated arrival departure times.

There is one more precondition that we can formalize, though: the length of each list in our dictionary. *Every* list should have length two, which translates naturally into a use of Python's `all` function:

```
def count_cancelled(flights: dict[str, list[int]]) -> int:
    """Return the number of cancelled flights for the given flight data.

    flights is a dictionary where each key is a flight ID,
    and whose corresponding value is a list of two numbers, where the first is
    the scheduled departure time and the second is the estimated departure time.

    Precondition:
      - all(len(flights[k]) == 2 for k in flights)

    >>> count_cancelled({'AC110': [10, 12], 'AC321': [12, 19], 'AC999': [1, 1]})
    1
    """
    cancelled_flights = {id for id in flights
                         if get_status2(flights[id][0], flights[id][1]) == 'Cancelled'}
    return len(cancelled_flights)
```

`Any`: a general type

Consider the following function definition which is often called the *identity function* in mathematical and computing contexts:

```
def identity(x: ...) -> ...:
    """Return the argument that was given.

    >>> identity(3)
    3
    >>> identity('Mario is cooler than David')
    'Mario is cooler than David'
    """
    return x
```

This is such a simple function, yet it poses one complication for us: what should the type contract of this function be? As the doctests illustrate, our `identity` function works on values of any data type; and so it would be misleading to use specific type annotation like `identity(x: int) -> int`.

Luckily, the Python language gives us a tool for expressing the concept of a parameter having any possible type. To do so, we import a special variable called `Any` from a built-in Python module we haven't yet discussed, but whose purpose should be clear from its name: `typing`.[2] Here is how we could complete the header of our `identity` function:

[2] Recall that modules can contain both functions (like `doctest.testmod`) and non-function variables (like `math.pi`). `typing.Any` is an instance of the latter.

```
import typing


def identity(x: typing.Any) -> typing.Any:
    """Return the argument that was given.

    >>> identity(3)
    3
    >>> identity('Mario is cooler than David')
    'Mario is cooler than David'
    """
    return x
```

What the type annotation `typing.Any` signals is that this function can be passed values of any type for the `x` parameter, and similarly that this function can return a value of any type. This type annotation won't come up very frequently in CSC110/111, but it does have its uses, so we wanted to cover it here!

Importing specific variables/functions

The above example illustrates one of the slight annoyances of importing modules: after importing the `typing` module, we have to write its name every time we want to access the `Any` variable defined within it. So, we'll end this section by introducing a new form of import statement, called the **import-from statement**.

```
# import-from statement syntax
from <module> import <name1>, <name2>, ...
```

In our above example, we can replace the import statement

```
import typing
```

with the import-from statement

```
from typing import Any
```

Now, why might we want to do this? When the Python interpreter executes an import-from statement, it makes the names after the `import` keyword available to be used directly in subsequent code, without requiring dot notation. For example, with an import-from statement, we could rewrite our above example as follows:

```
from typing import Any


def identity(x: Any) -> Any:
    """Return the argument that was given.

    >>> identity(3)
    3
    >>> identity('Mario is cooler than David')
    'Mario is cooler than David'
    """
    return x
```

The import statement at the top got a bit longer, but in exchange we could just write `Any` rather than `typing.Any` in the function header. Pretty neat!

You might be wondering whether there's any downside of using import-from statements rather than the plain import statement we learned in 2.5 Importing Python Modules (and if not, why didn't we introduce import-from statements earlier!?). There is one downside of using this syntax, which we will illustrate with a different example in the Python console.

```
>>> from math import sqrt
>>> sqrt(4.0)  # We can now use sqrt instead of math.sqrt
2.0
>>> math.pi   # Let's remind ourselves of what pi is
Traceback (most recent call last):
  ...
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

Yikes—`math` is not defined! *An import-from statement only introduces into scope the names explicitly listed after the `import`, and does not introduce the module itself.* If we wanted to also access `pi`, or `sin`, `cos`, etc. from the `math` module, we'd either need to list them after `sqrt`, or add additional import statements.

So one good rule of thumb to follow here is: use the direct import statement if you want to access several variables/functions from the module, and use an import-from statement if you want to access only one or a few.

Additional reading

- Appendix B.4 `typing`