10.3 Defining Our Own Methods

It is certainly possible to accomplish everything that we would ever want to do with our `Person` class from the previous section by writing top-level functions, and this is the approach we've taken with data classes up to this point. An alternate and commonly-used approach is to define *methods* for a data type, which become part of the interface of that data type. Remember that methods are just functions that belong to a data type—but this "belonging to" is not just a conceptual relationship! Defining and using methods have concrete requirements and consequences in Python, and software engineering in general. When we define a data class and top-level functions, the interface of a data class itself only consists of its attributes; we have to remember to import those functions separately in order to use them. When we define a class with methods, those methods are *always* bundled with the class, and so any instance of the class can use those methods, without needing to import them separately.

Defining a method: an example

We have seen one example of a method definition already: the initializer, `__init__`. More generally, any function that operates on an instance of a class can be converted into a method by doing the following:

- Indent the function so that it is part of the class body, underneath the instance attributes.
- Ensure that the first parameter of the function is an instance of the class, and name this parameter `self`.

For example, suppose we had the following function to increase a person's age:

```
def increase_age(person: Person, years: int) -> None:
    """Add the given number of years to the given person's

    >>> david = Person('David', 'Liu', 100, '40 St. George
    >>> increase_age(david, 10)
    >>> david.age
    110
    """
    person.age = person.age + years
```

We can turn `increase_age` into a `Person` method as follows:

```
class Person:
    """A custom data type that represents data for a person."""
    given_name: str
    family_name: str
    age: int
    address: str

    def __init__(self, given_name: str, family_name: str, age: int, address: str) -> None:
        """Initialize a new Person object."""
        self.given_name = given_name
        self.family_name = family_name
        self.age = age
        self.address = address

    def increase_age(self, years: int) -> None:
        """Add the given number of years to this person's age.

        >>> david = Person('David', 'Liu', 100, '40 St. George Street')
        >>> Person.increase_age(david, 10)
        >>> david.age
        110
        """
        self.age = self.age + years
```

Notice that we now use parameter `self` (without a type annotation) to access instance attributes, just as we did in the initializer. In our function docstring, the phrase "the given person" changes to "this person",[1] and our doctest example changes the call to `increase_age` to `Person.increase_age`.

[1] We typically use the word "this" in a method docstring to refer to the object instance that `self` refers to. In fact, some other programming languages also use `this` instead of `self` as a variable or keyword to refer to this object in code.

Defining a method: general syntax

In general, Python uses the following syntax for defining a method:

```
class <ClassName>:
    """..."""
    <instance attributes/types omitted>

    def <method_name>(self, <param>: <type>, ...) -> <retu
        """Method docstring"""
        <statement>
        ...
```

Shortcut syntax for method calls

Now that we are starting to define our own custom classes and methods, we are ready to see a shorthand for calling methods in Python. Let's take a look at the method call from our doctest above:

```
>>> Person.increase_age(david, 10)
```

This uses dot notation to access the `increase_age` method of the `Person` class, calling it with the two arguments `david` and `10`, which get assigned to parameters `self` and `years`, respectively.

The alternate form for calling the `increase_age` method is to use dot notation *with the `Person` instance before the dot*:

```
>>> david.increase_age(10)
```

When we call `david.increase_age(10)`, the Python interpreter does the following:

1. It looks up the type of `david`, which is the `Person` class.
2. It looks up the `increase_age` method of the `Person` class.
3. It calls `Person.increase_age` on `david` and `10`. In other words, the interpreter *automatically* passes the value to the left of the dot (in this case, the object `david` refers to) as the method's first parameter `self`.

This works not just for our custom class `Person`, but all built-in data types as well. For example, `list.append(lst, 10)` can be written as `lst.append(10)`, and `str.lower(s)` as simply `s.lower()`. More generally, our method calls of the form

```
type(obj).method(obj, arg1, arg2, ..., argn)
```

can be shortened to

```
obj.method(arg1, arg2, ..., argn)
```

Though we've been using the more explicit "class dot notation" style (`Person.increase_age`) so far in this course, we'll switch over to the "object dot notation" style (`david.increase_age`) starting in this chapter, as this is the much more common style in Python programming. There are two primary reasons why the latter style is standard:

1. It matches other languages with an *object-oriented* style of programming, where the object being operated on is of central importance. It also matches our syntax for instance attribute access.

   Because we read from left to right, every time we use dot notation with the instance object on the left, we are reminded that it is an object we are working with, whether we are accessing a piece of data bundled with that object or performing an operation on that object.

   We read `david.age` as "access `david`'s age" and `david.increase_age(10)` as "increase `david`'s age by 10". In both cases, `david` is the most important object in the expression.

2. Only the "object dot notation" style of method call supports *inheritance*, which is a technical feature of classes that we'll discuss towards the end of this chapter.