Classes and Methods

Contents

You can order print and ebook versions of *Think Python 3e* from Bookshop.org and Amazon.

Python is an **object-oriented language** – that is, it provides features that support object-oriented programming, which has these defining characteristics:

- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact.
- Programs include class and method definitions.

For example, in the previous chapter we defined a `Time` class that corresponds to the way people record the time of day, and we defined functions that correspond to the kinds of things people do with times. But there was no explicit connection between the definition of the `Time` class and the function definitions that follow. We can make the connection explicit by rewriting a function as a **method**, which is defined inside a class definition.

15.1. Defining methods

In the previous chapter we defined a class named `Time` and wrote a function named `print_time` that displays a time of day.

```
class Time:
    """Represents the time of day."""

def print_time(time):
    s = f'{time.hour:02d}:{time.minute:02d}:{time.second:02d}'
    print(s)
```

To make `print_time` a method, all we have to do is move the function definition inside the class definition. Notice the change in indentation.

At the same time, we'll change the name of the parameter from `time` to `self`. This change is not necessary, but it is conventional for the first parameter of a method to be named `self`.

```
class Time:
    """Represents the time of day."""

    def print_time(self):
        s = f'{self.hour:02d}:{self.minute:02d}:{self.second:02d}'
        print(s)
```

To call this method, you have to pass a `Time` object as an argument. Here's the function we'll use to make a `Time` object.

```
def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time
```

And here's a `Time` instance.

```
start = make_time(9, 40, 0)
```

Now there are two ways to call `print_time`. The first (and less common) way is to use function syntax.

```
Time.print_time(start)
```

```
09:40:00
```

In this version, `Time` is the name of the class, `print_time` is the name of the method, and `start` is passed as a parameter. The second (and more idiomatic) way is to use method syntax:

```
start.print_time()
```

```
09:40:00
```

In this version, `start` is the object the method is invoked on, which is called the **receiver**, based on the analogy that invoking a method is like sending a message to an object.

Regardless of the syntax, the behavior of the method is the same. The receiver is assigned to the first parameter, so inside the method, `self` refers to the same object as `start`.

15.2. Another method

Here's the `time_to_int` function from the previous chapter.

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

And here's a version rewritten as a method.

```
%%add_method_to Time

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds
```

The first line uses the special command `add_method_to`, which adds a method to a previously-defined class. This command works in a Jupyter notebook, but it is not part of Python, so it won't work in other environments. Normally, all methods of a class are inside the class definition, so they get defined at the same time as the class. But for this book, it is helpful to define one method at a time.

As in the previous example, the method definition is indented and the name of the parameter is `self`. Other than that, the method is identical to the function. Here's how we invoke it.

```
start.time_to_int()
```

```
34800
```

It is common to say that we "call" a function and "invoke" a method, but they mean the same thing.

15.3. Static methods

As another example, let's consider the `int_to_time` function. Here's the version from the previous chapter.

```
def int_to_time(seconds):
    minute, second = divmod(seconds, 60)
    hour, minute = divmod(minute, 60)
    return make_time(hour, minute, second)
```

This function takes `seconds` as a parameter and returns a new `Time` object. If we transform it into a method of the `Time` class, we have to invoke it on a `Time` object. But if we're trying to create a new `Time` object, what are we supposed to invoke it on?

We can solve this chicken-and-egg problem using a **static method**, which is a method that does not require an instance of the class to be invoked. Here's how we rewrite this function as a static method.

```
%%add_method_to Time

    def int_to_time(seconds):
        minute, second = divmod(seconds, 60)
        hour, minute = divmod(minute, 60)
        return make_time(hour, minute, second)
```

Because it is a static method, it does not have `self` as a parameter. To invoke it, we use `Time`, which is the class object.

```
start = Time.int_to_time(34800)
```

The result is a new object that represents 9:40.

```
start.print_time()
```

```
09:40:00
```

Now that we have `Time.from_seconds`, we can use it to write `add_time` as a method. Here's the function from the previous chapter.

```
def add_time(time, hours, minutes, seconds):
    duration = make_time(hours, minutes, seconds)
    seconds = time_to_int(time) + time_to_int(duration)
    return int_to_time(seconds)
```

And here's a version rewritten as a method.

```
%%add_method_to Time

    def add_time(self, hours, minutes, seconds):
        duration = make_time(hours, minutes, seconds)
        seconds = time_to_int(self) + time_to_int(duration)
        return Time.int_to_time(seconds)
```

`add_time` has `self` as a parameter because it is not a static method. It is an ordinary method – also called an **instance method**. To invoke it, we need a `Time` instance.

```
end = start.add_time(1, 32, 0)
print_time(end)
```

```
11:12:00
```

15.4. Comparing Time objects

As one more example, let's write `is_after` as a method. Here's the `is_after` function, which is a solution to an exercise in the previous chapter.

```
def is_after(t1, t2):
    return time_to_int(t1) > time_to_int(t2)
```

And here it is as a method.

```
%%add_method_to Time

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Because we're comparing two objects, and the first parameter is `self`, we'll call the second parameter `other`. To use this method, we have to invoke it on one object and pass the other as an argument.

```
end.is_after(start)
```

```
True
```

One nice thing about this syntax is that it almost reads like a question, "`end` is after `start`?"

15.5. The `__str__` method

When you write a method, you can choose almost any name you want. However, some names have special meanings. For example, if an object has a method named `__str__`, Python uses that method to convert the object to a string. For example, here is a `__str__` method for a time object.

```
%%add_method_to Time

    def __str__(self):
        s = f'{self.hour:02d}:{self.minute:02d}:{self.second:02d}'
        return s
```

This method is similar to `print_time`, from the previous chapter, except that it returns the string rather than printing it.

You can invoke this method in the usual way.

```
end.__str__()
```

```
'11:12:00'
```

But Python can also invoke it for you. If you use the built-in function `str` to convert a `Time` object to a string, Python uses the `__str__` method in the `Time` class.

```
str(end)
```

```
'11:12:00'
```

And it does the same if you print a `Time` object.

```
print(end)
```

```
11:12:00
```

Methods like `__str__` are called **special methods**. You can identify them because their names begin and end with two underscores.

15.6. The init method

The most special of the special methods is `__init__`, so-called because it initializes the attributes of a new object. An `__init__` method for the `Time` class might look like this:

```
%%add_method_to Time

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

Now when we instantiate a `Time` object, Python invokes `__init__`, and passes along the arguments. So we can create an object and initialize the attributes at the same time.

```
time = Time(9, 40, 0)
print(time)
```

```
09:40:00
```

In this example, the parameters are optional, so if you call `Time` with no arguments, you get the default values.

```
time = Time()
print(time)
```

```
00:00:00
```

If you provide one argument, it overrides `hour`:

```
time = Time(9)
print(time)
```

```
09:00:00
```

If you provide two arguments, they override `hour` and `minute`.

```
time = Time(9, 45)
print(time)
```

```
09:45:00
```

And if you provide three arguments, they override all three default values.

When I write a new class, I almost always start by writing `__init__`, which makes it easier to create objects, and `__str__`, which is useful for debugging.

15.7. Operator overloading

By defining other special methods, you can specify the behavior of operators on programmer-defined types. For example, if you define a method named `__add__` for the `Time` class, you can use the `+` operator on Time objects.

Here is an `__add__` method.

```
%%add_method_to Time

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return Time.int_to_time(seconds)
```

We can use it like this.

```
duration = Time(1, 32)
end = start + duration
print(end)
```

```
11:12:00
```

There is a lot happening when we run these three lines of code:

- When we instantiate a `Time` object, the `__init__` method is invoked.
- When we use the `+` operator with a `Time` object, its `__add__` method is invoked.
- And when we print a `Time` object, its `__str__` method is invoked.

Changing the behavior of an operator so that it works with programmer-defined types is called **operator overloading**. For every operator, like `+`, there is a corresponding special method, like `__add__`.

15.8. Debugging

A `Time` object is valid if the values of `minute` and `second` are between `0` and `60` – including `0` but not `60` – and if `hour` is positive. Also, `hour` and `minute` should be integer values, but we might allow `second` to have a fraction part. Requirements like these are called **invariants** because they should always be true. To put it a different way, if they are not true, something has gone wrong.

Writing code to check invariants can help detect errors and find their causes. For example, you might have a method like `is_valid` that takes a Time object and returns `False` if it violates an invariant.

```
%%add_method_to Time

    def is_valid(self):
        if self.hour < 0 or self.minute < 0 or self.second < 0:
            return False
        if self.minute >= 60 or self.second >= 60:
            return False
        if not isinstance(self.hour, int):
            return False
        if not isinstance(self.minute, int):
            return False
        return True
```

Then, at the beginning of each method you can check the arguments to make sure they are valid.

```
%%add_method_to Time

    def is_after(self, other):
        assert self.is_valid(), 'self is not a valid Time'
        assert other.is_valid(), 'self is not a valid Time'
        return self.time_to_int() > other.time_to_int()
```

The `assert` statement evaluates the expression that follows. If the result is `True`, it does nothing; if the result

is `False`, it causes an `AssertionError`. Here's an example.

```
duration = Time(minute=132)
print(duration)
```

```
00:132:00
```

```
start.is_after(duration)
```

```
AssertionError: self is not a valid Time
```

`assert` statements are useful because they distinguish code that deals with normal conditions from code that checks for errors.

15.9. Glossary

object-oriented language: A language that provides features to support object-oriented programming, notably user-defined types.

method: A function that is defined inside a class definition and is invoked on instances of that class.

receiver: The object a method is invoked on.

static method: A method that can be invoked without an object as receiver.

instance method: A method that must be invoked with an object as receiver.

special method: A method that changes the way operators and some functions work with an object.

operator overloading: The process of using special methods to change the way operators with with user-defined types.

invariant: A condition that should always be true during the execution of a program.

15.10. Exercises

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

15.10.1. Ask a virtual assistant

For more information about static methods, ask a virtual assistant:

- "What's the difference between an instance method and a static method?"
- "Why are static methods called static?"

If you ask a virtual assistant to generate a static method, the result will probably begin with `@staticmethod`, which is a "decorator" that indicates that it is a static method. Decorators are not covered in this book, but if you are curious, you can ask a VA for more information.

In this chapter we rewrote several functions as methods. Virtual assistants are generally good at this kind of code transformation. As an example, paste the following function into a VA and ask it, "Rewrite this function as a method of the `Time` class."

```
def subtract_time(t1, t2):
    return time_to_int(t1) - time_to_int(t2)
```

15.10.2. Exercise

In the previous chapter, a series of exercises asked you to write a `Date` class and several functions that work with `Date` objects. Now let's practice rewriting those functions as methods.

1. Write a definition for a `Date` class that represents a date – that is, a year, month, and day of the month.
2. Write an `__init__` method that takes `year`, `month`, and `day` as parameters and assigns the parameters to attributes. Create an object that represents June 22, 1933.
3. Write `__str__` method that uses an f-string to format the attributes and returns the result. If you test it with the `Date` you created, the result should be `1933-06-22`.
4. Write a method called `is_after` that takes two `Date` objects and returns `True` if the first comes after the second. Create a second object that represents September 17, 1933, and check whether it comes after the first object.

Hint: You might find it useful write a method called `to_tuple` that returns a tuple that contains the attributes of a `Date` object in year-month-day order.