Chapter 15
Classes and Methods

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O'Reilly, 2024.

https://allendowney.github.io/ThinkPython/

Contents

1. Defining Methods

- 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.

1. Defining Methods

- 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.

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

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

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

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

- **print_time** is a method of the class **Time**.
- It is not necessary, but it is conventional for the first parameter of a method to be named **self**.

1. Defining Methods

- There are two ways to call print_time.
  - The first (and less common) way is to use function syntax:

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

  - The second (and more idiomatic) way is to use the method syntax:

```
4 start.print_time()
```

    - **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.

```
1 def make_time(hour, minute, second):
2     time = Time()
3     time.hour = hour
4     time.minute = minute
5     time.second = second
6     return time
```

```
1 start = make_time(9, 40, 0)
2
3 Time.print_time(start)
4 start.print_time()
```

2. Another Methods

- All methods of a class are inside the class definition.

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

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

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

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

start.time_to_int()
```

```
34800
```

3. Static Methods

- A **static method** is a method that does not require an instance of the class to be invoked.
  - It does not have **self** as a parameter. To invoke it, we use the class **Time.**
- An ordinary method is also called an **instance method**.

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

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

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

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

```
start = Time.int_to_time(34800)

start.print_time()
```

```
09:40:00
```

3. Static Methods

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

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

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

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

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

```
start = Time.int_to_time(34800)

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

4. Comparing **Time** Objects

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

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

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

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

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

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

```
start = Time.int_to_time(34800)

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

end.is_after(start)
```

5. The __str__ Method

- Methods like **__str__** are called special methods. Python uses that method to **convert the object to a string**.
- When you write a method, you can choose almost any name you want. However, some names have special meanings.

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

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

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

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

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

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

```
start = Time.int_to_time(34800)

print(start.__str__())
print(start)
```

```
09:40:00
09:40:00
```

6. The __init__ Method

- The most special of the special methods is **__init__**, so-called because it initializes the attributes of a new object.

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

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

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

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

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

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

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

- 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.

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

6. The __init__ Method

```
4      def __init__(self, hour=0, minute=0, second=0):
5          self.hour = hour
6          self.minute = minute
7          self.second = second
```

- The parameters are optional
  - If you call **Time** with no arguments, you get the default values:

```
1 time = Time()
2 print(time)
```

```
00:00:00
```

  - If you provide one argument, it overrides **hour**:

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

```
09:00:00
```

  - If you provide two arguments, they override **hour** and **minute.** And if you provide three arguments, they override all three default values.

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

```
09:45:00
```

```
1 time = Time(9, 45, 2)
2 print(time)
```

```
09:45:02
```

6. The __init__ Method

- When we 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.

7. Operating 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.

7. Operating Overloading

```python
class Time:
    """Represents a time of day."""

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

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

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

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

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

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

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

```python
start = Time(7, 30, 0)
duration = Time(1, 30)
end = start + duration
print(end)
```

7. Operating Overloading

Operator	Special Method
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
>>	__rshift__(self, other)
<<	__lshift__(self, other)
&	__and__(self, other)
|	__or__(self, other)
^	__xor__(self, other)

Operator	Special Method
Unary Operators	
–	__neg__(self)
+	__pos__(self)
~	__invert__(self)
Comparison Operators	
<	__lt__(self, other)
>	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

7. Operating Overloading

Operator	Special Method
Assignment Operators	
-=	__isub__(self, other)
+=	__iadd__(self, other)
*=	__imul__(self, other)
/=	__idiv__(self, other)
//=	__ifloordiv__(self, other)
%=	__imod__(self, other)
**=	__ipow__(self, other)

Operator	Special Method
Assignment Operators	
>>=	__irshift__(self, other)
<<=	__ilshift__(self, other)
&=	__iand__(self, other)
|=	__ior__(self, other)
^=	__ixor__(self, other)