Although we are using some of Python's object-oriented features, the programs from the last two chapters are not really object-oriented because they don't represent the relationships between programmer-defined types and the functions that operate on them. The next step is to transform those functions into methods that make the relationships explicit.

17.1 Object-oriented features

Python is an object-oriented programming language, which means that it provides features that support object-oriented programming, which has these defining characteristics:
• Programs include class and method definitions.

• 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.

a method is a function that is associated with a particular class. We have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for programmer-defined types.

Methods are semantically the same as functions, but there are two syntactic differences:
• Methods are defined inside a class definition in order to make the relationship between the class and the method explicit.

• The syntax for invoking a method is different from the syntax for calling a function.

17.2 Printing objects

In [1]:
```
class Time:
    """Represents the time of day."""
def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

To call this function, you have to pass a Time object as an argument:

In [2]:
```
start = Time()
start.hour = 9
start.minute = 45
start.second = 00
print_time(start)
```

```
09:45:00
```

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.

In [6]:
```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

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

In [4]:
```
Time.print_time(start)
```

```
09:45:00
```

In this use of dot notation, Time is the name of the class, and print_time is the name of the method. start is passed as a parameter.

In [10]:
```
start = Time()
start.hour = 9
start.minute = 45
start.second = 00
```

The second (and more concise) way is to use method syntax:

In [11]:
```
start.print_time()
```

```
09:45:00
```

In this use of dot notation, print_time is the name of the method (again), and start is the object the method is invoked on, which is called the subject. Just as the subject of a sentence is what the sentence is about, the subject of a method invocation is what the method is about.

Inside the method, the subject is assigned to the first parameter, so in this case start is assigned to time.

By convention, the first parameter of a method is called self, so it would be more common to write print_time like this:

In [12]:
```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
```

The reason for this convention is an implicit metaphor:
• The syntax for a function call, print_time(start), suggests that the function is the active agent. It says something like, “Hey print_time! Here’s an object for you to print.”
• In object-oriented programming, the objects are the active agents. A method invocation like start.print_time() says “Hey start! Please print yourself.”

17.3 Another example

In [45]:
```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds
    def int_to_time(self,seconds):
        minutes, self.second = divmod(seconds, 60)
        self.hour, self.minute = divmod(minutes, 60)
    def increment(self, seconds):
        seconds += self.time_to_int()
        minutes, self.second = divmod(seconds, 60)
        self.hour, self.minute = divmod(minutes, 60)
        return self
```

In [46]:
```
start = Time()
start.hour = 9
start.minute = 45
start.second = 00
start.print_time()
```

```
09:45:00
```

In [47]:
```
end = start.increment(1337)
```

In [48]:
```
end.print_time()
```

```
10:07:17
```

The subject, start, gets assigned to the first parameter, self. The argument, 1337, gets assigned to the second parameter, seconds.
This mechanism can be confusing, especially if you make an error. For example, if you invoke increment with two arguments, you get:

In [49]:
```
end = start.increment(1337, 460)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-49-a8fd8b8bdfbc> in <module>
----> 1 end = start.increment(1337, 460)

TypeError: increment() takes 2 positional arguments but 3 were given
```

The error message is initially confusing, because there are only two arguments in parentheses. But the subject is also considered an argument, so all together that's three.

By the way, a positional argument is an argument that doesn't have a parameter name; that is, it is not a keyword argument. In this function call:
sketch(parrot, cage, dead=True)

parrot and cage are positional, and dead is a keyword argument.

17.4 A more complicated example

Rewriting is_after (from Section 16.1) is slightly more complicated because it takes two Time objects as parameters. In this case it is conventional to name the first parameter self and the second parameter other:

In [50]:
```
class Time:
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds
    def int_to_time(self,seconds):
        minutes, self.second = divmod(seconds, 60)
        self.hour, self.minute = divmod(minutes, 60)
    def increment(self, seconds):
        seconds += self.time_to_int()
        minutes, self.second = divmod(seconds, 60)
        self.hour, self.minute = divmod(minutes, 60)
        return self
    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

In [51]:
```
start = Time()
start.hour = 9
start.minute = 45
start.second = 00
start.print_time()
```

```
09:45:00
```

In [54]:
```
end = start.increment(1337)
```

In [55]:
```
end.is_after(start)
```

Out[55]:
```
False
```

17.5 The init method

The init method (short for “initialization”) is a special method that gets invoked when an object is instantiated. Its full name is __init__ (two underscore characters, followed by init, and then two more underscores). An init method for the Time class might look like this:

In [56]:
```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds
    def int_to_time(self,seconds):
        minutes, self.second = divmod(seconds, 60)
        self.hour, self.minute = divmod(minutes, 60)
    def increment(self, seconds):
        seconds += self.time_to_int()
        minutes, self.second = divmod(seconds, 60)
        self.hour, self.minute = divmod(minutes, 60)
        return self
    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

The parameters are optional, so if you call Time with no arguments, you get the default values.

In [57]:
```
time = Time()
time.print_time()
```

```
00:00:00
```

If you provide one argument, it overrides hour:

In [58]:
```
time = Time (9)
time.print_time()
```

```
09:00:00
```

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

In [59]:
```
time = Time(9, 45)
time.print_time()
```

```
09:45:00
```

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

17.6 The **str** method

__str__ is a special method, like __init__, that is supposed to return a string representation of an object.

In [60]:
```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
    def print_time(self):
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second))
    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds
    def int_to_time(self,seconds):
        minutes, self.second = divmod(seconds, 60)
        self.hour, self.minute = divmod(minutes, 60)
    def increment(self, seconds):
        seconds += self.time_to_int()
        minutes, self.second = divmod(seconds, 60)
        self.hour, self.minute = divmod(minutes, 60)
        return self
    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

In [61]:
```
time = Time(9, 45)
```

In [62]:
```
print(time)
```

```
09:45:00
```

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

17.7 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 what the definition might look like:

In [63]:
```
# add it in time class
def __add__(self, other):
    seconds = self.time_to_int() + other.time_to_int()
    return int_to_time(seconds)
```

And here is how you could use it:

> > > start = Time(9, 45)
> > > duration = Time(1, 35)
> > > print(start + duration)
> > > 11:20:00

When you apply the + operator to Time objects, Python invokes __add__. When you print the result, Python invokes __str__. So there is a lot happening behind the scenes!

Changing the behavior of an operator so that it works with programmer-defined types is called operator overloading. For every operator in Python there is a corresponding special method, like $\_\_\mathrm{add}\_\_$.

17.8 Type-based dispatch

In the previous section we added two Time objects, but you also might want to add an integer to a Time object. The following is a version of **add** that checks the type of other and invokes either add_time or increment:

In [64]:
```
# inside class Time:
def __add__(self, other):
    if isinstance(other, Time):
        return self.add_time(other)
    else:
        return self.increment(other)
    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

The built-in function isinstance takes a value and a class object, and returns True if the value is an instance of the class.

If other is a Time object, $\_\_\mathrm{add}\_\_$ invokes add_time. Otherwise it assumes that the parameter is a number and invokes increment. This operation is called a type-based dispatch because it dispatches the computation to different methods based on the type of the arguments.

In [65]:
```
# >>> start = Time(9, 45)
# >>> duration = Time(1, 35)
# >>> print(start + duration)
# #11:20:00
# >>> print(start + 1337)
# #10:07:17
```

Unfortunately, this implementation of addition is not commutative. If the integer is the first operand, you get

> > > print(1337 + start)

TypeError: unsupported operand type(s) for +: 'int' and 'instance' The problem is, instead of asking the Time object to add an integer, Python is asking an integer to add a Time object, and it doesn't know how. But there is a clever solution for this problem: the special method $\_\_\mathrm{radd}\_\_$, which stands for "right-side add". This method is invoked when a Time object appears on the right side of the + operator. Here's the definition:

In [66]:
```
# inside class Time:
def __radd__(self, other):
    return self.__add__(other)
```

In [67]:
```
# And here’s how it’s used:
# >>> print(1337 + start)
# 10:07:17
```

17.9 Polymorphism

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always necessary. Often you can avoid it by writing functions that work correctly for arguments with different types.
Many of the functions we wrote for strings also work for other sequence types. For example, in Section 11.2 we used histogram to count the number of times each letter appears in a word.

In [68]:
```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

This function also works for lists, tuples, and even dictionaries, as long as the elements of s are hashable, so they can be used as keys in d.

In [69]:
```
t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
histogram(t)
```

Out[69]:
```
{'spam': 4, 'egg': 1, 'bacon': 1}
```

Functions that work with several types are called polymorphic. Polymorphism can facilitate code reuse. For example, the built-in function sum, which adds the elements of a sequence, works as long as the elements of the sequence support addition.

In [70]:
```
sum([1,2,4]+[2,3,5])
```

Out[70]:
```
17
```

In [73]:
```
sum((1,2))
```

Out[73]:
```
3
```

Since Time objects provide an add method, they work with sum:

In [77]:
```
# t1 = Time(7, 43)
# t2 = Time(7, 41)
# t3 = Time(7, 37)
# total = sum([t1, t2, t3])
# print(total)
```

In general, if all of the operations inside a function work with a given type, the function works with that type.

The best kind of polymorphism is the unintentional kind, where you discover that a function you already wrote can be applied to a type you never planned for.

If you are not sure whether an object has a particular attribute, you can use the built-in function hasattr (see Section 15.7).

Another way to access attributes is the built-in function vars, which takes an object and returns a dictionary that maps from attribute names (as strings) to their values:

In [80]:
```
class Point:
    """Represents a point in 2-D space."""
p = Point()
p.x = 3
p.y = 4
vars(p)
```

Out[80]:
```
{'x': 3, 'y': 4}
```

For purposes of debugging, you might find it useful to keep this function handy:

In [81]:
```
def print_attributes(obj):
    for attr in vars(obj):
        print(attr, getattr(obj, attr))
```

In [82]:
```
print_attributes(p)
```

```
x 3
y 4
```

print_attributes traverses the dictionary and prints each attribute name and its corresponding value.
The built-in function getattr takes an object and an attribute name (as a string) and returns the attribute’s value.

17.11 Interface and implementation

One of the goals of object-oriented design is to make software more maintainable, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements. A design principle that helps achieve that goal is to keep interfaces separate from implementations. For objects, that means that the methods a class provides should not depend on how the attributes are represented. For example, in this chapter we developed a class that represents a time of day. Methods provided by this class include time_to_int, is_after, and add_time. We could implement those methods in several ways. The details of the implementation depend on how we represent time. In this chapter, the attributes of a Time object are hour, minute, and second. As an alternative, we could replace these attributes with a single integer representing the number of seconds since midnight. This implementation would make some methods, like is_after, easier to write, but it makes other methods harder. After you deploy a new class, you might discover a better implementation. If other parts of the program are using your class, it might be time-consuming and error-prone to change the interface. But if you designed the interface carefully, you can change the implementation without changing the interface, which means that other parts of the program don’t have to change.

In [ ]: