Warning: the HTML version of this document is generated from Latex and may contain translation errors. In particular, some mathematical expressions are not translated correctly.

Chapter 14

Classes and methods

14.1 Object-oriented features

Python is an **object-oriented programming language**, which means that it provides features that support **object-oriented programming**.

It is not easy to define object-oriented programming, but we have already seen some of its characteristics:

- Programs are made up of object definitions and function definitions, and most of the computation is expressed in terms of operations on objects.
- Each object definition corresponds to some object or concept in the real world, and the functions that operate on that object correspond to the ways real-world objects interact.

For example, the `Time` class defined in Chapter 13 corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the `Point` and `Rectangle` classes correspond to the mathematical concepts of a point and a rectangle.

So far, we have not taken advantage of the features Python provides to support object-oriented programming. Strictly speaking, these features are not necessary. For the most part, they provide an alternative syntax for things we have already done, but in many cases, the alternative is more concise and more accurately conveys the structure of the program.

For example, in the `Time` program, there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every function takes at least one `Time` object as an argument.

This observation is the motivation for **methods**. We have already seen some methods, such as `keys` and `values`, which were invoked on dictionaries. Each method is associated with a class and is intended to be invoked on instances of that class.

Methods are just like functions, with two 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.

In the next few sections, we will take the functions from the previous two chapters and transform them into methods. This transformation is purely mechanical; you can do it simply by following a sequence of steps. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

14.2 printTime

In Chapter 13, we defined a class named `Time` and you wrote a function named `printTime`, which should have looked something like this:

```
class Time:
  pass

def printTime(time):
  print str(time.hours) + ":" + \
        str(time.minutes) + ":" + \
        str(time.seconds)
```

To call this function, we passed a `Time` object as an argument:

```
>>> currentTime = Time()
>>> currentTime.hours = 9
>>> currentTime.minutes = 14
>>> currentTime.seconds = 30
>>> printTime(currentTime)
```

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

```
class Time:
  def printTime(time):
    print str(time.hours) + ":" +  \
          str(time.minutes) + ":" +  \
          str(time.seconds)
```

Now we can invoke `printTime` using dot notation.

```
>>> currentTime.printTime()
```

As usual, the object on which the method is invoked appears before the dot and the name of the method appears after the dot.

The object on which the method is invoked is assigned to the first parameter, so in this case `currentTime` is assigned to the parameter `time`.

By convention, the first parameter of a method is called `self`. The reason for this is a little convoluted, but it is based on a useful metaphor.

The syntax for a function call, `printTime(currentTime)`, suggests that the function is the active agent. It says something like, "Hey `printTime`! Here's an object for you to print."

In object-oriented programming, the objects are the active agents. An invocation like `currentTime.printTime()` says "Hey `currentTime`! Please print yourself!"

This change in perspective might be more polite, but it is not obvious that it

is useful. In the examples we have seen so far, it may not be. But sometimes shifting responsibility from the functions onto the objects makes it possible to write more versatile functions, and makes it easier to maintain and reuse code.

14.3 Another example

Let's convert `increment` (from Section 13.3) to a method. To save space, we will leave out previously defined methods, but you should keep them in your version:

```
class Time:
  #previous method definitions here...

  def increment(self, seconds):
    self.seconds = seconds + self.seconds

    while self.seconds >= 60:
      self.seconds = self.seconds - 60
      self.minutes = self.minutes + 1

    while self.minutes >= 60:
      self.minutes = self.minutes - 60
      self.hours = self.hours + 1
```

The transformation is purely mechanical — we move the method definition into the class definition and change the name of the first parameter.

Now we can invoke `increment` as a method.

```
currentTime.increment(500)
```

Again, the object on which the method is invoked gets assigned to the first parameter, `self`. The second parameter, `seconds` gets the value `500`.

> *As an exercise, convert `convertToSeconds` (from Section 13.5) to a method in the `Time` class.*

14.4 A more complicated example

The `after` function is slightly more complicated because it operates on two `Time` objects, not just one. We can only convert one of the parameters to `self`; the other stays the same:

```
class Time:
  #previous method definitions here...

  def after(self, time2):
    if self.hour > time2.hour:
      return 1
    if self.hour < time2.hour:
      return 0

    if self.minute > time2.minute:
      return 1
    if self.minute < time2.minute:
      return 0

    if self.second > time2.second:
      return 1
    return 0
```

We invoke this method on one object and pass the other as an argument:

```
if doneTime.after(currentTime):
```

```
  print "The bread is not done yet."
```

You can almost read the invocation like English: "If the done-time is after the current-time, then..."

14.5 Optional arguments

We have seen built-in functions that take a variable number of arguments. For example, `string.find` can take two, three, or four arguments.

It is possible to write user-defined functions with optional argument lists. For example, we can upgrade our own version of `find` to do the same thing as `string.find`.

This is the original version from Section 7.7:

```
def find(str, ch):
  index = 0
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

This is the new and improved version:

```
def find(str, ch, start=0):
  index = start
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```

The third parameter, `start`, is optional because a default value, `0`, is provided. If we invoke `find` with only two arguments, it uses the default value and starts from the beginning of the string:

```
>>> find("apple", "p")
1
```

If we provide a third argument, it **overrides** the default:

```
>>> find("apple", "p", 2)
2
>>> find("apple", "p", 3)
-1
```

> *As an exercise, add a fourth parameter, `end`, that specifies where to stop looking.*
>
> *Warning: This exercise is a bit tricky. The default value of `end` should be `len(str)`, but that doesn't work. The default values are evaluated when the function is defined, not when it is called. When `find` is defined, `str` doesn't exist yet, so you can't find its length.*

14.6 The initialization method

The **initialization method** is a special method that is invoked when an object is created. The name of this method is `__init__` (two underscore characters, followed by `init`, and then two more underscores). An

initialization method for the `Time` class looks like this:

```
class Time:
  def __init__(self, hours=0, minutes=0, seconds=0):
    self.hours = hours
    self.minutes = minutes
    self.seconds = seconds
```

There is no conflict between the attribute `self.hours` and the parameter `hours`. Dot notation specifies which variable we are referring to.

When we invoke the `Time` constructor, the arguments we provide are passed along to `init`:

```
>>> currentTime = Time(9, 14, 30)
>>> currentTime.printTime()
9:14:30
```

Because the arguments are optional, we can omit them:

```
>>> currentTime = Time()
>>> currentTime.printTime()
0:0:0
```

Or provide only the first:

```
>>> currentTime = Time (9)
>>> currentTime.printTime()
9:0:0
```

Or the first two:

```
>>> currentTime = Time (9, 14)
>>> currentTime.printTime()
9:14:0
```

Finally, we can make assignments to a subset of the parameters by naming them explicitly:

```
>>> currentTime = Time(seconds = 30, hours = 9)
>>> currentTime.printTime()
9:0:30
```

14.7 Points revisited

Let's rewrite the `Point` class from Section 12.1 in a more object-oriented style:

```
class Point:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

The initialization method takes `x` and `y` values as optional parameters; the default for either parameter is 0.

The next method, `__str__`, returns a string representation of a `Point` object. If a class provides a method named `__str__`, it overrides the default behavior of the Python built-in `str` function.

```
>>> p = Point(3, 4)
>>> str(p)
```

```
'(3, 4)'
```

Printing a `Point` object implicitly invokes `__str__` on the object, so defining `__str__` also changes the behavior of `print`:

```
>>> p = Point(3, 4)
>>> print p
(3, 4)
```

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

14.8 Operator overloading

Some languages make it possible to change the definition of the built-in operators when they are applied to user-defined types. This feature is called **operator overloading**. It is especially useful when defining new mathematical types.

For example, to override the addition operator `+`, we provide a method named `__add__`:

```
class Point:
  # previously defined methods here...

  def __add__(self, other):
    return Point(self.x + other.x, self.y + other.y)
```

As usual, the first parameter is the object on which the method is invoked. The second parameter is conveniently named `other` to distinguish it from `self`. To add two `Point`s, we create and return a new `Point` that contains the sum of the `x` coordinates and the sum of the `y` coordinates.

Now, when we apply the `+` operator to `Point` objects, Python invokes `__add__`:

```
>>>   p1 = Point(3, 4)
>>>   p2 = Point(5, 7)
>>>   p3 = p1 + p2
>>>   print p3
(8, 11)
```

The expression `p1 + p2` is equivalent to `p1.__add__(p2)`, but obviously more elegant.

> *As an exercise, add a method `__sub__(self, other)` that overloads the subtraction operator, and try it out.*

There are several ways to override the behavior of the multiplication operator: by defining a method named `__mul__`, or `__rmul__`, or both.

If the left operand of `*` is a `Point`, Python invokes `__mul__`, which assumes that the other operand is also a `Point`. It computes the **dot product** of the two points, defined according to the rules of linear algebra:

```
def __mul__(self, other):
  return self.x * other.x + self.y * other.y
```

If the left operand of `*` is a primitive type and the right operand is a `Point`, Python invokes `__rmul__`, which performs **scalar multiplication**:

```
def __rmul__(self, other):
  return Point(other * self.x,  other * self.y)
```

The result is a new `Point` whose coordinates are a multiple of the original coordinates. If `other` is a type that cannot be multiplied by a floating-point number, then `__rmul__` will yield an error.

This example demonstrates both kinds of multiplication:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
```

What happens if we try to evaluate `p2 * 2`? Since the first operand is a `Point`, Python invokes `__mul__` with `2` as the second argument. Inside `__mul__`, the program tries to access the `x` coordinate of `other`, which fails because an integer has no attributes:

```
>>> print p2 * 2
AttributeError: 'int' object has no attribute 'x'
```

Unfortunately, the error message is a bit opaque. This example demonstrates some of the difficulties of object-oriented programming. Sometimes it is hard enough just to figure out what code is running.

For a more complete example of operator overloading, see Appendix B.

14.9 Polymorphism

Most of the methods we have written only work for a specific type. When you create a new object, you write methods that operate on that type.

But there are certain operations that you will want to apply to many types, such as the arithmetic operations in the previous sections. If many types support the same set of operations, you can write functions that work on any of those types.

For example, the `multadd` operation (which is common in linear algebra) takes three arguments; it multiplies the first two and then adds the third. We can write it in Python like this:

```
def multadd (x, y, z):
  return x * y + z
```

This method will work for any values of `x` and `y` that can be multiplied and for any value of `z` that can be added to the product.

We can invoke it with numeric values:

```
>>> multadd (3, 2, 1)
7
```

Or with `Point`s:

```
>>> p1 = Point(3, 4)
>>> p2 = Point(5, 7)
>>> print multadd (2, p1, p2)
(11, 15)
>>> print multadd (p1, p2, 1)
```

```
44
```

In the first case, the `Point` is multiplied by a scalar and then added to another `Point`. In the second case, the dot product yields a numeric value, so the third argument also has to be a numeric value.

A function like this that can take arguments with different types is called **polymorphic**.

As another example, consider the method `frontAndBack`, which prints a list twice, forward and backward:

```
def frontAndBack(front):
  import copy
  back = copy.copy(front)
  back.reverse()
  print str(front) + str(back)
```

Because the `reverse` method is a modifier, we make a copy of the list before reversing it. That way, this method doesn't modify the list it gets as an argument.

Here's an example that applies `frontAndBack` to a list:

```
>>>   myList = [1, 2, 3, 4]
>>>   frontAndBack(myList)
[1, 2, 3, 4][4, 3, 2, 1]
```

Of course, we intended to apply this function to lists, so it is not surprising that it works. What would be surprising is if we could apply it to a `Point`.

To determine whether a function can be applied to a new type, we apply the fundamental rule of polymorphism:

> **If all of the operations inside the function can be applied to the type, the function can be applied to the type.**

The operations in the method include `copy`, `reverse`, and `print`.

`copy` works on any object, and we have already written a `__str__` method for `Point`s, so all we need is a `reverse` method in the `Point` class:

```
def reverse(self):
  self.x , self.y = self.y, self.x
```

Then we can pass `Point`s to `frontAndBack`:

```
>>>   p = Point(3, 4)
>>>   frontAndBack(p)
(3, 4)(4, 3)
```

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

14.10 Glossary

object-oriented language
: A language that provides features, such as user-defined classes and inheritance, that facilitate object-oriented programming.

object-oriented programming
: A style of programming in which data and the operations that manipulate it are organized into classes and methods.

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

override
: To replace a default. Examples include replacing a default value with a particular argument and replacing a default method by providing a new method with the same name.

initialization method
: A special method that is invoked automatically when a new object is created and that initializes the object's attributes.

operator overloading
: Extending built-in operators (`+`, `-`, `*`, `>`, `<`, etc.) so that they work with user-defined types.

dot product
: An operation defined in linear algebra that multiplies two `Point`s and yields a numeric value.

scalar multiplication
: An operation defined in linear algebra that multiplies each of the coordinates of a `Point` by a numeric value.

polymorphic
: A function that can operate on more than one type. If all the operations in a function can be applied to a type, then the function can be applied to a type.

Warning: the HTML version of this document is generated from Latex and may contain translation errors. In particular, some mathematical expressions are not translated correctly.