Part III

Object-Oriented Programming

Chapter 12

Classes and objects

12.1 User-defined types

We have used many of Python's built-in types; now we are going to define a new type. As an example, we will create a type called `Point` that represents a point in two-dimensional space.

In mathematical notation, points are often written in parentheses with a comma separating the coordinates. For example, $(0,0)$ represents the origin, and (x,y) represents the point x units to the right and y units up from the origin.

There are several ways we might represent points in Python:

- We could store the coordinates separately in two variables, `x` and `y`.
- We could store the coordinates as elements in a list or tuple.
- We could create a new type to represent points as objects.

Creating a new type is (a little) more complicated than the other options, but it has advantages that will be apparent soon.

A user-defined type is also called a **class**. A class definition looks like this:

```
class Point:
    """represents a point in 2-D space"""
```

This header indicates that the new class is called `Point`. The body is a docstring that explains what the class is for. You can define variables and functions inside a class definition, but we will get back to that later.

Defining a class named `Point` creates a class object, also named `Point`.

```
>>> print Point
__main__.Point
>>> type(Point)
<type 'classobj'>
```

Because `Point` is defined at the top level, its "full name" is `__main__.Point`.

The class object is like a factory for creating objects. To create a Point, you call `Point` as if it were a function.

```
>>> blank = Point()
>>> print blank
<__main__.Point instance at 0xb7e9d3ac>
```

The return value is a reference to a Point object, which we assign to `blank`. Creating a new object is called **instantiation**, and the object is an **instance** of the class.

12.2 Attributes

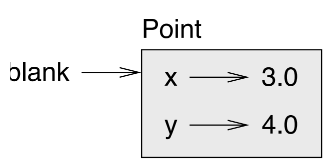
You can assign values to an instance using dot notation:

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

This syntax is similar to the syntax for selecting a variable from a module, such as `math.pi` or `string.uppercase`. In this case, though, we are assigning values to named elements of an object. These elements are called **attributes**.

As a noun, "AT-trib-ute" is pronounced with emphasis on the first syllable, as opposed to "a-TRIB-ute," which is a verb.

The following diagram shows the result of these assignments. A state diagram that shows an object and its attributes is called an **object diagram**:



The variable `blank` refers to a Point object, which contains two attributes. Each attribute refers to a floating-point number.

We can read the value of an attribute using the same syntax:

```
>>> print blank.y
4.0
>>> x = blank.x
>>> print x
3.0
```

The expression blank.x means, "Go to the object blank refers to and get the value of x." In this case, we assign that value to a variable named x. There is no conflict between the variable x and the attribute x.

You can use dot notation as part of any expression. For example:

```
>>> print '(%g, %g)' % (blank.x, blank.y)
(3.0, 4.0)
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> print distance
5.0
```

You can pass an instance as an argument in the usual way. For example:

```
def print_point(p):
    print '(%g, %g)' % (p.x, p.y)
```

print_point takes a point as an argument and displays it in mathematical notation. To invoke it, you can pass blank as an argument:

```
>>> print_point(blank)
(3.0, 4.0)
```

Inside the function, p is an alias for blank, so if the function modifies p, blank changes.

Exercise 12.1. *Write a function called* distance *that it takes two Points as arguments and returns the distance between them.*

12.3 Rectangles

Sometimes it is obvious what the attributes of an object should be, but other times you have to make decisions. For example, imagine you are designing a class to represent rectangles. What attributes would you use to specify the location and size of a rectangle? You can ignore angle; to keep things simple, assume that the rectangle is either vertical or horizontal.

There are at least two possibilities:

- You could specify one corner of the rectangle (or the center), the width, and the height.
- You could specify two opposing corners.

At this point it is hard to say whether either is better than the other, so we'll implement the first one, just as an example.

Here is the class definition:

```
class Rectangle:
    """represent a rectangle.
       attributes: width, height, corner.
    """
```

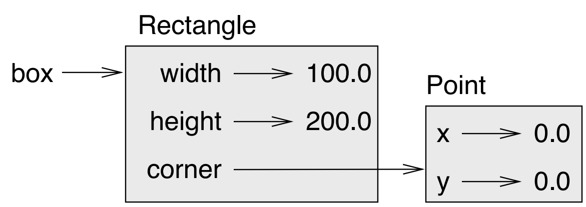
The docstring lists the attribute names. `width` and `height` are numbers; `corner` is a Point object that specifies the lower-left corner.

To represent a rectangle, you have to instantiate a Rectangle object and assign values to the attributes:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

The expression `box.corner.x` means, "Go to the object `box` refers to and select the attribute named `corner`; then go to that object and select the attribute named `x`."

The figure shows the state of this object:



12.4 Instances as return values

Functions can return instances. For example, `find_center` takes a `Rectangle` as an argument and returns a `Point` that contains the coordinates of the center of the `Rectangle`:

```
def find_center(box):
    p = Point()
    p.x = box.corner.x + box.width/2.0
    p.y = box.corner.y + box.height/2.0
    return p
```

Here is an example that passes `box` as an argument and assign the resulting Point to `center`:

```
>>> center = find_center(box)
>>> print_point(center)
(50.0, 100.0)
```

12.5 Objects are mutable

We can change the state of an object by making an assignment to one of its attributes. For example, to change the size of a rectangle without changing its position, you can modify the values of `width` and `height`:

```
box.width = box.width + 50
box.height = box.width + 100
```

You can also write functions that modify objects. For example, `grow_rectangle` takes a Rectangle object and two numbers, `dwidth` and `dheight`, and adds the numbers to the width and height of the rectangle:

```
def grow_rectangle(rect, dwidth, dheight) :
    rect.width += dwidth
    rect.height += dheight
```

Here is an example that demonstrates the effect:

```
>>> print box.width
100.0
>>> print box.height
200.0
>>> grow_rectangle(box, 50, 100)
>>> print box.width
150.0
>>> print box.height
300.0
```

Inside the function, `rect` is an alias for `box`, so if the function modifies `rect`, `box` changes.

Exercise 12.2. *Write a function named* `move_rectangle` *that takes a Rectangle and two numbers named* `dx` *and* `dy`*. It should change the location of the rectangle by adding* `dx` *to the* `x` *coordinate of* `corner` *and adding* `dy` *to the* `y` *coordinate of* `corner`.

12.6 Copying

Aliasing can make a program difficult to read because changes made in one place might have unexpected effects in another place. It is hard to keep track of all the variables that might refer to a given object.

Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object:

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0
```

```
>>> import copy
>>> p2 = copy.copy(p1)
```

p1 and p2 contain the same data, but they are not the same Point.

```
>>> print_point(p1)
(3.0, 4.0)
>>> print_point(p2)
(3.0, 4.0)
>>> p1 is p2
False
>>> p1 == p2
False
```

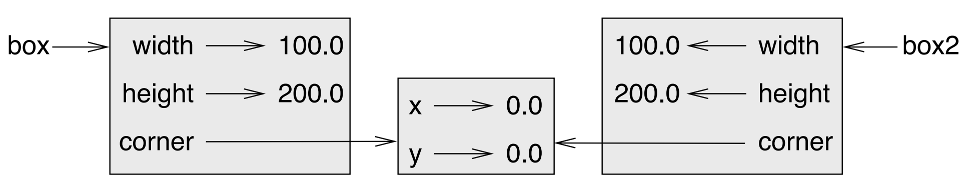
The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same as the is operator; it checks object identity, not object equivalence.

This behavior can be changed, so for many objects defined in Python modules, the == operator checks equivalence (in whatever sense is appropriate). But the default is to check identity.

If you use copy.copy to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

Here is what the object diagram looks like:



This operation is called a **shallow copy** because it copies the object and any references it contains, but not the embedded objects.

For most applications, this is not what you want. In this example, invoking grow_rectangle on one of the Rectangles would not affect the other, but invoking `move_rectangle` on either would affect both! This behavior is confusing and error-prone.

Fortunately, the `copy` module contains a method named `deepcopy` that copies not only the object but also the objects it refers to, and the objects *they* refer to, and so on. You will not be surprised to learn that this operation is called a **deep copy**.

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

`box3` and `box` are completely separate objects.

Exercise 12.3. *Write a version* `move_rectangle` *that it creates and returns a new Rectangle instead of modifying the old one.*

12.7 Debugging

When you start working with objects, you are likely to encounter some new exceptions. If you try to access an attribute that doesn't exist, you get an `AttributeError`:

```
>>> p = Point(3, 4)
>>> print p.z
AttributeError: Point instance has no attribute 'z'
```

If you are not sure what type an object is, you can ask:

```
>>> type(p)
<type 'instance'>
```

This result tells us that `p` is an object, but not what kind. But all objects have a special attribute named `__class__` that refers to the object's class.

```
>>> print p.__class__
__main__.Point
```

If you are not sure whether an object has a particular attribute, you can use the built-in function `hasattr`:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

The first argument can be any object; the second argument is a *string* that contains the name of the attribute.

Another way to access the attributes of an object is through the special attribute `__dict__`, which is a dictionary that maps from attribute names (as strings) and values:

```
>>> print p.__dict__
{'y': 4, 'x': 3}
```

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

```
def print_attributes(obj):
    for attr in obj.__dict__:
        print attr, getattr(obj, attr)
```

`print_attributes` traverses the items in the object's dictionary print each attrbute 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.

12.8 Glossary

class: A user-defined type. A class definition creates a new class object.

class object: An object that contains information about a user-defined time. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

attribute: One of the named values associated with an object.

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the `copy` function in the `copy` module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the `deepcopy` function in the `copy` module.

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

12.9 Exercises

Chapter 13

Classes and functions

13.1 Time

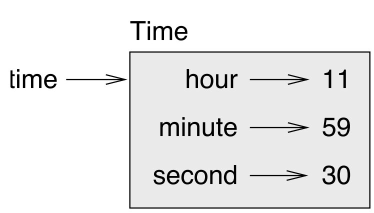
As another example of a user-defined type, we'll define a class called `Time` that records the time of day. The class definition looks like this:

```
class Time:
    """represents the time of day
       attributes: hour, minute, second"""
```

We can create a new `Time` object and assign attributes for hours, minutes, and seconds:

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

The state diagram for the `Time` object looks like this:



Exercise 13.1. *Write a function* `print_time` *that takes a Time object and prints it in the form* `hour:minute:second`*.*
Exercise 13.2. *Write a boolean function* `after` *that takes two Time objects,* `t1` *and* `t2`*, and returns* `True` *if* `t1` *follows* `t2` *chronologically and* `False` *otherwise.*

13.2 Pure functions

In the next few sections, we'll write two versions of a function called add_time, which calculates the sum of two Time objects. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a development plan I'll call **prototype and patch**, which is a way of tackling a complex problem by starting with a simple prototype and incrementally dealing with the complications.

Here is a simple prototype of add_time:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

The function creates a new Time object, initializes its attributes, and returns a reference to the new object. This is called a **pure function** because it does not modify any of the objects passed to it as arguments and it has no side effects, such as displaying a value or getting user input.

To test this function, I'll create two Time objects: start contains the start time of a movie, like *Monty Python and the Holy Grail*, and duration contains the run time of the movie, which is one hour 35 minutes.

add_time figures out when the movie will be done.

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second =  0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

The result, 10:80:00 might not be what you were hoping for. The problem is that this function does not deal with cases where the number of seconds or minutes adds up to more than sixty. When that happens, we have to "carry" the extra seconds into the minute column or the extra minutes into the hour column.

Here's an improved version:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

Although this function is correct, it is starting to get big. We will see a shorter alternative later.

13.3 Modifiers

Sometimes it is useful for a function to modify the objects it gets as parameters. In that case, the changes are visible to the caller. Functions that work this way are called **modifiers**.

`increment`, which adds a given number of seconds to a `Time` object, can be written naturally as a modifier. Here is a rough draft:

```
def increment(time, seconds):
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

The first line performs the basic operation; the remainder deals with the special cases we saw before.

Is this function correct? What happens if the parameter `seconds` is much greater than sixty? In that case, it is not enough to carry once; we have to keep doing it until `time.second` is less than sixty. One solution is to replace the `if` statements with `while` statements. That would make the function correct, but not very efficient.

Exercise 13.3. *Write a correct version of* `increment` *that doesn't contain any loops.*

Anything that can be done with modifiers can also be done with pure functions. In fact, some programming languages only allow pure functions. There is some evidence that programs that use pure functions are faster to develop and less error-prone than programs that use modifiers. But modifiers are convenient at times, and functional programs tend to be less efficient.

In general, I recommend that you write pure functions whenever it is reasonable and resort to modifiers only if there is a compelling advantage. This approach might be called a **functional programming style**.

Exercise 13.4. *Write a "pure" version of* `increment` *that creates and returns a new Time object rather than modifying the parameter.*

13.4 Prototyping versus planning

In this chapter, I demonstrated development plan called "prototype and patch." For each function, I wrote a rough draft that performed the basic calculation and then tested it, correcting flaws along the way.

This approach can be effective, especially if you don't yet have a deep understanding of the problem. But incremental patching can generate code that is unnecessarily complicated—since it deals with many special cases—and unreliable—since it is hard to know if you have found all the errors.

An alternative is **planned development**, in which high-level insight into the problem can make the programming much easier. In this case, the insight is that a Time object is really a three-digit number in base 60! The `second` attribute is the "ones column," the `minute` attribute is the "sixties column," and the `hour` attribute is the "thirty-six hundreds column."

When we wrote `add_time` and `increment`, we were effectively doing addition in base 60, which is why we had to carry from one column to the next.

This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic.

Here is the function that converts Times to integers:

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

And here is the function that converts integers to Times (recall that `divmod` divides the first argument by the second and returns the quotient and remainder as a tuple).

```
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. But once they are debugged, you can use them to rewrite `add_time`:

```
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

This version is shorter than the original, and easier to verify.
Exercise 13.5. *Rewrite* `increment` *using* `time_to_int` *and* `int_to_time`.

In some ways, converting from base 60 to base 10 and back is harder than just dealing with times. Base conversion is more abstract; our intuition for dealing with times is better.

But if we have the insight to treat times as base 60 numbers and make the investment of writing the conversion functions (`time_to_int` and `int_to_time`), we get a program that is shorter, easier to read and debug, and more reliable.

It is also easier to add features later. For example, imagine subtracting two Times to find the duration between them. The naïve approach would be to implement subtraction with borrowing. Using the conversion functions would be easier and more likely to be correct.

Ironically, sometimes making a problem harder (or more general) makes it easier (because there are fewer special cases and fewer opportunities for error).

13.5 Glossary

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

planned development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

pure function: A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are fruitless.

functional programming style: A style of program design in which the majority of functions are pure.

13.6 Exercises

Exercise 13.6. *Write a function called* `mul_time` *that takes a Time object and a number and returns a new Time object that contains the product of the original Time and the number.*

Then use `mul_time` *to write a function that takes a Time object that represents the finishing time in a race, and a number that represents the distance, and returns a Time object that represents the average pace (time per mile).*

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**; a method is a function that is associated with a particular class. For example, we have seen methods for strings, lists, dictionaries and tuples. In this chapter, we will define methods for user-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.

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 print_time

In Chapter 13, we defined a class named Time and in Exercise 13.1, you wrote a function named print_time:

```
class Time:
    """represents the time of day
       attributes: hour, minute, second"""

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:

```
>>> 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.

```
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:

```
>>> 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.

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

```
>>> 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:

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

The reason for this convention is convoluted, but it is based on a useful 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."

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.

Exercise 14.1. *Rewrite* `time_to_int` *(from Section 13.4) as a method. It is probably not appropriate to rewrite* `int_to_time` *as a method; it's not clear what object you would invoke it on!*

14.3 Another example

Here's a version of `increment` (from Section 13.3) rewritten as a method:

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

This version assumes that time_to_int is written as a method, as in Exercise 14.1. Also, note that it is a pure function, not a modifier.

Here's how you would invoke increment:

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> 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:

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes exactly 2 arguments (3 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.

14.4 A more complicated example

after (from Exercise 13.2) 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:

```
# inside class Time:

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

To use this method, you have to invoke it on one object and pass the other as an argument:

```
>>> end.after(start)
True
```

One nice thing about this syntax is that it has the same word order as English, subject-verb-object.

14.5 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:

```
# inside class Time:

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

It is common for the parameters of __init__ to have the same names as the attributes. The statement

```
        self.hour = hour
```

stores the value of the parameter hour as an attribute in the new Time object self.

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

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

If you provide one argument, it overrides hour:

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

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

```
>>> time = Time(9, 45)
>>> time.print_time()
09:45:00
```

And if you provide three arguments, they override all three default values.
Exercise 14.2. *Write an init method for the* Point *class that takes* x *and* y *as optional parameters and assigns them to the corresponding attributes.*

14.6 The str method

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

For example, here is a `str` method for Time objects:

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

When you `print` an object, Python invokes the `str` method:

```
>>> time = Time(9, 45)
>>> 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 almost always useful for debugging.
Exercise 14.3. *Write a* `str` *method for the* `Point` *class. Create a Point object and print it.*

14.7 Operator overloading

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

Here is what the definition might look like:

```
# inside class Time:

    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 quite a lot happening behind the scenes!

Changing the behavior of an operator so that it works with user-defined types is called **operator overloading**. For every operator in Python there is a corresponding special method, like `__add__`.
Exercise 14.4. *Write an* `add` *method for the Point class.*

14.8 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 an alternative version of `__add__` that checks the type of `other` and invokes either `add_time` or `increment`:

```
# 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, `__add__` invokes `add_time`. Otherwise it assumes that the seconds 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.

Here are examples that use the `+` operator with different types:

```
>>> 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 to do that. But there is a clever solution for this problem, the `radd` method, 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:

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

And here's how it's used:

```
>>> print 1337 + start
10:07:17
```

Exercise 14.5. *Write an* add *method for Points that works with either a Point object or a tuple:*

- *If the second operand is a Point, the method should return a new Point whose x coordinate is the sum of the x coordinates of the operands, and likewise for the y coordinates.*
- *If the second operand is a tuple, the method should add the first element of the tuple to the x coordinate and the second element to the y coordinate, and return a new Point with the result.*

14.9 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 will actually work for any kind of sequence. For example, in Section 11.1 we used histogram to count the number of times each letter appears in a word.

```
def histogram(s):
    d = {}
    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.

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

Functions that can work with several types are called **polymorphic**.

Many of the built-in functions are polymorphic. For example, `sum` works with any kind of sequence, as long as the elements support the addition operator.

```
>>> t = [1, 2.0, 42L]
>>> print sum(t)
45.0
```

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

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print total
23:01:00
```

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

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 you never planned for.

14.10 Exercises

Exercise 14.6. *Write a definition for a class named* `Kangaroo` *with the following methods:*

1. *An* `__init__` *method that initializes an attribute named* `pouch_contents` *to an empty list.*
2. *A method named* `put_in_pouch` *that takes an object of any type and adds it to* `pouch_contents`*.*

Test your code by creating two `Kangaroo` *objects, assigning them to variables named* `kanga` *and* `roo`*, and then adding* `roo` *to the contents of* `kanga`*'s pouch.*

14.11 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.

subject: The object a method is invoked on.

operator overloading: Changing the behavior of an operator like + so it works with a user-defined type.

type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.

polymorphic: Pertaining to a function that can work with more than one type.