Classes and objects

15.1 Programmer-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 more complicated than the other options, but it has advantages that will be apparent soon.

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

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

Point

blank → x → 3.0

y → 4.0

Figure 15.1: Object diagram.

The header indicates that the new class is called Point. The body is a docstring that ex-plains what the class is for. You can define variables and methods inside a class definition, but we will get back to that later.

Defining a class named Point creates a **class object**.

```
>>> Point
<class __main__.Point>
```

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()
>>> blank
<__main__.Point object 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.

When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

Every object is an instance of some class, so “object” and “instance” are interchangeable. But in this chapter I use “instance” to indicate that I am talking about a programmer-defined type.

15.2 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.whitespace . 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**; see Figure 15.1.

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

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

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

The expression blank.x means, “Go to the object blank refers to and get the value of x.” In the example, 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:

```
>>> (%g, %g) % (blank.x, blank.y) (3.0, 4.0)

>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> 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.

As an exercise, write a function called distance_between_points that takes two Points as arguments and returns the distance between them.

15.3 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 ig-nore 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:

Figure 15.2: Object diagram.

```
class Rectangle:
"""Represents a rectangle.

attributes: width, height, corner.
"""
```

The docstring lists the attributes: 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.”

Figure 15.2 shows the state of this object. An object that is an attribute of another object is **embedded**.

15.4 Instances as return values

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

```
def find_center(rect):
p = Point()
p.x = rect.corner.x + rect.width/2
p.y = rect.corner.y + rect.height/2
return p
```

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

```
    >>>  center = find_center(box)
    >>>  print_point(center)
(50, 100)
```

15.5 Objects are mutable

You 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.height + 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:

```
>>>  box.width, box.height
(150.0, 300.0)
>>>  grow_rectangle(box, 50, 100)
>>>  box.width, box.height
(200.0, 400.0)
```

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

As an exercise, 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.

15.6 Copying

Aliasing can make a program difficult to read because changes 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,
4)
>>>  print_point(p2) (3,
4)
>>>  p1 is p2
False
```

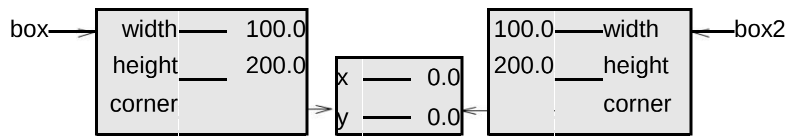

Figure 15.3: Object diagram.

```
>>> 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. That’s because for programmer-defined types, Python doesn’t know what should be considered equivalent. At least, not yet.

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
```

Figure 15.3 shows 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 provides 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.

As an exercise, write a version of move_rectangle that creates and returns a new Rectangle instead of modifying the old one.

15.7 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()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute z
```

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

```
>>> type(p)
<class __main__.Point>
```

You can also use isinstance to check whether an object is an instance of a class:

```
>>> isinstance(p, Point)
True
```

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.

You can also use a try statement to see if the object has the attributes you need:

```
try:
x = p.x
except AttributeError:
x = 0
```

This approach can make it easier to write functions that work with different types; more on that topic is coming up in Section 17.9.

Classes and functions

Now that we know how to create new types, the next step is to write functions that take programmer-defined objects as parameters and return them as results. In this chapter I also present "functional programming style" and two new program development plans.

Code examples from this chapter are available from http://thinkpython2.com/code/ Time1.py. Solutions to the exercises are at http://thinkpython2.com/code/Time1_soln. py.

16.1 Time

As another example of a programmer-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 Figure 16.1.

As an exercise, write a function called print_time that takes a Time object and prints it in the form hour:minute:second . Hint: the format sequence %.2d prints an integer using at least two digits, including a leading zero if necessary.

Write a boolean function called is_after that takes two Time objects, t1 and t2, and re-turns True if t1 follows t2 chronologically and False otherwise. Challenge: don't use an if statement.

Figure 16.1: Object diagram.

16.2 Pure functions

In the next few sections, we'll write two functions that add time values. They demonstrate two kinds of functions: pure functions and modifiers. They also demonstrate a develop-ment 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 effect, like displaying a value or getting user input, other than returning a value.

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.

16.3 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 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. As an exercise, 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**.

As an exercise, write a "pure" version of increment that creates and returns a new Time object rather than modifying the parameter.

16.4 Prototyping versus planning

The development plan I am demonstrating is called "prototype and patch". For each function, I wrote a prototype that performed the basic calculation and then tested it, patching errors along the way.

This approach can be effective, especially if you don't yet have a deep understanding of the problem. But incremental corrections 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 **designed 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 (see http://en.wikipedia.org/wiki/Sexagesimal .)! 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 a 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 a function that converts an integer to a Time (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. One way to test them is to check that time_to_int(int_to_time(x)) == x for many values of x. This is an example of a consistency check.

Once you are convinced they are correct, 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. As an exercise, 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 time values 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 naive 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).

16.5 Debugging

A Time object is well-formed if the values of minute and second are between 0 and 60 (including 0 but not 60) and if hour is positive. hour and minute should be integral 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 function like valid_time that takes a Time object and returns False if it violates an invariant:

```
def valid_time(time):
if time.hour < 0 or time.minute < 0 or time.second < 0:
return False
if time.minute >= 60 or time.second >= 60:
return False
return True
```

At the beginning of each function you could check the arguments to make sure they are valid:

```
def add_time(t1, t2):
if not valid_time(t1) or not valid_time(t2):
raise ValueError( invalid Time object in add_time )
seconds = time_to_int(t1) + time_to_int(t2)
return int_to_time(seconds)
```

Or you could use an **assert statement**, which checks a given invariant and raises an excep-tion if it fails:

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

assert statements are useful because they distinguish code that deals with normal condi-tions from code that checks for errors.

Classes and methods

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 relation-ships 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.

Code examples from this chapter are available from http://thinkpython2.com/code/ Time2.py, and solutions to the exercises are in http://thinkpython2.com/code/Point2_ soln.py.

17.1 Object-oriented features

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

For example, the Time class defined in Chapter 16 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 in Chapter 15 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. These features are not strictly necessary; most of them provide 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 Time1.py there is no obvious connection between the class definition and the function definitions that follow. With some examination, it is apparent that every func-tion 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. 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.

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 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.

17.2 Printing objects

In Chapter 16, we defined a class named Time and in Section 16.1, you wrote a function named print_time:

```
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:

```
>>> 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 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 invoca-tion 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 (or methods), and makes it easier to maintain and reuse code.

As an exercise, rewrite time_to_int (from Section 16.4) as a method. You might be tempted to rewrite int_to_time as a method, too, but that doesn’t really make sense because there would be no object to invoke it on.

17.3 Another example

Here’s a version of increment (from Section 16.3) 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. 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 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:

```
# inside class Time:

def is_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.is_after(start)
True
```

One nice thing about this syntax is that it almost reads like English: "end is after start?"

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:

```
# 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 of 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.

As an exercise, write an init method for the Point class that takes x and y as optional parameters and assigns them to the corresponding attributes.

17.6 The __str__ method

__str__ is a special method, like __init__, that is supposed to return a string representa-tion 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 useful for debugging.

As an exercise, write a str method for the Point class. Create a Point object and print it.

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:

```
# 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 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 spe-cial method, like __add__. For more details, see http://docs.python.org/3/reference/datamodel.html#specialnames .

As an exercise, write an add method for the Point class.

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:

```
# 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 param-eter 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 argu-ments.

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. But there is a clever solution for this problem: the special method __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:

```
# inside class Time:

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

And here's how it's used:

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

As an exercise, 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.

17.9 Polymorphism

Type-based dispatch is useful when it is necessary, but (fortunately) it is not always neces-sary. 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 exam-ple, in Section 11.2 we used histogram to count the number of times each letter appears in a word.

```
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.

```
>>> t = [spam, egg, spam, spam, bacon, spam]
>>> histogram(t)
{bacon: 1, egg: 1, spam: 4}
```

Functions that work with several types are called **polymorphic**. Polymorphism can fa-cilitate 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.

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, the function works with that type.

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

17.10 Debugging

It is legal to add attributes to objects at any point in the execution of a program, but if you have objects with the same type that don't have the same attributes, it is easy to make mistakes. It is considered a good idea to initialize all of an object's attributes in the init method.

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:

```
>>> p = Point(3, 4)
>>> vars(p)
{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 vars(obj):
print(attr, getattr(obj, attr))
```

print_attributes traverses the dictionary and prints each attribute name and its corre-sponding 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 imple-mentations. 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.