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:

In [1]:
```
class Point:
    """Represents a point in 2-D space."""
```

The 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 methods inside a class definition, but we will get back to that later.

Defining a class named Point creates a class object.

In [3]:
```
print(Point)
```

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

Because Point is defined at the top level, its "full name" is $\_\_\mathrm{main}\_\_.\mathrm{Point}$.

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

In [4]:
```
blank = Point()
print(blank)
```

```
<__main__.Point object at 0x00000277E6B05748>
```

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:

In [5]:
```
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

In [6]:
```
print(blank.y)
x = blank.x
print(x)
```

```
4.0
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.

In [7]:
```
'(%g, %g)' % (blank.x, blank.y)
```

Out[7]:
```
'(3, 4)'
```

In [9]:
```
import math
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:

In [10]:
```
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:

In [11]:
```
print_point(blank)
```

```
(3, 4)
```

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

15.3 Rectangles

In [12]:
```
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:

In [13]:
```
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.”

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 argument and returns a Point that contains the coordinates of the center of the Rectangle:

In [14]:
```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

In [15]:
```
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:

In [16]:
```
box.width = box.width + 50
box.height = box.height + 100
```

You can also write functions that modify objects.

In [17]:
```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

In [18]:
```
box.width, box.height
```

Out[18]:
```
(150.0, 300.0)
```

In [19]:
```
grow_rectangle(box, 50, 100)
```

In [20]:
```
box.width, box.height
```

Out[20]:
```
(200.0, 400.0)
```

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

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:

In [21]:
```
p1 = Point()
p1.x = 3.0
p1.y = 4.0
```

In [23]:
```
import copy
p2=copy.copy(p1)
```

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

In [24]:
```
print(p1)
print(p2)
```

```
<__main__.Point object at 0x00000277E6BDB780>
<__main__.Point object at 0x00000277E61C38D0>
```

In [25]:
```
print_point(p1)
print_point(p2)
```

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

In [26]:
```
p1 is p2
```

Out[26]: False

In [27]:
```
p1 == p2
```

Out[27]: 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.

In [28]:
```
box2 = copy.copy(box)
box2 is box
```

Out[28]: False

In [29]:
```
box2.corner is box.corner
```

Out[29]: True

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

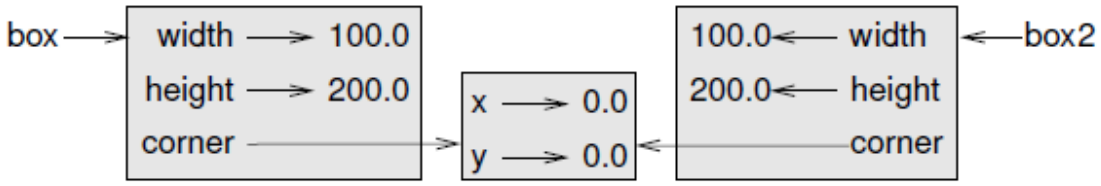

Figure 15.3: Object diagram.

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.

In [30]:
```
box3 = copy.deepcopy(box)
box3 is box
```

Out[30]: False

In [31]:
```
box3.corner is box.corner
```

Out[31]: False

box3 and box are completely separate objects

In [33]:
```
p = Point()
p.x = 3
p.y = 4
isinstance(p, Point)
```

Out[33]: True

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

In [34]:
```
hasattr(p, 'x')
```

Out[34]: True

In [35]:
```
hasattr(p, 'z')
```

Out[35]: False

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

In [36]:
```
#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
```

In [ ]:
```

```