Classes and objects

13.1. Object-oriented programming

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

Object-oriented programming has its roots in the 1960s, but it wasn't until the mid 1980s that it became the main programming paradigm used in the creation of new software. It was developed as a way to handle the rapidly increasing size and complexity of software systems, and to make it easier to modify these large and complex systems over time.

Up to now we have been writing programs using a procedural programming paradigm. In procedural programming the focus is on writing functions or *procedures* which operate on data. In object-oriented programming the focus is on the creation of **objects** which contain both data and functionality together.

13.2. User-defined compound types

A class in essence defines a new **data type**. We have been using several of Python's built-in types throughout this book, we are now ready to create our own user-defined type: the Point.

Consider the concept of a mathematical point. In two dimensions, a point is two numbers (coordinates) that are treated collectively as a single object. 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 xunits to the right and y units up from the origin.

A natural way to represent a point in Python is with two numeric values. The question, then, is how to group these two values into a compound object. The quick and dirty solution is to use a list or tuple, and for some applications that might be the best choice.

An alternative is to define a new user-defined compound type, also called a **class**. This approach involves a bit more effort, but it has advantages that will be apparent soon.

A class definition looks like this:

```
class Point:
    pass
```

Class definitions can appear anywhere in a program, but they are usually near the beginning (after the import statements). The syntax rules for a class definition are the same as for other compound statements. There is a header which begins with the keyword, class, followed by the name of the class, and ending with a colon.

This definition creates a new class called Point. The **pass** statement has no effect; it is only necessary because a compound statement must have something in its body. A docstring could serve the same purpose:

```
class Point:
    "Point class for storing mathematical points."
```

By creating the Point class, we created a new type, also called Point. The members of this type are called **instances** of the type or **objects**. Creating a new instance is called **instantiation**, and is accomplished by **calling the class**. Classes, like functions, are callable, and we instantiate a Point object by calling the Point class:

```
>>> type(Point)
<type 'classobj'>
>>> p = Point()
>>> type(p)
<type 'instance'>
```

The variable p is assigned a reference to a new Point object.

It may be helpful to think of a class as a factory for making objects, so our Point class is a factory for making points. The class itself isn't an instance of a point, but it contains the machinary to make point instances.

13.3. Attributes

Like real world objects, object instances have both form and function. The form consists of data elements contained within the instance.

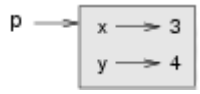
We can add new data elements to an instance using dot notation:

```
>>> p.x = 3
>>> p.y = 4
```

This syntax is similar to the syntax for selecting a variable from a module, such as math.pi or string.uppercase. Both modules and instances create their own

namespaces, and the syntax for accessing names contained in each, called **attributes**, is the same. In this case the attribute we are selecting is a data item from an instance.

The following state diagram shows the result of these assignments:



The variable p refers to a Point object, which contains two attributes. Each attribute refers to a number.

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

```
>>> print p.y
4
>>> x = p.x
>>> print x
3
```

The expression p.x means, “Go to the object p 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. The purpose of dot notation is to identify which variable you are referring to unambiguously.

You can use dot notation as part of any expression, so the following statements are legal:

```
print '(%d, %d)' % (p.x, p.y)
distance_squared = p.x * p.x + p.y * p.y
```

The first line outputs (3, 4); the second line calculates the value 25.

13.4. The initialization method and self

Since our Point class is intended to represent two dimensional mathematical points, *all* point instances ought to have x and y attributes, but that is not yet so with ourPoint objects.

```
>>> p2 = Point()
>>> p2.x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
AttributeError: Point instance has no attribute 'x'
>>>
```

To solve this problem we add an **initialization method** to our class.

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

A **method** behaves like a function but it is part of an object. Like a data attribute it is accessed using dot notation. The initialization method is called automatically when the class is called.

Let's add another method, distance_from_origin, to see better how methods work:

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

    def distance_from_origin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5
```

Let's create a few point instances, look at their attributes, and call our new method on them:

```
>>> p = Point(3, 4)
>>> p.x
3
>>> p.y
4
>>> p.distance_from_origin()
5.0
>>> q = Point(5, 12)
>>> q.x
5
>>> q.y
12
```

```
>>> q.distance_from_origin()
13.0
>>> r = Point()
>>> r.x
0
>>> r.y
0
>>> r.distance_from_origin()
0.0
```

When defining a method, the first parameter refers to the instance being created. It is customary to name this parameter **self**. In the example session above, the selfparameter refers to the instances p, q, and r respectively.

13.5. Instances as parameters

You can pass an instance as a parameter to a function in the usual way. For example:

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

print_point takes a point as an argument and displays it in the standard format. If you call print_point(p) with point p as defined previously, the output is (3, 4).

13.6. Glossary

class

A user-defined compound type. A class can also be thought of as a template for the objects that are instances of it.

instantiate

To create an instance of a class.

instance

An object that belongs to a class.

object

A compound data type that is often used to model a thing or concept in the real world.

attribute

One of the named data items that makes up an instance.

13.7. Exercises

Create and print a Point object, and then use id to print the object's unique identifier. Translate the hexadecimal form into decimal and confirm that they match.

Rewrite the distance function from chapter 5 so that it takes two Points as parameters instead of four numbers.

Source: http://openbookproject.net/thinkcs/python/english2e/ch13.html