Classes and objects

User-defined compound types

Consider the concept of a mathematical point. 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.

A natural way to represent a point in Python is with two floating-point values.

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.**

Definition of class

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).

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.

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.

To instantiate a Point object, we call a function named Point:

```
blank = Point()
```

The variable blank is assigned a reference to a new Point object. A function like Point that creates new objects is called a constructor.

Attributes

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

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

In this case, though, we are selecting a data item from an instance. These named items are called attributes.

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
```

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.

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

```
#!/usr/bin/python

class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

"This would create first object of Employee class"
emp1 = Employee("Zara", 2000)
"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
emp1.displayEmployee()
emp2.displayEmployee()
print "Total Employee %d" % Employee.empCount
```

You can add, remove, or modify attributes of classes and objects at any time –

```
emp1.age = 7  # Add an 'age' attribute.
emp1.age = 8  # Modify 'age' attribute.
del emp1.age  # Delete 'age' attribute.
```

Instead of using the normal statements to access attributes, you can use the following functions –

- The **getattr(obj, name[, default])** : to access the attribute of object.
- The **hasattr(obj,name)** : to check if an attribute exists or not.
- The **setattr(obj,name,value)** : to set an attribute. If attribute does not exist, then it would be created.
- The **delattr(obj, name)** : to delete an attribute.

```
hasattr(emp1, 'age')    # Returns true if 'age' attribute exists
getattr(emp1, 'age')    # Returns value of 'age' attribute
setattr(emp1, 'age', 8) # Set attribute 'age' at 8
delattr(empl, 'age')    # Delete attribute 'age'
```

```
class Employee:
   'Common base class for all employees'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1

   def displayCount(self):
     print "Total Employee %d" % Employee.empCount

   def displayEmployee(self):
      print "Name : ", self.name,  ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

Dot Notation in Expression

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

```
print '(' + str(blank.x) + ', ' + str(blank.y) + ')'
distanceSquared = blank.x * blank.x + blank.y * blank.y
```

The first line outputs (3.0, 4.0); the second line calculates the value 25.0.

```
>>> print blank
<__main__.Point instance at 80f8e70>
```

The result indicates that blank is an instance of the Point class and it was defined in main . 80f8e70 is the unique identifier for this object, written in hexadecimal (base 16).

Instances as arguments

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

```
def printPoint(p):
print '(' + str(p.x) + ', ' + str(p.y) + ')’
```

printPoint takes a point as an argument and displays it in the standard format.

If you call printPoint(blank), the output is (3.0, 4.0).

Sameness

The meaning of the word “same” seems perfectly clear until you give it some thought, and then you realize there is more to it than you expected.

For example 1: if you say, “Chris and I have the same car," you mean that his car and yours are the same make and model, but that they are two different cars.

Example 2: If you say, “Chris and I have the same mother," you mean that his mother and yours are the same person.1 So the idea of \sameness" is different depending on the context.

To find out if two references refer to the same object, use the == operator. For example:

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> p1 == p2
False
```

Shallow Equality

If we assign p1 to p2, then the two variables are aliases of the same object:

```
>>> p2 = p1
>>> p1 == p2
True
```

This type of equality is called shallow equality because it compares only the references, not the contents of the objects.

Deep Equality

When compare the contents of the objects, it is called deep equality. We can write a function called samePoint:

```
def samePoint(p1, p2) :
return (p1.x == p2.x) and (p1.y == p2.y)
```

Now if we create two different objects that contain the same data, we can use samePoint to find out if they represent the same point.

```
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Point()
>>> p2.x = 3
>>> p2.y = 4
>>> samePoint(p1, p2)
True
```

Of course, if the two variables refer to the same object, they have both shallow and deep equality.

Class: Rectangles

Now, we want a class to represent a rectangle, then there are a few possibilities to specify :

- center of the rectangle
- size (width and height) or
- one of the corners and the size or
- two opposing corners.

A conventional choice is to specify the upper-left corner of the rectangle and the size. Again, we'll define a new class:

```
class Rectangle:
pass
```

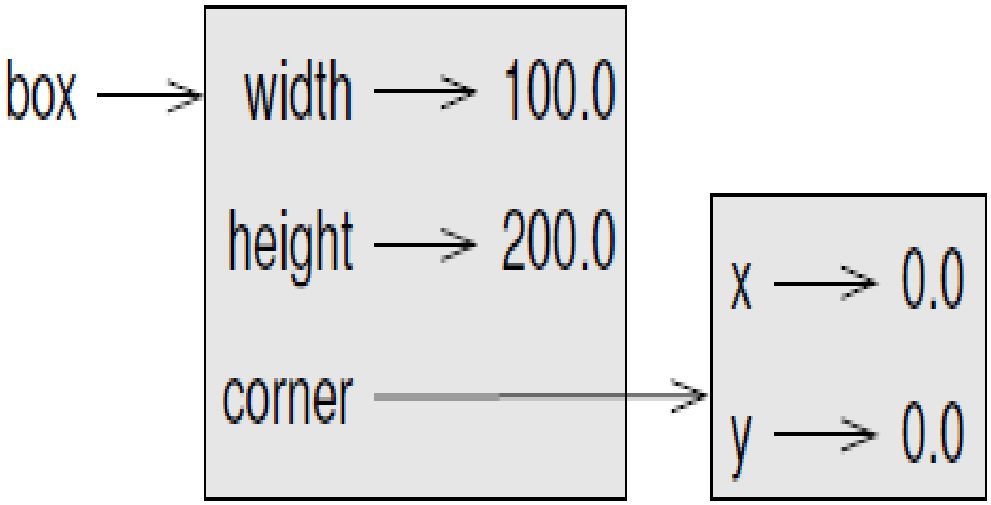
And instantiate it:

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
```

To specify the upper-left corner, we can embed an object within an object.

```
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

The figure shows the state of this object:



Instances as return values

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

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

To call this function, pass box as an argument and assign the result to a variable:

```
>>> center = findCenter(box)
>>> printPoint(center)
(50.0, -100.0)
```

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, we could modify the values of width and height:

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

We could encapsulate this code in a method and generalize it to grow the rectangle by any amount:

```
def growRect(box, dwidth, dheight) :
box.width = box.width + dwidth
box.height = box.height + dheight
```

For example, we could create a new Rectangle named bob and pass it to growRect:

```
>>> bob = Rectangle()
>>> bob.width = 100.0
>>> bob.height = 200.0
>>> bob.corner = Point()
>>> bob.corner.x = 0.0
>>> bob.corner.y = 0.0
>>> growRect(bob, 50, 100)
```

While growRect is running, the parameter box is an alias for bob. Any changes made to box also affect bob.

Copying

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

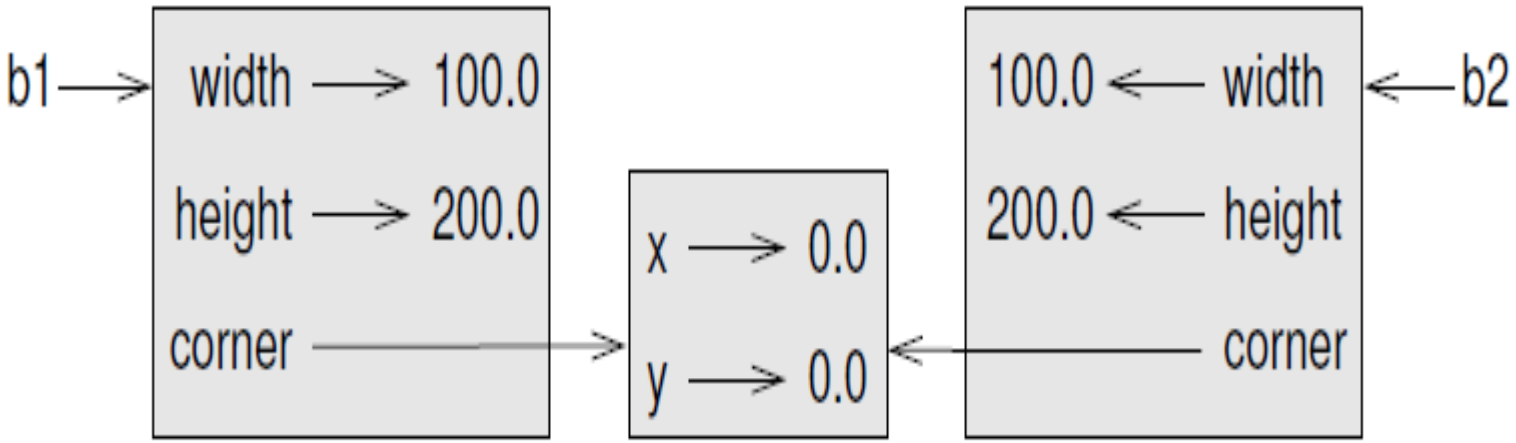
```
>>> import copy
>>> p1 = Point()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2                    False
>>> samePoint(p1, p2)           True
```

Shallow Copying

To copy a simple object like a Point, which doesn't contain any embedded objects, copy is sufficient. This is called **shallow copying.**

For something like a Rectangle, which contains a reference to a Point, copy doesn't do quite the right thing. It copies the reference to the Point object, so both the old Rectangle and the new one refer to a single Point.

If we create a box, b1, in the usual way and then make a copy, b2, using copy, the resulting state diagram looks like this:



Deepcopy

Fortunately, the copy module contains a method named deepcopy that copies not only the object but also any embedded objects.

```
>>> b2 = copy.deepcopy(b1)
```

Now b1 and b2 are completely separate objects.

We can use deepcopy to rewrite growRect so that instead of modifying an existing Rectangle, it creates a new Rectangle that has the same location as the old one but new dimensions:

```
def growRect(box, dwidth, dheight) :
import copy
newBox = copy.deepcopy(box)
newBox.width = newBox.width + dwidth
newBox.height = newBox.height + dheight
return newBox
```