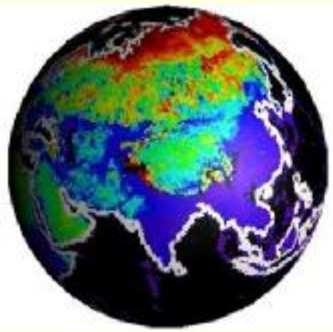



# Chapter 15 Classes and objects

**Jihn-Fa Jan, Ph.D.**
**Associate Professor**
**Department of Land Economics**
**National Chengchi University**

# 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.**

# User-defined types (cont'd)

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

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

- **This header indicates that the new class is a Point, which is a kind of object, which is a built-in type.**
- **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.**

# User-defined types (cont'd)

- Defining a class named Point creates a class object.

```
>>> print 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()
>>> 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.

# User-defined types (cont'd)

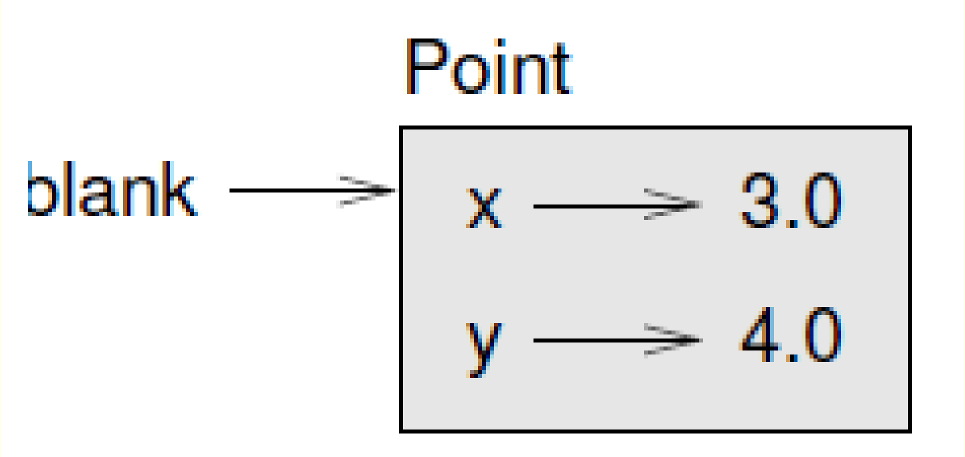
- When you print an instance, Python tells you what class it belongs to and where it is stored in memory (the prefix 0x means that the following number is in hexadecimal).

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



# Attributes

- **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:**

```
>>> 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.

# Attributes (cont'd)

- 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)
```

# 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.

# Rectangles (cont'd)

- **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(object):
    """represent a rectangle.
       attributes: width, height, corner.
    """
```

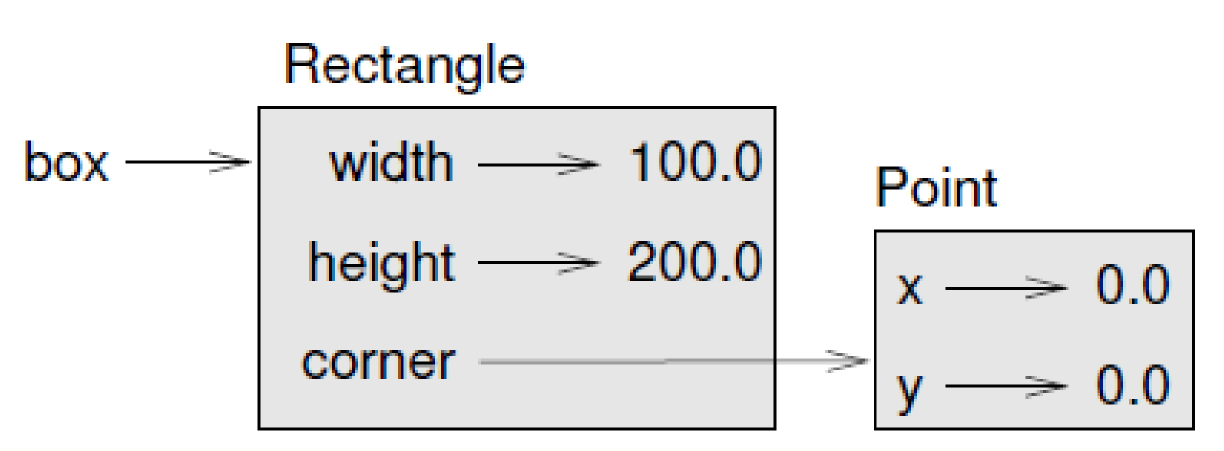
  - The docstring lists the attributes: width and height are numbers; corner is a Point object that specifies the lower-left corner.

- **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
```

# Rectangles (cont'd)

- **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:**




- **An object that is an attribute of another object is embedded.**

# 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 assigns the resulting Point to center:

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

# 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.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
```

# Objects are mutable (cont'd)

- **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.

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

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

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

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

# Copying (cont'd)

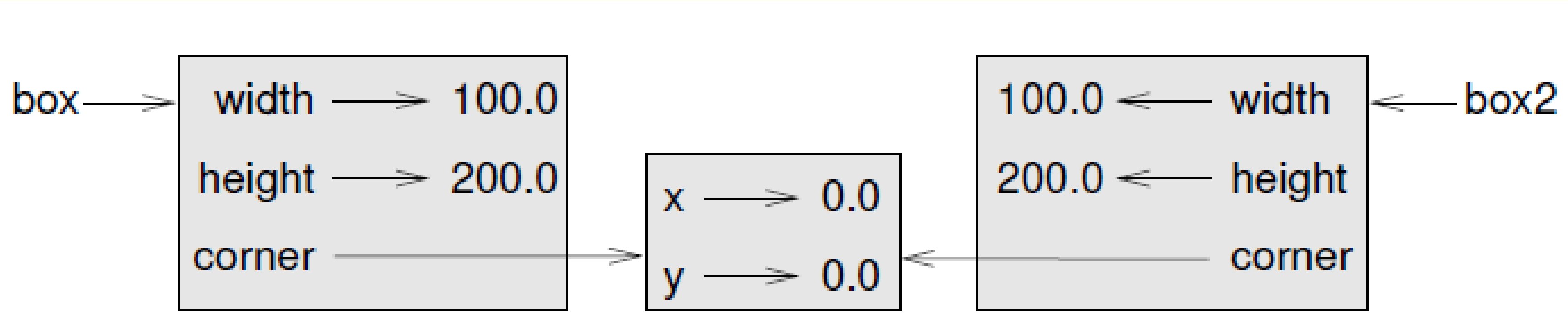
- The is operator indicates that p1 and p2 are not the same object, which is what we expected. But you might have expected == to yield True because these points contain the same data. In that case, you will be disappointed to learn that for instances, the default behavior of the == operator is the same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

# Copying (cont'd)

- **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.

# Copying (cont'd)

- **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.

# 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()
>>> 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 '__main__.Point'>
```

# Debugging (cont'd)

- 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.