Object-Oriented Programming

- Defining new Python Classes
- Overloaded Operators

Object-Oriented Programming (OOP)

Code reuse is a key benefit of organizing code into new classes; it is made possible through abstraction and encapsulation.

Abstraction: The idea that a class object can be manipulated by users through method invocations alone and without knowledge of the implementation of these methods.

- Abstraction facilitates software development because the programmer works with objects abstractly (i.e., through "abstract", meaningful method names rather than "concrete", technical code).

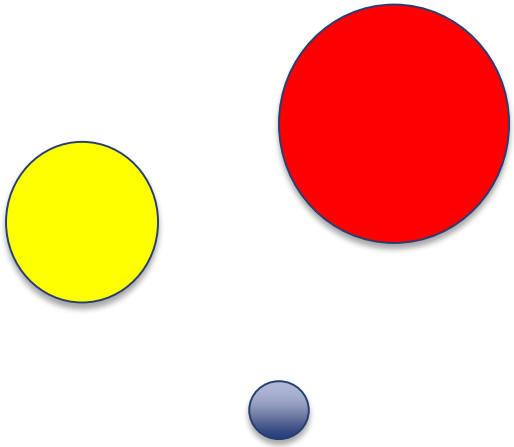
Encapsulation: In order for abstraction to be beneficial, the "concrete" code and data associated with objects must be encapsulated (i.e., made "invisible" to the program using the object).

- Encapsulation is achieved thanks to the fact that (1) every class defines a namespace in which class attributes live, and (2) every object has a namespace, that inherits the class attributes, in which instance attributes live.

OOP is an approach to programming that achieves modular code through the use of objects and by structuring code into user-defined classes.

The object of this class…

- is to understand the difference between **objects** and classes
- Describe the circles;



Classifying objects

- Although the 3 circles are different they are all circles, they have the **same descriptive figures**
  - Color
  - Radius
  - Center coordinates (x,y)
- They belong to the same “class” of objects

Object-Oriented Design (OOD)

1. Object combines ***data*** and ***operations*** into a unit
   - Data - Descriptive attributes
   - Operations - Behaviors
2. A ***class*** is a collection of objects that share the same data attributes and behavior; those object are said to be ***instances*** of their class
3. Though all objects of a class share the same data attributes, each has its own ***state***, or value of the data attributes.

Class syntax

```
class <Class Name>:
        <class variable 1> = <value>
        <class variable 2> = <value>
        . . .
        def <class method 1>(self, arg1, arg2, . . .):
            <implementation of class method 1>
        def <class method 2>(self, arg1, arg2, . . .):
            <implementation of class method 2>
```

Usually consists of

- Variables
- Methods

A new class: `Point`

Suppose we would like to have a class that represents points on a plane

- for a graphics app, say

Let's first informally describe how we would like to use this class

```
>>> point = Point()
>>> point.setx(3)
>>> point.sety(4)
>>> point.get()
(3, 4)
>>> point.move(1, 2)
>>> point.get()
(4, 6)
>>> point.setx(-1)
>>> point.get()
(-1, 6)
>>>
```

Usage	Explanation
`p.setx(xcoord)`	Sets the x coordinate of point `p` to `xcoord`
`p.sety(ycoord)`	Sets the y coordinate of point `p` to `ycoord`
`p.get()`	Returns the x and y coordinates of point `p` as a tuple (x, y)
`p.move(dx, dy)`	Changes the coordinates of point `p` from the current (x, y) to (x+dx, y+dy)

How do we create this new class `Point`?

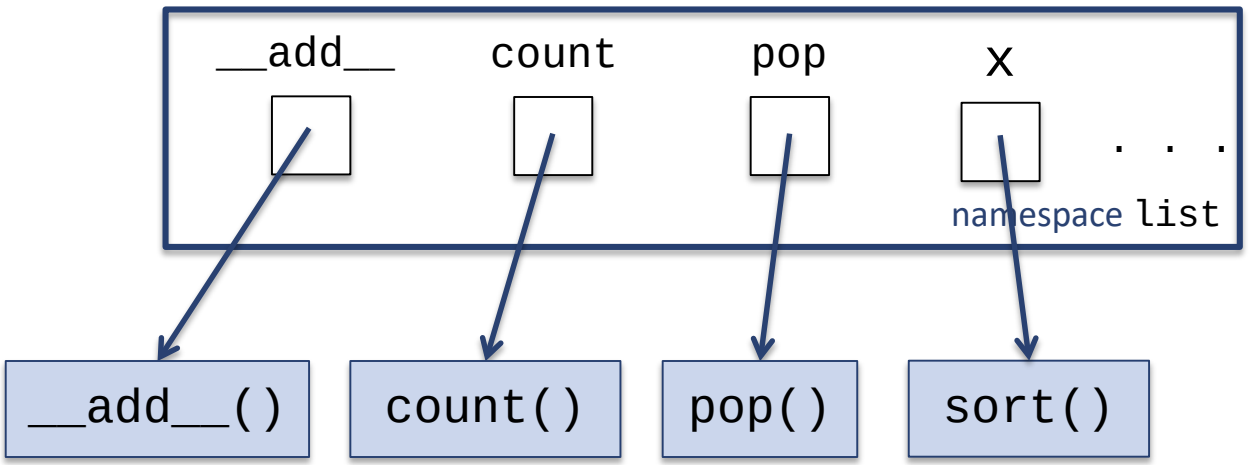
A class is a namespace (REVIEW)

A class is really a namespace

- The name of this namespace is the name of the class
- The names defined in this namespace are the class attributes (e.g., class methods)
- The class attributes can be accessed using the standard namespace notation

```
>>> list.pop
<method 'pop' of 'list' objects>
>>> list.sort
<method 'sort' of 'list' objects>
>>> dir(list)
['__add__', '__class__',
...
'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

Function `dir()` can be used to list the class attributes



Class methods (REVIEW)

A class method is really a function defined in the class namespace; when Python executes

```
instance.method(arg1, arg2, …)
```

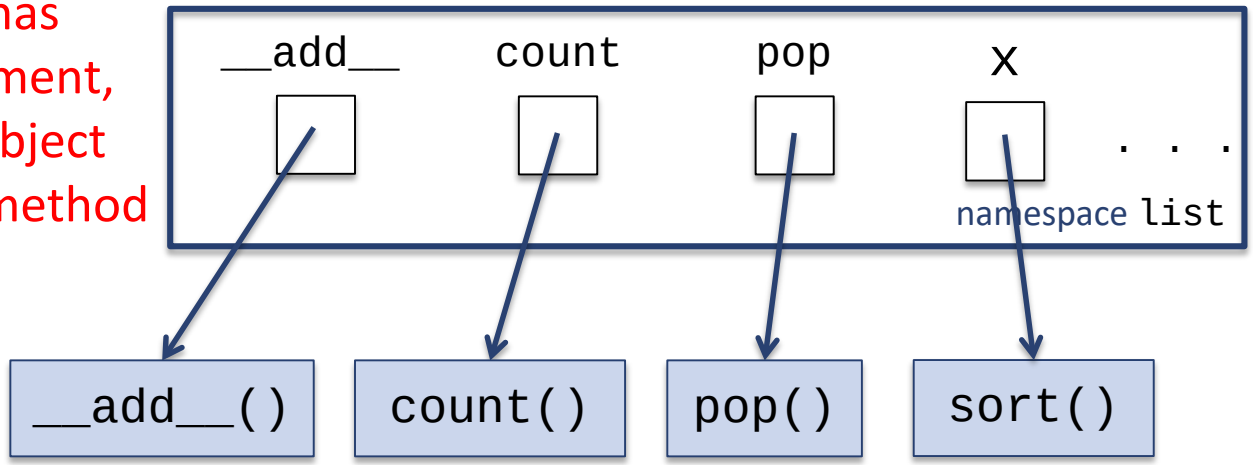
it first translates it to

```
class.method(instance, arg1, arg2, …)
```

and actually executes this last statement

The function has an extra argument, which is the object invoking the method

```
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> lst.sort()
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst = [9, 1, 8, 2, 7, 3]
>>> lst
[9, 1, 8, 2, 7, 3]
>>> list.sort(lst)
>>> lst
[1, 2, 3, 7, 8, 9]
>>> lst.append(6)
>>> lst
[1, 2, 3, 7, 8, 9, 6]
>>> list.append(lst, 5)
>>> lst
[1, 2, 3, 7, 8, 9, 6, 5]
```

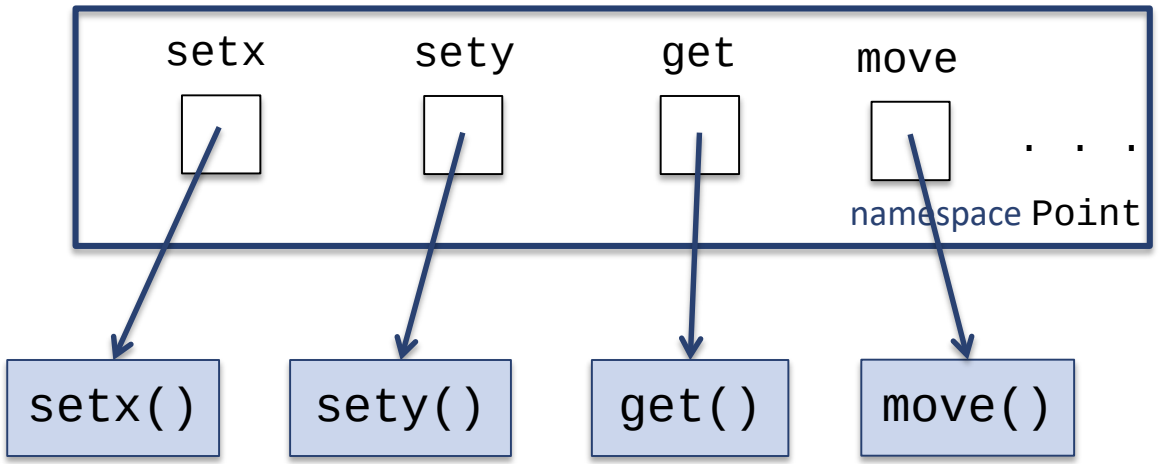


Developing the class `Point`

A namespace called `Point` needs to be defined

Namespace `Point` will store the names of the 4 methods (the class attributes)

Usage	Explanation
`p.setx(xcoord)`	Sets the x coordinate of point `p` to `xcoord`
`p.sety(ycoord)`	Sets the y coordinate of point `p` to `ycoord`
`p.get()`	Returns the x and y coordinates of point `p` as a tuple (x, y)
`p.move(dx, dy)`	Changes the coordinates of point `p` from the current (x, y) to (x+dx, y+dy)



Defining the class `Point`

A namespace called `Point` needs to be defined

Namespace `Point` will store the names of the 4 methods (the class attributes)

Each method is a function that has an extra (first) argument which refers to the object that the method is invoked on

Usage	Explanation
`setx(p, xcoord)`	Sets the x coordinate of point `p` to `xcoord`
`sety(p, ycoord)`	Sets the y coordinate of point `p` to `ycoord`
`get(p)`	Returns the x and y coordinates of point `p` as a tuple (x, y)
`move(p, dx, dy)`	Changes the coordinates of point `p` from the current (x, y) to (x+dx, y+dy)

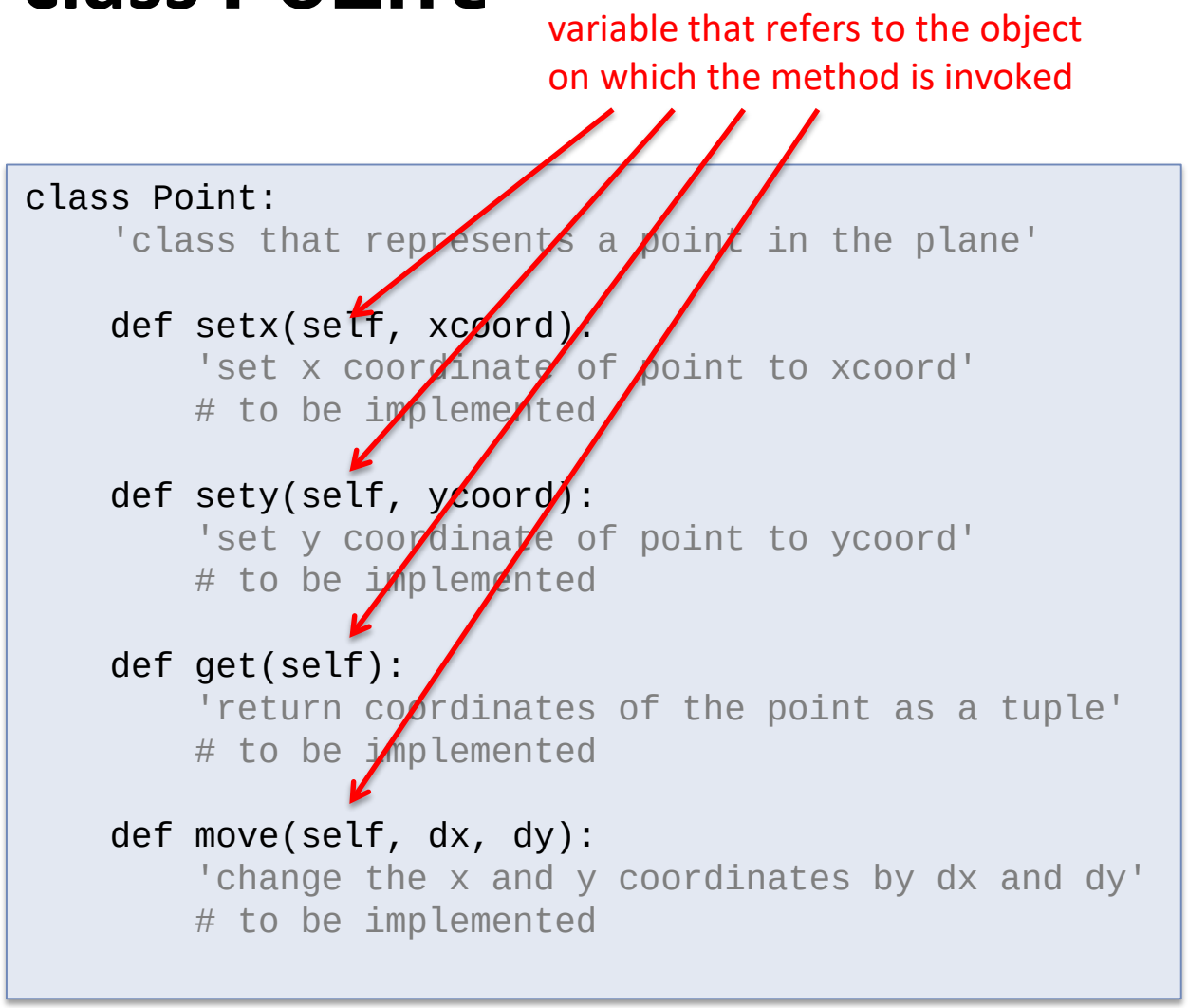
```
>>> Point.get(point)
(-1, 6)
>>> Point.setx(point, 0)
>>> Point.get(point)
(0, 6)
>>> Point.sety(point, 0)
>>> Point.get(point)
(0, 0)
>>> Point.move(point, 2, -2)
>>> Point.get(point)
(2, -2)
```

Defining the class `Point`

A namespace called `Point` needs to be defined

Namespace `Point` will store the names of the 4 methods (the class attributes)

Each method is a function that has an extra (first) argument which refers to the object that the method is invoked on



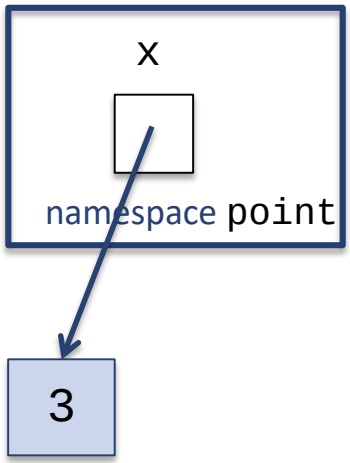
The Python `class` statement defines a new class (and associated namespace)

The object namespace

We know that a namespace is associated with every class

A namespace is also associated with every object

```
>>> point = Point()
>>> Point.setx(point, 3)
>>>
```




```
class Point:
    'class that represents a point in the plane'

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        # to be implemented

    def get(self):
        'return coordinates of the point as a tuple'
        # to be implemented

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        # to be implemented
```

The Python `class` statement defines a new class

Defining the class `Point`

A namespace called `Point` needs to be defined

Namespace `Point` will store the names of the 4 methods (the class attributes)

Each method is a function that has an extra (first) argument which refers to the object that the method is invoked on

```
class Point:
    'class that represents a point in the plane'

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        self.y = ycoord

    def get(self):
        'return coordinates of the point as a tuple'
        return (self.x, self.y)

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        self.x += dx
        self.y += dy
```

Exercise

Add new method `getx()` to class Point

```
>>> point = Point()
>>> point.setx(3)
>>> point.getx()
3
```

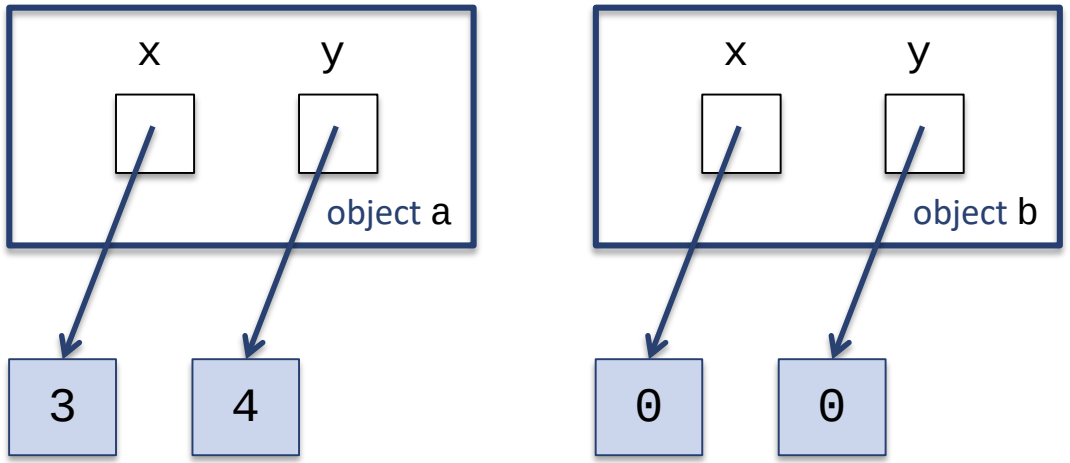
```
class Point:
    'class that represents a point in the plane'

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        self.y = ycoord

    def get(self):
        'return coordinates of the point as a tuple'
        return (self.x, self.y)

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        self.x += dx
        self.y += dy

    def getx(self):
        'return x coordinate of the point'
        return self.x
```

The instance namespaces

Variables stored in the namespace of an object (instance) are called instance variables (or instance attributes)

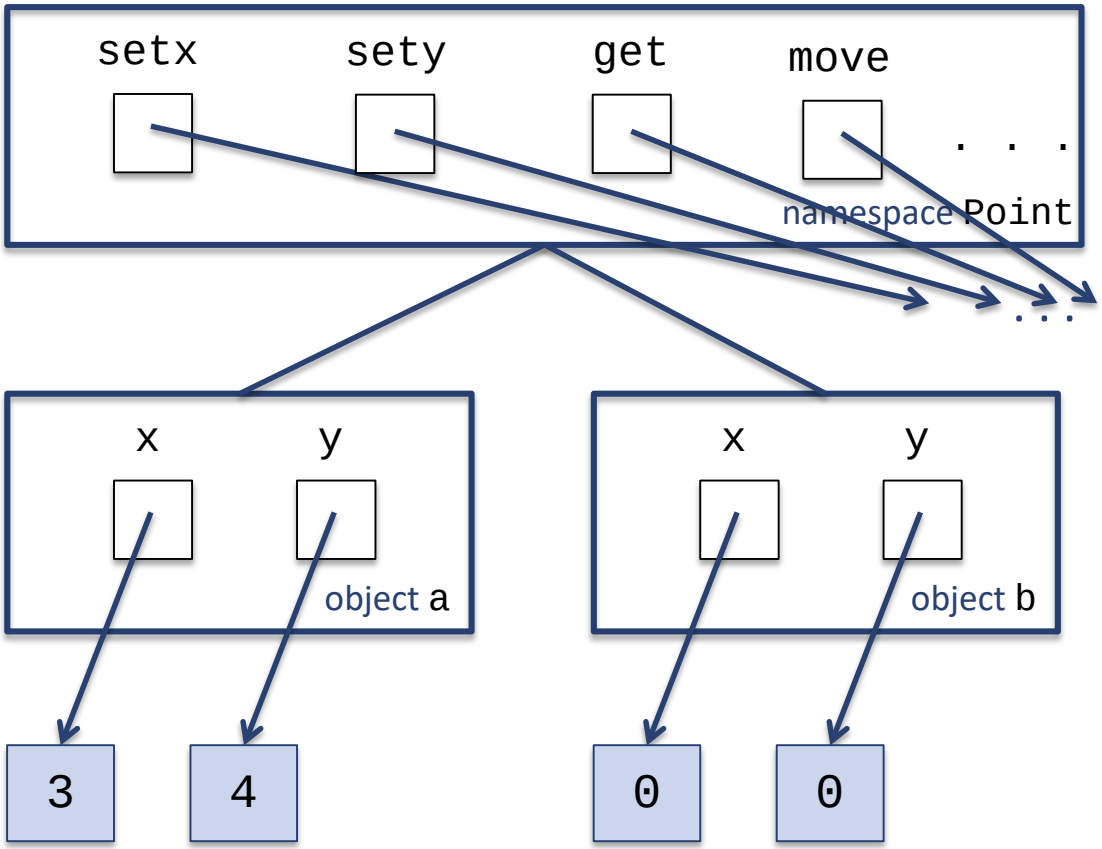
Every object will have its own namespace and therefore its own instance variables

```
>>> a = Point()
>>> a.setx(3)
>>> a.sety(4)
>>> b = Point()
>>> b.setx(0)
>>> b.sety(0)
>>> a.get()
(3, 4)
>>> b.get()
(0, 0)
>>> a.x
3
>>> b.x
0
>>>
```



The class and instance attributes

An instance of a class inherits all the class attributes




```
>>> dir(a)
['__class__', '__delattr__',
'__dict__', '__doc__',
'__eq__', '__format__',
'__ge__', '__getattribute__',
'__gt__', '__hash__',
'__init__', '__le__',
'__lt__', '__module__',
'__ne__', '__new__',
'__reduce__', '__reduce_ex__',
'__repr__', '__setattr__',
'__sizeof__', '__str__',
'__subclasshook__',
'__weakref__', 'get', 'move',
'setx', 'sety', 'x', 'y']
```

class Point attributes inherited by a instance attributes of a

Function `dir()` returns the attributes of an object, including the inherited ones

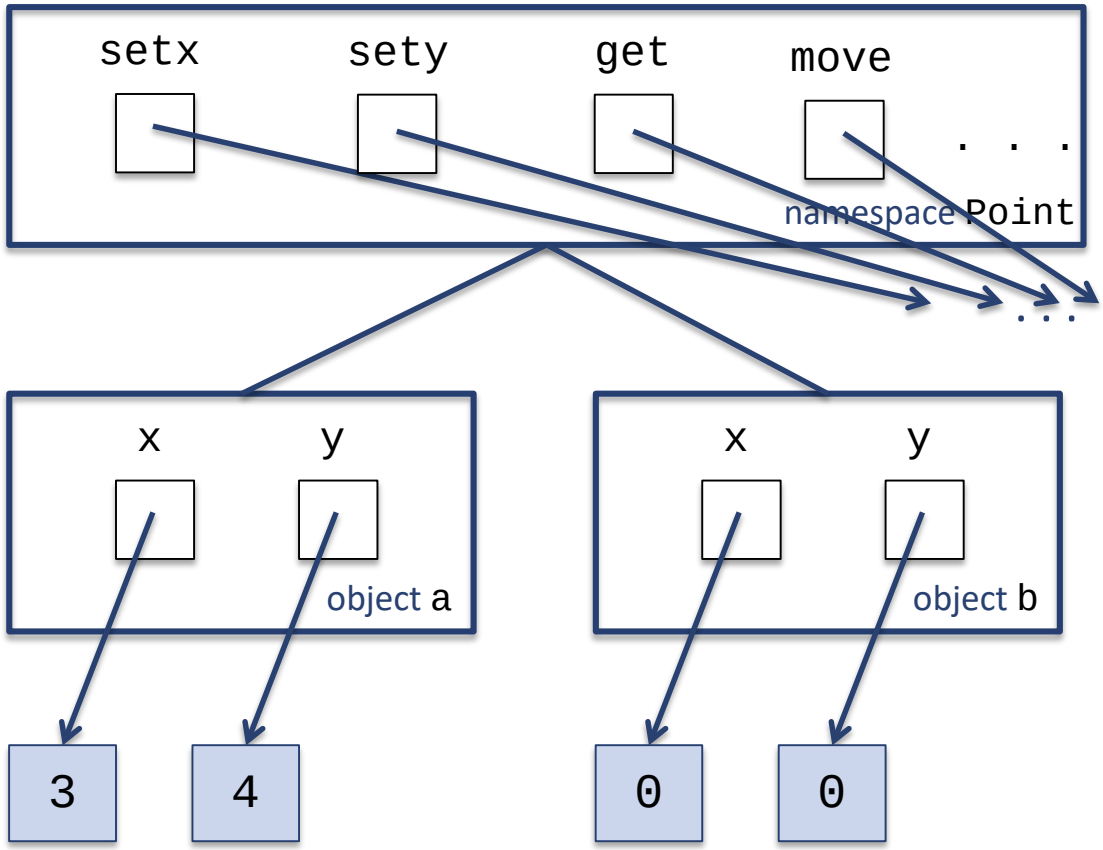
The class and instance attributes

Method names `setx`, `sety`, `get`, and `move` are defined in namespace `Point`

- not in namespace `a` or `b`.

Python does the following when evaluating expression `a.setx`:

1. It first attempts to find name `setx` in object (namespace) `a`.
2. If name `setx` does not exist in namespace `a`, then it attempts to find `setx` in namespace `Point`



Class definition, in general

```
class Point:

    def setx(self, xcoord):
        self.x = xcoord

    def sety(self, ycoord):
        self.y = ycoord

    def get(self):
        return (self.x, self.y)

    def move(self, dx, dy):
        self.x += dx
        self.y += dy
```

Note: no documentation

(No) class documentation

```
>>> help(Point)
Help on class Point in module __main__:

class Point(builtins.object)
 |  Methods defined here:
 |
 |  get(self)
 |
 |  move(self, dx, dy)
 |
 |  setx(self, xcoord)
 |
 |  sety(self, ycoord)
 |
 |  ----------------------------------------------------------------------
-
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Class documentation

```
class Point:
    'class that represents a point in the plane'

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        self.y = ycoord

    def get(self):
        'return coordinates of the point as a tuple'
        return (self.x, self.y)

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        self.x += dx
        self.y += dy
```

Class documentation

```
>>> help(Point)
Help on class Point in module __main__:

class Point(builtins.object)
 |  class that represents a point in the plane
 |
 |  Methods defined here:
 |
 |  get(self)
 |      return a tuple with x and y coordinates of the point
 |
 |  move(self, dx, dy)
 |      change the x and y coordinates by dx and dy
 |
 |  setx(self, xcoord)
 |      set x coordinate of point to xcoord
 |
 |  sety(self, ycoord)
 |      set y coordinate of point to ycoord
 |
 |  ----------------------------------------------------------------------
-
 |  Data descriptors defined here:
...
```

Overloaded constructor

called by Python each time a `Point` object is created

It takes 3 steps to create a `Point` object at specific x and y coordinates

```
>>> a = Point()
>>> a.setx(3)
>>> a.sety(4)
>>> a.get()
(3, 4)
>>>
```

It would be better if we could do it in one step

```
>>> a = Point(3, 4)
>>> a.get()
(3, 4)
>>>
```

```
class Point:
    'class that represents a point in the plane'

 def __init__(self, xcoord, ycoord):
        'initialize coordinates to (xcoord, ycoord)'
        self.x = xcoord
        self.y = ycoord

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        self.y = ycoord

    def get(self):
        'return coordinates of the point as a tuple'
        return (self.x, self.y)

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        self.x += dx
        self.y += dy
```

Default constructor

xcoord is set to 0 if the argument is missing

ycoord is set to 0 if the argument is missing

Problem: Now we can't create an uninitialized point

Built-in types support default constructors

Want to augment class Point so it supports a default constructor

```
>>> a = Point()
Traceback (most recent call
  File "<pyshell#0>", line 1
    a = Point()
TypeError: __init__() takes
>>>
```

```
>>> a = Point()
>>> a.get()
(0, 0)
>>>
```

```
>>> n
0
>>>
```

```
class Point:
    'class that represents a point in the plane'

 def __init__(self, xcoord=0, ycoord=0):
        'initialize coordinates to (xcoord, ycoord)'
        self.x = xcoord
        self.y = ycoord

    def setx(self, xcoord):
        'set x coordinate of point to xcoord'
        self.x = xcoord

    def sety(self, ycoord):
        'set y coordinate of point to ycoord'
        self.y = ycoord

    def get(self):
        'return coordinates of the point as a tuple'
        return (self.x, self.y)

    def move(self, dx, dy):
        'change the x and y coordinates by dx and dy'
        self.x += dx
        self.y += dy
```

Exercise

Develop class `Animal` that supports methods:

- `setSpecies(species)`
- `setLanguage(language)`
- `speak()`

```
>>> snoopy = Animal()
>>> snoopy.setpecies('dog')
>>> snoopy.setLanguage('bark')
>>> snoopy.speak()
I am a dog and I bark.
```

```
class Animal:
    'represents an animal'

    def setSpecies(self, species):
        'sets the animal species'
        self.spec = species

    def setLanguage(self, language):
        'sets the animal language'
        self.lang = language

    def speak(self):
        'prints a sentence by the animal'
        print('I am a {} and I {}.'.format(self.spec, self.lang))
```