

## 19. Introduction to Classes

### Topics:

- Class Definitions and Objects
- Accessing Attributes
- Copying Objects
- Functions and classes
- Lists of Objects

## Motivation

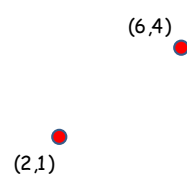
### What a Simple Class Definition Looks Like

```
class Point:
    """
    Attributes:
        x: float, the x-coordinate of a point
        y: float, the y-coordinate of a point
    """
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

A class can be used to ``package'' related data

### One Reason for classes: They Elevate the Level Thinking

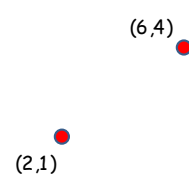
```
>>> P = Point(2,1)
>>> Q = Point(6,4)
>>> d = dist(P,Q)
>>> print d
5
```



Here, `dist` is a function that takes two `Points` and computes the distance between them.

### One Reason for classes: They Elevate the Level Thinking

```
>>> P = Point(2,1)
>>> Q = Point(6,4)
>>> d = dist(P,Q)
>>> print d
5
```



By having a `Point` class we can think at the "point level" instead of at the "xy level"

## Classes and Types

Recall that a type is a set of values and operations that can be performed on those values.

The four basic "built-in" types:

`int`, `float`, `str`, `bool`

Classes are a way to define new types

## Examples

By suitably defining a rectangle class, we could say something like

```
if R1.intersect(R2):
    print 'Rectangles R1 and R2 intersect'
```

## Examples

By suitably defining a polynomial class, we could perform operations like

$$p = q + r$$

where  $q$  and  $r$  are polynomials that are added together to produce a polynomial  $p$

## How to Define a Class

## A Point Class

```
class Point:
    """
    Attributes:
        x: float, the x-coordinate of a point
        y: float, the y-coordinate of a point
    """
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

A "blue print" for packaging data.  
The data will be stored in the attributes.

## A Point Class

```
class Point:
    """
    Attributes:
        x: float, the x-coordinate of a point
        y: float, the y-coordinate of a point
    """
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

This special function, called a constructor, does the packaging.

## A Point Class

```
class Point:
    """
    Attributes:
        x: float, the x-coordinate of a point
        y: float, the y-coordinate of a point
    """
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

The name of this class is "Point"

### The "\_\_init\_\_" Function

```
def __init__(self,x,y):
    """ Creates a Point object

    PreC: x and y are floats
    """
    self.x = x
    self.y = y
```

That's a double underscore: \_\_init\_\_

### The "\_\_init\_\_" Function

```
def __init__(self,x,y):
    """ Creates a Point object

    PreC: x and y are floats
    """
    self.x = x
    self.y = y
```

"self" is always the first argument for any function defined in a class.

### The "\_\_init\_\_" Function

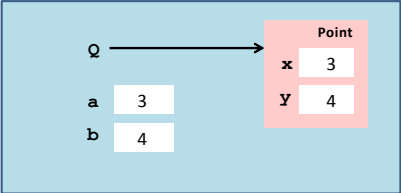
```
def __init__(self,x,y):
    """ Creates a Point object

    PreC: x and y are floats
    """
    self.x = x
    self.y = y
```

The attributes are assigned values.

### Call the Constructor to Create an Object

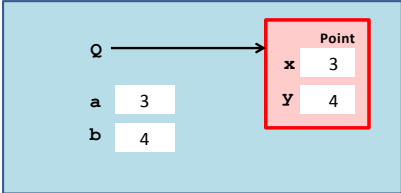
### Calling the Constructor



```
>>> a = 3
>>> b = 4
>>> Q = Point(a,b)
```

The constructor's name is the name of the class

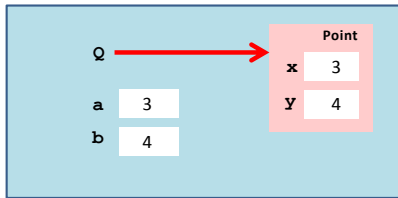
### Calling the Constructor



```
>>> a = 3
>>> b = 4
>>> Q = Point(a,b)
```

This creates a Point object

## Calling the Constructor



```
>>> a = 3
>>> b = 4
>>> Q = Point(a,b)
```

The constructor returns a reference

## Objects: The Folder Metaphor

Manila folders organize data.

Objects organize data.

A point object houses float variables `x` and `y`, called the attributes, where `(x,y)` is the point.

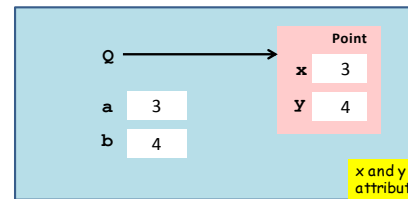
## Objects: The Folder Metaphor

Manila folders organize data.

Objects organize data.

A color object might house an rgb triple `[1,0,1]` and a name 'magenta'

## Visualizing a Point Object



```
>>> a = 3
>>> b = 4
>>> Q = Point(a,b)
```

`x` and `y` are attributes

Attributes are variables that live inside objects

## Accessing an Attribute The "Dot Notation" Again

Not a coincidence: modules are objects

## Accessing Attributes

```
>>> Q = Point(3,4)
>>> print Q
( 3.000, 4.000)
>>> Q.x = Q.x + 5
>>> print Q
( 8.000, 4.000)
```

`Q.x` is a variable and can "show up" in all the usual places, i.e., in an assignment statement.

### Accessing Attributes

```

>>> Q = Point(3,4)
>>> print Q
( 3.000, 4.000)
>>> Q.x = Q.x + 5
>>> print Q
( 8.000, 4.000)

```

Seems that we can print an object!

### The "\_\_str\_\_" function

```

def __str__(self):
    return '%6.3f,%6.3f' %(self.x,self.y)

```

This function is part of the class definition.

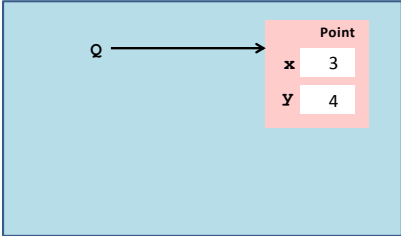
Whenever a statement like

```
print P
```

is encountered, then P is printed according to format rules.

### A Note on Copying an Object

### Not Making a Copy of a Point

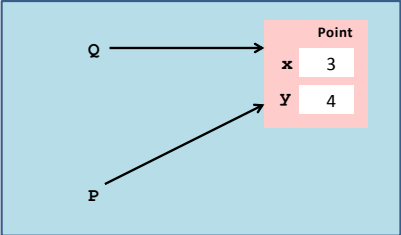


```

>>> Q = Point(3,4)
>>> P = Q

```

### Not Making a Copy of a Point

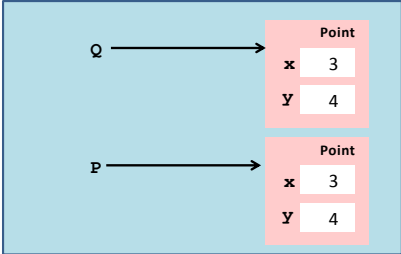


```

>>> Q = Point(3,4)
>>> P = Q

```

### Making a Copy of a Point



```

>>> Q = Point(3,4)
>>> P = copy(Q)

```

## The Module copy

```
from copy import copy
```

Import this function and use it to make copies of objects

`deepcopy` is another useful function from this module—more later.

## Using copy

```
>>> Q = Point(3,4)
>>> P1 = copy(Q)
>>> P1.x = 5
>>> print Q
( 3.000, 4.000)
>>> print P1
( 5.000, 4.000)
```

We are modifying P1, but Q remains the same.

## Example: A Function that Returns a Point Object

## Computing a Random Point

```
def RandomPoint(L,R):
    """ Returns a point that is randomly chosen
    from the square L<=x<=R, L<=y<=R.

    PreC: L and R are floats with L<R
    """
    x = randu(L,R)
    y = randu(L,R)
    P = Point(x,y)
    return P
```

calling the  
constructor

## Another Example: Computing the Midpoint

```
def MidPoint(P1,P2):
    """ Returns a point that is the midpoint of
    a line segment that connects P1 and P2.

    PreC: P1 and P2 are points.
    """
    xm = (P1.x + P2.x)/2.0
    ym = (P1.y + P2.y)/2.0
    Q = Point(xm,ym)
    return Q
```

## Computing the Midpoint

```
def MidPoint(P1,P2):
    """ Returns a point that is the midpoint of
    a line segment that connects P1 and P2.

    PreC: P1 and P2 are points.
    """
    xm = (P1.x + P2.x)/2.0
    ym = (P1.y + P2.y)/2.0
    Q = Point(xm,ym)
    return Q
```

referencing  
a point's  
attributes

calling the  
constructor

## Distance Between Two Points

```
def Dist(P1,P2):
    """ Returns a float that is the distance
    from P1 to P2.

    PreC: P1 and P2 are points
    """
    d = sqrt((P1.x-P2.x)**2+(P1.y-P2.y)**2)
    return d
```

## Affirmation of Midpoint

```
>>> P1 = RandomPoint(-10,10)
>>> P2 = RandomPoint(-10,10)
>>> M = MidPoint(P1,P2)
>>> print Dist(M,P1)
4.29339610681
>>> print Dist(M,P2)
4.29339610681
```

## A List of Objects

We would like to assemble a list whose elements are not numbers or strings, but references to objects.

For example, we have a hundred points in the plane and a length-100 list of points called `ListOfPoints`.

Let's compute the centroid.

## A List of Objects

```
sx = 0
sy = 0
for P in ListOfPoints:
    sx += P.x
    sy += P.y
N = len(ListOfPoints)
TheCentroid = Point(sx/N,sy/N)
```

A lot of familiar stuff. Running sums. A for-loop. The len function, Etc

## A List of Random Points

```
def RandomCloud(L,R,n):
    """ Returns a length-n list of points,
    each chosen randomly from the square
    L<=x<=R, L<=y<=R.

    PreC: L and R are floats with L<R,
    n is a positive int.
    """
    A = []
    for k in range(n):
        P = RandomPoint(L,R)
        A.append(P)
    return A
```

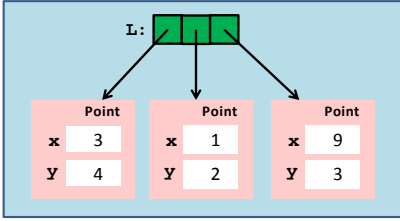
## A List of Random Points

```
def RandomCloud(L,R,n):
    """ Returns a length-n list of points,
    each chosen randomly from the square
    L<=x<=R, L<=y<=R.

    PreC: L and R are floats with L<R,
    n is a positive int.
    """
    A = []
    for k in range(n):
        P = RandomPoint(L,R)
        A.append(P)
    return A
```

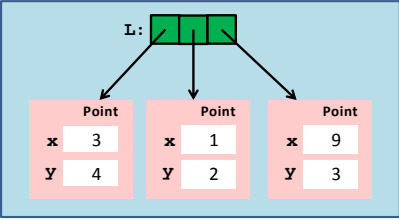
The append method for lists works for lists of objects

### Visualizing a List of Points



```
>>> P = Point(3,4) ; Q = Point(1,2) ; R = Point(9,3)
>>> L = [P,Q,R]
```

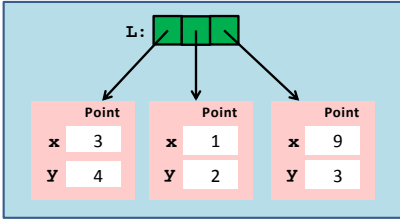
### Visualizing a List of Points



```
>>> P = Point(3,4) ; Q = Point(1,2) ; R = Point(9,3)
>>> L = [P,Q,R]
```

More accurate: A List of references to Point objects

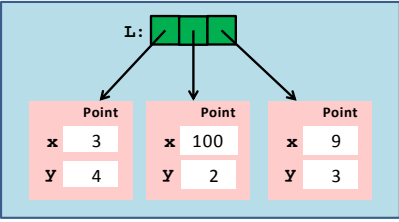
### Operations on a List of Points



```
>>> L[1].x = 100
```

Before

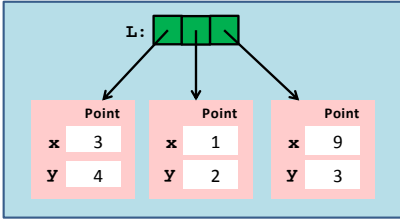
### Operations on a List of Points



```
>>> L[1].x = 100
```

After

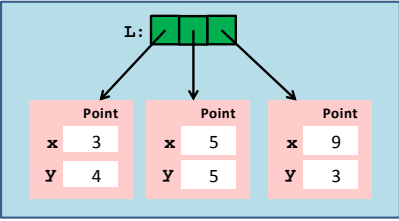
### Operations on a List of Points



```
>>> L[1] = Point(5,5)
```

Before

### Operations on a List of Points



```
>>> L[1] = Point(5,5)
```

After



## Printing a List of Points

```
def printCloud(A):
    """ Prints the points in A

    PreC : A is a list of points.
    """
    for a in A:
        print a
```

Synonym for the loop:

```
for k in range(len(A)):
    print A[k]
```

## An Odometer Function

```
def odometer(A):
    """ Returns a float that is the
    perimeter of the polygon whose vertices
    are the points in A.

    PreC: A is a list of points.
    """
    d = 0
    n = len(A)
    for k in range(n-1):
        d = d + Dist(A[k],A[k+1])
    d = d + Dist(A[n-1],A[0])
    return d
```

## More on Copying Objects

A subtle issue is involved if you try to copy objects that have attributes that are objects themselves.

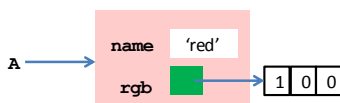
## More on Copying Objects

To illustrate consider this class

```
class MyColor:
    """
    Attributes:
        rgb: length-3 float list
        name: str
    """
    def __init__(self,rgb,name):
        self.rgb = rgb
        self.name = name
```

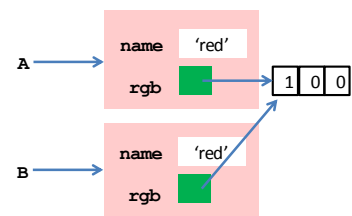
## More on Copying Objects

```
>>> A = MyColor([1,0,0], 'red')
```



## More on Copying Objects

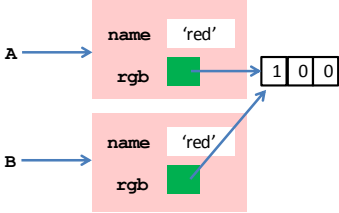
```
>>> B = copy(A)
```



### More on Copying Objects

```
>>> B = copy(A)
```

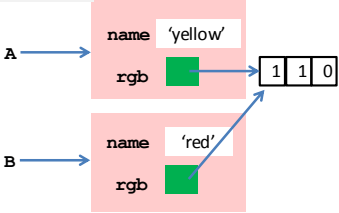
Now let's make A yellow



### More on Copying Objects

```
>>> A.rgb[1]=1
>>> A.name = 'yellow'
```

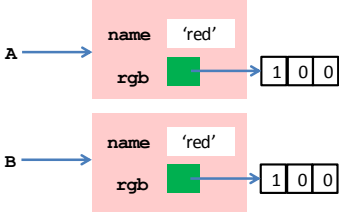
Unintended Effect  
B.Rgb refers to a yellow triple



### More on Copying Objects

```
>>> B = deepcopy(A)
```

deepcopy copies everything



### Summary: Base Types vs Classes

Base Types	Classes
Built into Python	Defined in Modules
Instances are values	Instances are objects
Instantiate w/ Literals	Instantiate w/ constructors
Immutable	Mutable