Contents

Python Objects

November 15, 2015

1 Python Objects

1.1 Learning Objectives:

This is an extremely long section, but there are big chunks of code here, so don't be too worried. Also, be sure to look at the original iPython code here. This requires that you've installed Jupyter, but it might be helpful for you to install it anyway (make sure you install it for Python 2.7). This helps you keep text and code together in one bundle. It's made writing these course notes way easier!

1. Understand what a class object is
    a. How do we declare a class
    b. Why do we use classes?
    c. How do we assign values to classes?
    d. How do we assign methods to classes?
2. Understand object/class inheritance and parent/child relationships
    a. Understand how composition is different from inheritance
3. Understand how to prevent direct manipulation of class attributes

2 The What and Why of Object Oriented Programming (OOP)

Reference: Think, p. 161-182

2.1 Objects

We talked about objects in Python earlier. At the simplest level, objects are things that include both data and methods. , but they are more than that. They provide a way to mimic real-world entities that have both state and behaviors. For example: Bank accounts have a balance and offer ways to add, subtract and query the amount. Farms have an owner, size, soil moisture status, crop condition and a host of other properties, but they also grow crops Lakes have area, volume, pH, etc., and also grow and shrink with inputs and losses Each individual bank account, farm, and lake is unique, but also has much in common with other members of the class. (If not, the "class" doesn't have much meaning.)

OOP vocabulary distinguishes between a class, which is an abstraction (e.g., the concept of a lake), and instances, or individual objects within the class (e.g., a particular lake).

2.2 Why OOP? What are the benefits?

1. **Encapsulation -** bundling and hiding. Many data objects have complex data structures, and particular methods associated with those data structures. If we didn't have "objects", the user would have to work through these methods each time they wanted to do some analysis. With objects, the user only needs to know how to use the object, not what happens inside. Think of a car's automatic transmission. It is a terribly complicated device, but the internals are encapsulated; almost anybody can use it because the interface is so simple.

2. **Polymorphism -** same method name might be found in multiple classes, but have different operations appropriate for the class. We've seen examples of this. The + operator means one thing for numbers but another for strings. Or, consider that the sort() method works very differently for numbers and strings.

3. **Inheritance -** new classes can be based on existing classes. You don't have to rewrite the code each time, and you can build more complex classes on simpler sets of classes.

2.3 Object Oriented Programming in Python

Python does not require OOP, but it supports OOP. As more "serious" programmers have taken up Python, and as it has been applied to more complex problems, object-orientation has grown with successive releases.

Python's idea of OOP is at odds with some basic principles of OOP as seen in other languages. True believers offer explanations of why the "Pythonic way" (see The Zen of Python) allows such things, but those explanations leave many people unconvinced. If you come to Python from another OO language you might be surprised by some aspects of OOP in Python.

3 Basics of Class Definition and Use

3.0.1 Defining Classes.

A class is defined in much the same way as a function, but we do not use `def`. Functions are defined with `def`, objects are defined with `class`:

```
In [ ]: class ClassName:
          Statement1
          Statement2
          Statement3
          .
          .
          .
          Statementn
```

Here `ClassName` is the name of the new class. `ClassName` follows the rules for variable and function names we've defined earlier. Standard practice is to start classes with capital letter and use CamelCase (see the Style Guide entry for class names). All statements within the class definition are indented by the same amount, just like in a function.

Examples of empty classes, just to show examples of legal class names (pass is a function that does nothing):

```
In [10]: class City:
               pass            #pass is a placeholding do-nothing statement

         class polygon:       #first letter p ok, but P is better
              pass

         class Point3d:      #numbers allowed in a class name
              pass

         class Water_well:    #underscores allowed
              pass

         class StreamReach:  #camel naming
              pass
```

3.1 Using Classes

3.1.1 Creating Instances.

The above statements merely define the classes, no Point3d or City objects exist. To use a class, we create one or more instances. Instantiation generates new members of the general class. Simplest syntax is:

```
In [ ]: ClassInstance = ClassName()
```

Where `classInstance` is a variable name and `ClassName` is a previously defined class. We've already defined some classes above (even if they are 'empty' in some sense), so we can assign these classes to variables:

```
In [12]: Dane       = polygon()
         burtsWell  = Water_well()
         madison    = City() #an instance of the City class
         losAngeles = City() #another City, different than madison
```

The reason we try to adhere to the style guide is that it makes things easier to review down the road. If we know that our style guide tells us to capitalize classes and leave function names lower case then we can tell the difference between a variable assigned from a class (as above) and one that gets its value from a function.

Instances can also be created by assignment and copying:

```
In [13]: import copy

         p1 = Point3d()
         p2 = p1                  #p2 is an alias of p1

         p3 = copy.deepcopy(p1)   #p3 is a new instance
```

3.2 Data Attribute Assignment and Use:

Objects have properties (e.g., hair color, bank balance, number of enrolled students) reflecting the status of the particular class instance. In Python an object's data attributes – like its methods – are accessed using the familiar "dot" notation we've seen earlier. Examples:

3.2.1 Assignment:

```
In [21]: import math

         losAngeles = City()
         losAngeles.latitude  = 34.05
         losAngeles.longitude = -118.24

         burtsWell = Water_well()
         burtsWell.depth      = 257
         burtsWell.contractor = "Ekhardt"
         burtsWell.pH         = 8.3

         burtsWell.pH   = "alkaline" #now depth is a string

         Water_well.depth     = 257 #error "depth" not in class def

         # Can use values in other statements:
```

```
print losAngeles.latitude, losAngeles.longitude

logDepth = math.log(float(burtsWell.depth))

print losAngeles    #can we print the entire object?
```

```
34.05 -118.24
<__main__.City instance at 0x02D889E0>
```

Note the last output that looks really weird: `<__main__.City instance at 0x02D889E0>`

This is what the class output looks like. Even though we've assigned all sorts of variables into the class, they're encapsulated, so when we try to print out the class we don't see anything. We'll see how to generate special print methods for classes in a bit.

Important Point:

Assignment statements can change values or create new class members. Assignment statements are fine for changing values, but not the best way to create properties that all instances must have. E.g., if all wells must have a `depth` attribute, there is a better way to create that property (see **init** below).

4 Class Methods

Behavior is another attribute of classes, known as "methods" in OO languages. You have seen these before, e.g. lists have their `sort()` method.

Methods are like functions, but are defined *within* the class are therefore part of the class and not part of the global environment.

Syntax:

```
In [ ]: class  ClassName:
          def method1(self, parm1, parm2, ...):
            statement 1
            statement 2
            statement 3
            .
            .
            .
          def method2(self, parm1, parm2, ...):
            statement 1
            statement 2
            .
            .
            .
```

Where `method1()` and `method2()` are methods belonging to the class `ClassName`. The first parameter (`self`) is always required, but `parm1`, `parm2`, ... are optional, and can be called whatever you want, depending on what you want the method to do.

4.1 Example 12.1 (Example12.01.py):

```
In [22]: class City:
           def printLatitude(self):
             print "%s is at latitude %.2f" %(self.name,self.lat)
```

```
LA = City()
LA.name = "Los Angeles"
LA.lat  = 33.93

Mad = City()
Mad.name = "Madison"
Mad.lat = 43.08

LA.printLatitude()
Mad.printLatitude()
```

```
Los Angeles is at latitude 33.93
Madison is at latitude 43.08
```

4.2 Exercise 12.1:

Add a method to the city example that prints the longitude.

Go to Exercise 12.1

`self` refers to a particular instance (like "this" in Java, C++, etc.). The word "self" appears in the definition, but not in the invocation of the method.

But why is self needed? Two examples:

4.3 Example 12.2 (Example12.02.py):

```
In [23]: class BadCity:
           def printLatitude(self):
             print "lat is",lat

         lat = 999  # define a global variable called 'lat'

         LA = BadCity()
         LA.name = "Los Angeles"
         LA.lat = 33.93

         Mad = BadCity()
         Mad.name = "Madison"
         Mad.lat  = 43.08

         LA.printLatitude()
         Mad.printLatitude()
```

```
lat is 999
lat is 999
```

Why did this happen? When we defined `printLatitude` we used the variable name `lat`, and this was associated with a global variable we subsequently defined. We use `self` to clearly indicate that the method is using attribute of the class itself.

Another Example: Suppose we want a method that computes and prints the distance north from another city. We will call the method like so:

```
In [ ]: LA.printMilesNorth(Mad)
```

If we want this to work, then the method must be defined in the class:

4.4 Example 12.3 (Example12.03.py):

```
In [34]: class City:
           def printLatitude(self):
             print "%s is at latitude %.2f" (self.name,self.lat)
           def printMilesNorth(self, othercity):
             dist   = (self.lat - othercity.lat) * 24859./360.
             format = "%s is %.1f miles north of %s"
             values = (self.name, dist, othercity.name)
             print format % values

         LA = City()
         LA.name = "Los Angeles"
         LA.lat = 33.93

         Mad = City()
         Mad.name = "Madison"
         Mad.lat  = 43.08

         Mad.printMilesNorth(LA)
         LA.printMilesNorth(Mad)
```

```
Madison is 631.8 miles north of Los Angeles
Los Angeles is -631.8 miles north of Madison
```

We see `self` was needed to distinguish between the two latitudes, and, as we saw before, you need to differentiate values from within the class from global variables. (Use of "self" is actually just a convention; other variable names can be used but this is not recommended)

4.5 Exercise 12.2:

1. add a new method to the City class called printLatDiff which prints the latitude difference (absolute value) between two cities.
2. Define a separate function outside of any class to accomplish the same functionality as printLatDiff . The function definition should be:

   def printLatDiff(city1,city2):

3. Now suppose at some point in the future you decide to store a City's latitude and longitude as a tuple. Is anything more than a change to the class definition required? Would the new function created in b) need to change?

Go to Exercise 12.2

5 The `__init__` Method (aka constructor)

An *init*ialization method is a special method that is called when an instance is created. In Python the name of this method is `__init__`. This is **two** underscores _ _ followed by `init`, followed by two more underscores _ _ with no blanks.

We use `__init__` to assign default values to data attributes, create any essential variables, and possibly ensure every instance has its own set of values. This way we don't have to go through the process

Syntax:

```
In [ ]: def __init__(self, parm1=val1, parm2=val2, . . .):
```

Where `val1`, `val2`, &cetera are default values assigned to `parm1` and `parm2`, respectively. Setting default values is optional but strongly recommended - you need a good reason not to use default values (even if it's a nonsense value)!

5.1 Example 12.4:

```
In [36]: class City:
             def __init__(self, name='n/a', lat = -999, lon = -999):
                 self.name = name
                 self.lat  = lat
                 self.lon  = lon

             def printLatitude(self):
                 print "%s is at latitude %.2f" % (self.name,self.lat)

             def printMilesNorth(self,city):
                 dist = (self.lat-city.lat)*24859./360.
                 Fmt  = "%s is %.1f miles north of %s"
                 vals = (self.name, dist, city.name)
                 print fmt % vals

         #create LA and over-ride all default values
         LA = City(name = 'Los Angeles', lat=33.93, lon=-120.)

         #create Mad and over-ride two default values
         Mad = City(name='Madison', lat=43.08)

         print 'longitude of LA is',LA.lon
         print 'longitude of Mad is',Mad.lon
```

```
longitude of LA is -120.0
longitude of Mad is -999
```

Alternative 1 - same result, since the arguments are in the same order as they're declared in the `__init__`:

```
In [37]: LA  = City("Los Angeles", 33.93, -120.)
         Mad = City("Madison", 43.08)
         print 'longitude of LA is',LA.lon
         print 'longitude of Mad is',Mad.lon
```

```
longitude of LA is -120.0
longitude of Mad is -999
```

Alternative 2 (same result):

```
In [39]: LA  = City()
         LA.name = "Los Angeles"
         LA.lat  = 43.08
         LA.lon  = -120.

         Mad = City()
         Mad.name = "Madison"
         Mad.lat = 43.08
```

```
        print 'longitude of LA is',LA.lon
        print 'longitude of Mad is',Mad.lon
```

```
longitude of LA is -120.0
longitude of Mad is -999
```

`__init__()` is not especially useful in this example, but we get a sense of some of the reasons it might be useful:

1. `__init__` is a good place to establish essential data attributes and set default values if not provided during initialization
2. The `__init()` statement can be used to test for valid input values (Lon and Lat in this case)

Be careful that you're using the right assignment variable names, and taking care to watch what gets assigned to `self.` and what is refering to the variables being passed into the class. It's easy to get errors raised, telling you a class attribute doesn't exist, because you failed to assign it as `self.whatever` in the `__init__()` methond.

5.2 Exercise 12.3:

Modify the City class `__init__()` to test if the input lon and lat values are valid. If not the function should assign the default value.

Go to Exercise 12.3

6 Inheritance

A selling point of OOP is code re-use; this is partially achieved though inheritance

Basic Idea: Create descendants from a base class (or classes)

Ancestors are identified in sub-class definition. Descendants have all of their ancestor methods. Can also ensure all ancestor data attributes are inherited by descendant.

Vocabulary variable but fairly obvious. All of these make the same distinction:

ancestor vs. descendant parent vs. child base class vs. sub-class

The syntax is trivial, we simply name any ancestors in a class parameter list:

```
In [ ]: class ChildClassName(Parent1, Parent2, . . .):
            statement1
            statement2
            statement3
```

where `ChildClassName` is the name of the new class. This will derive from classes `Parent1`, `Parent2`, ... In practice there will usually be only one parent class. What this means is that the attributes and methods associated with the Parent Class become attributes of the Child Class, along with any new attributes that are declared within the child class. As shown above it is possible to inherit from multiple parent classes. This is not the case in many programming languages, and requires a relatively complex system of "method resolution". For example, if `Parent1` and `Parent2` both contain a method `sum`, that varies between the two classes, then `ChildClassName` needs to inherit a single `sum` method. This is documented well for Python 2.3 here, I don't know of a similar document for v2.7.

What if there's no parent? That is, what if we are defining top-level class, and therefore no base or ancestor class exists? Two options:

1. Don't give a parent name, as in all the examples above:

```
In [ ]: class ClassName:
            pass
```

2. Use the keyword "object" as the parent:

```
In [ ]: class ClassName(object):
            pass
```

For example:

```
In [ ]: class Water_well(object):
            pass
```

Option 1 defines "old-style" classes. Option 2 defines "new-style" classes. The distinction doesn't matter to us in Geog 378, and we will use old-style classes througout.

Syntax Examples:

```
In [ ]: class Furniture:
          statement1
          statement2

        class Chair(Furniture): #subclass of Furniture
          statement1
          statement2

        class RockingChair(Chair):#descendant of Chair & Furniture
          statement1
          statement2

        class FoldingChair(Chair):#descendant of Chair & Furniture
          statement1
          statement2

        class Shape:
          pass

        class Rectangle(Shape):
          pass

        class Polygon(Shape,Arc): #descendant of Shape and Arc
          pass
```

6.1 Example 12.5 (Example12.05.py):

```
In [40]: class Base:
             def __init__(self,A=2,B=3):
                 self.a = A
                 self.b = B

             def ApB(self):
                 return self.a + self.b
```

```
        class SubClass(Base):
            pass

        child = SubClass()
        print "Child a,b,ApB: ",child.a,  child.b,  child.ApB()
```

```
Child a,b,ApB:  2 3 5
```

Over-rides: When a method is defined in the child that also exists in the parent, the child's method "replaces" the parent's method. That is, the child's method gets used in the child. The parent version is not deleted, but it is not used when the child method is called. Example:

6.2 Example 12.6 (Example12.06.py):

Redefining & replacing a Parent method within a child class.

```
In [42]: class Base:
             def __init__(self,A=2,B=3):
                 self.a = A
                 self.b = B
             def ApB(self):
                 return self.a + self.b

         class SubClass(Base):
             def ApB(self): #over-rides parent's definition
                 return self.a * self.b

         parent = Base()
         child  = SubClass()
         print "Parent a,b,ApB:",parent.a, parent.b, parent.ApB()
         print "Child  a,b,ApB:",child.a,  child.b,  child.ApB()
```

```
Parent a,b,ApB: 2 3 5
Child  a,b,ApB: 2 3 6
```

See `ApB()` in the child class `SubClass` has replaced the parent definition.

6.3 Exercise 12.4:

Write a method for the `Base` class shown above that is called `equals(self, testValue)`: If `a` and `testValue` are equal, return `True`; otherwise, return `False`. In the child class, write a method to override the Parent's `equals`: If either `a` or `b` is equal to the test value, return `True`; otherwise, return `False`.

Go to Exercise 12.4

Just as the child can over write methods, if `__init__()` exists in a child the parents' `__init__()` is over-ridden and therefore not invoked. To illustrate:

6.4 Example 12.7 (Example12.07.py):

Overwriting the `__init__()` method here will prevent the child class `SubClass` from assigning a value to `self.a` and `self.b`, even though these assignments are defined in the parent class `Base`.

```
In [43]: class Base:
             def __init__(self, A = 2, B = 3):
                 self.a = A
                 self.b = B

             def ApB(self):
                 return self.a + self.b

         class SubClass(Base):
             def __init__(self):
                 self.c = 4

             def ApB(self):
                 return self.a * self.b

         child = SubClass()

         print "child.c is",child.c
         print "child.a is", child.a
```

```
child.c is 4
child.a is
```

```
        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-43-204d330f8d45> in <module>()
         18
         19 print "child.c is",child.c
    ---> 20 print "child.a is",child.a


        AttributeError: SubClass instance has no attribute 'a'
```

This error occurs because we don't have an `a` in the `child` object.

6.5 Example 12.8 (Example12.08.py):

How do we add to the parent's initialization method without having to rewrite it completely? We explicitly call the parent method's `__init__`.

```
In [45]: class Base:
             def __init__(self,A=2,B=3):
                 self.a = A
                 self.b = B

             def ApB(self):
                 return self.a + self.b
```

```
        class SubClass(Base):
            def __init__(self):
                Base.__init__(self)   #call parent's initializer
                self.c = 4
            def ApB(self): #over-rides Base definition
                return self.a * self.b


        child = SubClass()
        print "child.c is",child.c
        print "child.a is",child.a
```

```
child.c is 4
child.a is 2
```

6.6 A more complete example:

Consider a more complete example for the analysis of spatial data. To accurately represent the spatial data we need a variety of vector-type spatial units: cities, states, land parcels, park areas. All of these features, no matter their type, have common attributes and methods:

1. Common data attributes:
   - area
   - center
   - a set of vertices defining a perimeter (values differ, but every instance has these values)
2. Common methods: Regardless of a feature's type, we need methods to:
   1. find the center
   2. compute the area
   3. test for point inclusion

Each type of feature also has distinct attributes: e.g., states have a capital; parcels have an owner, &cetera. We could consider all of these objects as a particular type of polygon. Class hierarchy:
Classes needed:

- **Point**: objects with a location (x,y) but nothing else. Needed for center and perimeter.
- **Polygon** the base spatial unit class: Has a center point, list of perimeter points, an area value, and the above common methods:
  - Compute Center, (xc, yc) : xc = Σ xi/n, yc = Σ yi/n
  - Compute Area, A = $|\frac{1}{2} \sum$ (xi yi+1 – xi+1 yi)|
  - Point in Polygon Test for point (x,y) : Extend ray from (x,y) to the right, count number of times ray crosses the perimeter. If it's zero times then the point is outside the polygon. If it crosses an odd number of times then (x,y) is inside. If it's an even number of times, then it's outside.

In what follows, don't worry about the programming details of these methods. The important thing is to understand how they fit in the class.
3 Derived Classes:

- City Class: Polygon with a name and population
- Parcel Class: Polygon with an owner and assessed value
- County Class: Polygon with a name, seat, and count of incorporated cities in the county

Important Point: Assignment statements can change values or create new class members. Assignment statements are fine for changing values, but not the best way to create properties that all instances must have. E.g., if all wells must have a depth, there is a better way to create that property (see `__init__` below).

6.7 Example 12.9 (Example12.09.py):

This is a very long program. The code for this program and the other examples is included in a zipped fille associated with this lecture. The program output interspersed and shown in red.

Here we use -999 and -999 as default x and y values. It is dangerous to have default values that could be actual values. For example, [0, 0] is a real place on the globe, just south of Ghana. Even if it's unlikely to be a city the fact that it's a remotely reasonable value may result in problem if you're looking at a large number of cities and you're not paying attention.

```
In [72]: class Point:
             def __init__(self, x = -999, y = -999):
                 self.x = x
                 self.y = y

         class Polygon:
             #could be initialized with set of points
             def __init__(self, pts = [Point(-999,-999)]):
                 self.pts    = pts
                 self.computeCenter()
                 self.computeArea()

             def computeCenter(self):
                 'Find the average (x,y) of perimeter'
                 n = len(self.pts)
                 self.center = Point(0,0)
                 if n > 0:
                     for point in self.pts:
                         self.center.x += point.x
                         self.center.y += point.y

                     self.center.x /= float(n)
                     self.center.y /= float(n)

                 return self.center

             def computeArea(self):
                 'Find area defined by the perimeter'
                 n = len(self.pts)
                 self.area = 0
                 if n > 2:
                     for i in xrange(-1,n-1):

                         xi = self.pts[i].x
                         yi = self.pts[i].y

                         xj = self.pts[i+1].x
                         yj = self.pts[i+1].y

                         self.area = self.area + xi*yj - xj*yi

                     self.area = abs(self.area)/2.

                 return self.area

             def surroundsPt(self, p):
```

```
        'Point in Polygon test using ray count method'
        n = len(self.pts)

        # You can't have a polygon with less than three points!
        if n < 3: return false

        # p is the point we're testing:
        x = p.x
        y = p.y

        crossingCount = 0

        for i in xrange(-1,n-1):
          xi = self.pts[i].x
          yi = self.pts[i].y

          xj = self.pts[i+1].x
          yj = self.pts[i+1].y

          if((yi <= y and y < yj) or (yj <= y and y < yi)):

             xCross = float(xj-xi)*(y-yi)/(yj-yi) + xi

             if  (x < xCross): crossingCount += 1

        #odd crossingCount means (x,y) is inside
        if crossingCount % 2 == 1: return True
        else                     : return False


class City(Polygon):
    def __init__(self, name = '', population = 0, pts = [Point(-999,-999)]):
        Polygon.__init__(self,pts)
        self.name = name
        self.pop  = population


class Parcel(Polygon):
    def __init__(self, owner = '', number = 0, assesment = 0, pts = [Point(-999,-999)]):
        Polygon.__init__(self,pts)
        self.owner      = owner
        self.number     = number
        self.assessment = assessment


class County(Polygon):
    def __init__(self,name = '',seat = '', cityCount = 0, pts = [Point(-999,-999)]):
        Polygon.__init__(self, pts)
        self.name      = name
        self.seat      = seat
        self.cityCount = cityCount
        Polygon.__init__(self)

triangle = Polygon([Point(20,20),Point(40,40),Point(60,20)])
```

```
        print 'triangle center: [%.2f,%.2f]' %(triangle.center.x, triangle.center.y)
        print 'triangle area: {:9.2f}'.format(triangle.area)
```

```
triangle center: [40.00,26.67]
triangle area:    400.00
```

```
In [73]: # Note, this could be par of an __repr__ in the City definition:
         a_city = City(name='c1',population = 200,pts = [Point(20,20),Point(40,40),Point(60,20)])

         print "a_city's properties:"
         print '   name: ' + a_city.name
         print '    pop: %.2f' % a_city.pop
         print '   area: %.2f' % a_city.area
         print ' center: [%.2f,%.2f]' % (a_city.center.x,a_city.center.y)
```

```
a_city's properties:
   name: c1
    pop: 200.00
   area: 400.00
 center: [40.00,26.67]
```

```
In [74]: bTown = City('Boxton, WI',117)

         print "City with default center and area"
         vals = (bTown.name, bTown.pop, \
                 bTown.center.x, bTown.center.y, bTown.area)
         fmt = "%s (population %d) is centered on (%.1f,%.1f), its area is %.1f\n"

         print fmt % vals

         #  Here we see that Boxton gets a wonky initial value for X and Y because
         #  we've defined default x and y values throughout.
         #  If it were [0,0] we might run into problems, but the value
         #  is so large we're unlikely to let this slip by.
```

```
City with default center and area
Boxton, WI (population 117) is centered on (-999.0,-999.0), its area is 0.0
```

```
In [75]: bTown.pts = [ Point(0,0), Point(0,1), Point(2,1), Point(2,0)]
         bTown.computeArea()
         bTown.computeCenter()

         vals = (bTown.name, bTown.pop, \
                 bTown.center.x, bTown.center.y, bTown.area)

         print "City with revised attributes"
         print fmt % vals
```

```
City with revised attributes
Boxton, WI (population 117) is centered on (1.0,0.5), its area is 2.0
```

```
In [76]: print 'Is (.1,.1) inside boxTown?',bTown.surroundsPt(Point(.1,.1))
         print 'Is   (3,3) inside boxTown?',bTown.surroundsPt(Point(3,3))
```

```
Is (.1,.1) inside boxTown? True
Is   (3,3) inside boxTown? False
```

```
In [77]: triCo = County(name='Tri County')
         triCo.pts = [Point(20,20),Point(40,40),Point(80,17)]

         print 'Does tri surround (50.,50.)?',triCo.surroundsPt(Point(50.,50))
```

```
Does tri surround (50.,50.)? False
```

Note:

1. Inheritance allowed us to re-use code (center, area, point inclusion)
2. We could easily replace the point inclusion test with another. Only one change would be needed. SeeExample12.09_Version2.py.

6.8 Exercise 12.5:

Design a simple mapping system class hierarchy in Python. Call the base class Shape and the child classes are Point, Polyline and Polygon. The base class has a method draw and each child class has its own draw method. You don't need to implement the draw method. Just provide the program outline using pass statements as placeholders. Feel free to add class members and even other classes you think are necessary.

Go to Exercise 12.5

7 Composition

Just like in the real world, Python objects can contain other objects. That is, objects can be nested. We saw this with the `Polygon` class above, which contained a list of points. These were assumed to be Point objects. This is composition, different than inheritance. Consider a slightly more complicated example:

7.1 Example 12.10

This example assumes Polygon and City classes are defined as above.

```
In [79]: class State(Polygon):
             def __init__(self,name = '',population=0):
                 self.name    = name
                 self.pop     = population
                 self.capital = City(name='',population=0)
                 Polygon.__init__(self)

         MadTown = City(name='Madison', population = 245323)

         WI = State('Wisconsin',5536201)
         print 'WI name and pop:',WI.name,WI.pop
         print 'WI capital name and capital pop:', \
                WI.capital.name, WI.capital.pop

         WI.capital = MadTown;
```

```
print 'WI capital name and capital pop:', \
      WI.capital.name,WI.capital.pop
```

```
WI name and pop: Wisconsin 5536201
WI capital name and capital pop:  0
WI capital name and capital pop: Madison 245323
```

When `WI` was created, it got the default value for its capital, a city with no name and zero population. Later `MadTown` was assigned as the `capital` of `WI`, and so we see its name and population when we use `print`.

The importance of carefully working out the class hierarchy and composition cannot be overstated!

7.2 Exercise 12.6:

What do you think of the example hierarchy above, where the Polygon class is the parent of City, Parcel and County classes? Can you suggest another hierarchy that is closer to how we think of cities, counties and land parcels?

Go to Exercise 12.6

8 Data Hiding

OOP promises encapsulation, with an implication of data integrity. That is, users can't assign inappropriate values, or destroy essential data members. Other languages provide this by allowing members to be declared private: accessible only within the class. Python doesn't have this, and so the following is perfectly legal:

```
In [82]: class City:
             def __init__(self, name='n/a', lat = 0, lon = 0):
                 self.name = name

                 if -90  <= lat <= 90  : self.lat  = float(lat)
                 else                  : self.lat  = 0.0

                 if -180 <= lon <= 180 : self.lon  = float(lon)
                 else                  : self.lon  = 0.0



         #create a city with a valid location
         m = City(name='Madison',lat=43,lon=-89)

         #this clobbers the latitude with an invalid value
         m.lat = 999
```

Python programmers typically approach this issue by maintaining variables and methods within the class that non-members are supposed to ignore. In this example, the City class would have longitude and latitude variables that outsiders don't access. The convention is to use an underscore as the first character of such "private" members. Of course, these are not truly private; this is just an understanding between the person writing the class and the person using the class.

Rather than assign a value to such a variable, a user would call a "set" method. To obtain the value of a private variable a user calls a "get" method. See below how this would play out for the longitude and latitude of the City class:

8.1 Example 12.11 (Example12.11.py):

```
In [84]: class City:
             def __init__(self, name='n/a', lat = 0, lon = 0):
                self.name = name
                self.setLonLat(lon,lat)

             def setLonLat(self,lon,lat):
                # Note the underscores here:
                if -90  <= lat <= 90  : self._lat  = float(lat)
                if -180 <= lon <= 180 : self._lon  = float(lon)

             def getLonLat(self):
                return (self._lon,self._lat)

          #create a city with a valid location
          m = City(name='Madison',lat=43,lon=-89)

          #this leaves _lon and _lat unchanged:
          m.setLonLat(999,999)

          print 'The lon and lat of',m.name,'are',m.getLonLat()
```

```
The lon and lat of Madison are (-89.0, 43.0)
```

Explanation: The class needs valid values for longitude and latitude. It uses "private" variables for these named `_lon` and `_lat` respectively. All operations within the class would use these private versions. The class has a method to set values (`setLonLat`) and a method to deliver the current values (`getLonLat`) should a user wish to know them. The setter method will never assign an invalid value. (More robust code would use `try` and `except` blocks in `setLonLat` for added safety. It would also return a value to indicate whether or not the requested `lon` and `lat` changes were successful.)

8.2 Exercise 12.7:

Modify the example above so that the city name is also a "private" variable. Add setter and getter methods to access the name variable.

Go to Exercise 12.7

To recap: this business of underscore variables is very common in the Python world. For example, if you do `dir([])`, you will see many variables whose name begins with `_`. We don't touch those variables. There are also private *methods* we are not meant to use. Like the `_xxx` variables, such `_yyy` methods are technically accessible.

The practice of using "Set"-like and "Get"-like methods is also very common. The next topic will cover the use of GDAL from Python. We will often see "Get" methods used to acquire values from GDAL objects. This is the reason why: GDAL doesn't want us monkeying with its precious data in a non-standard way.