Object oriented programming with Python

Andrew Walker

andrew.walker@bris.ac.uk

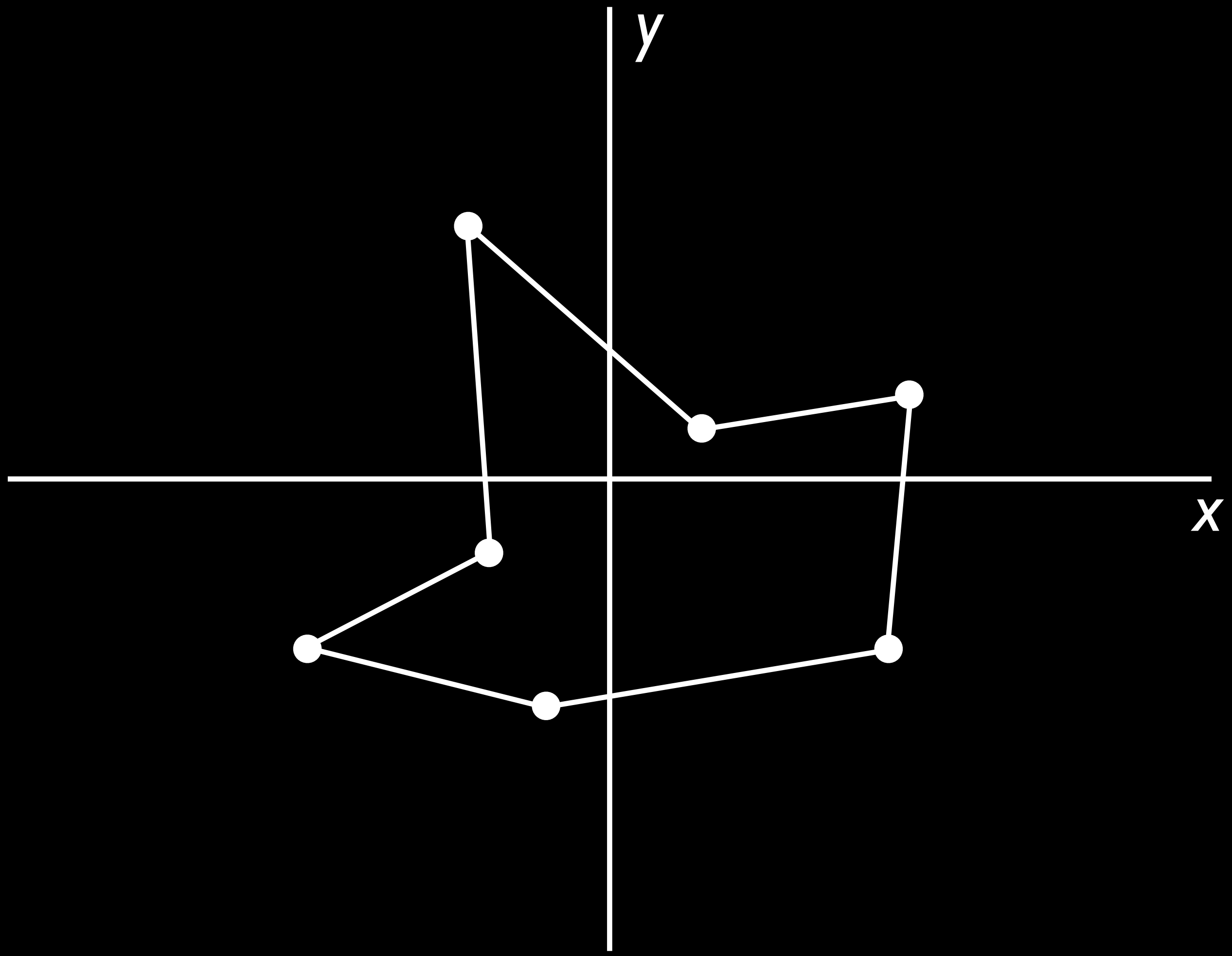
for Earth Scientists:
27 & 29 Sept. 2011

Why care about OO?

You can get a long way with Python without knowing anything about objects, but:

- Objects are in the language, and understanding them will make the syntax make more sense.
- Essentially all mainstream languages developed since ~1970 (C++, Java, JavaScript...) are OO and others have introduced OO (even Fortran).
- Objects can be useful in your code. They are often essential if you use other peoples code.
- The way objects work in Python is fairly standard and quite easy. If you need to learn about objects, Python is a good language to use.

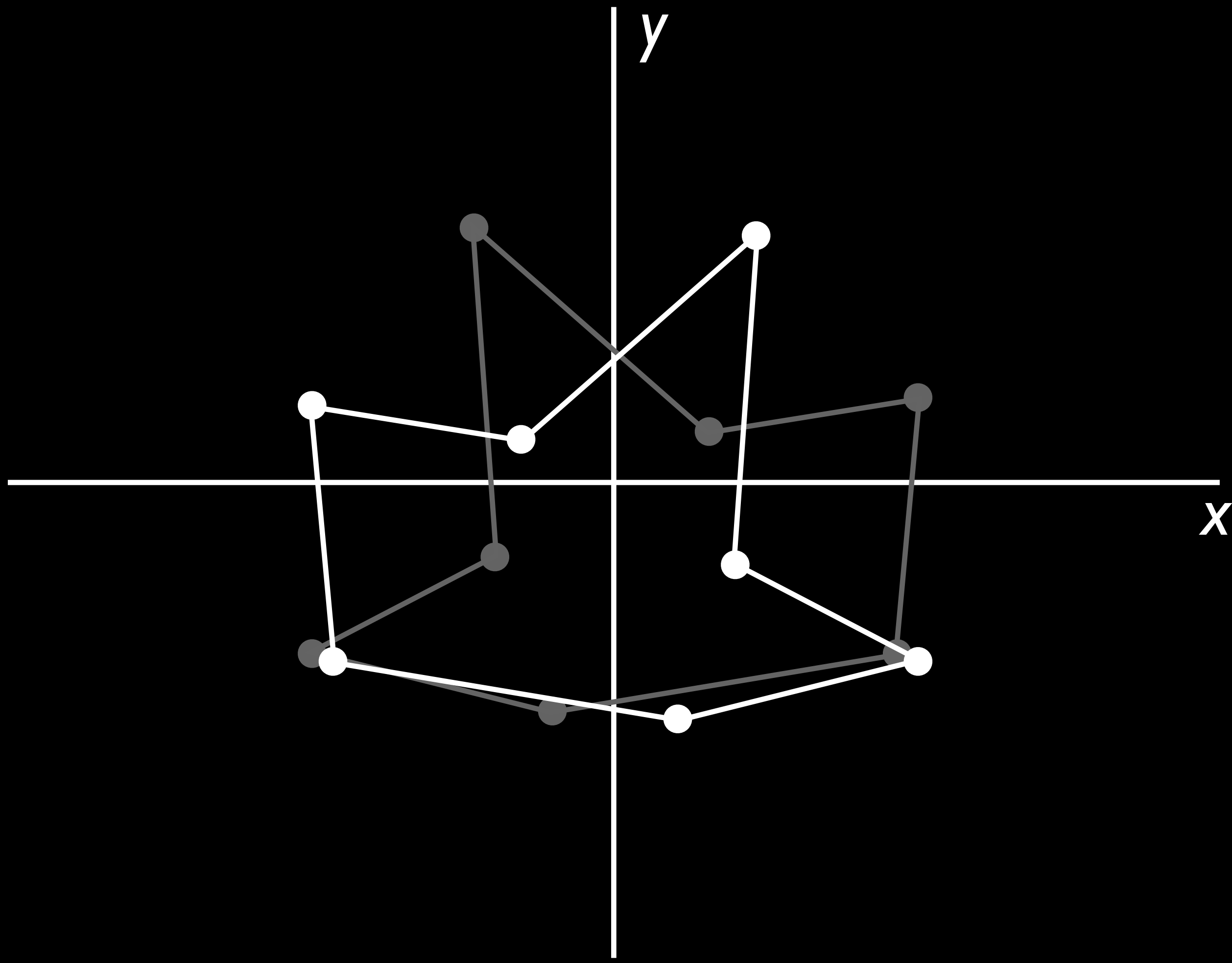
Imagine you need to write a program to deal with a shape on a plane...



Model a shape as two lists of points:

```
x_pts = [0.5, 1.2,
          1.1, ...]
y_pts = [0.5, 0.6,
         -0.9, ...]
```

Imagine you need to write a program to deal with a shape on a plane...



Model a shape as two lists of points:

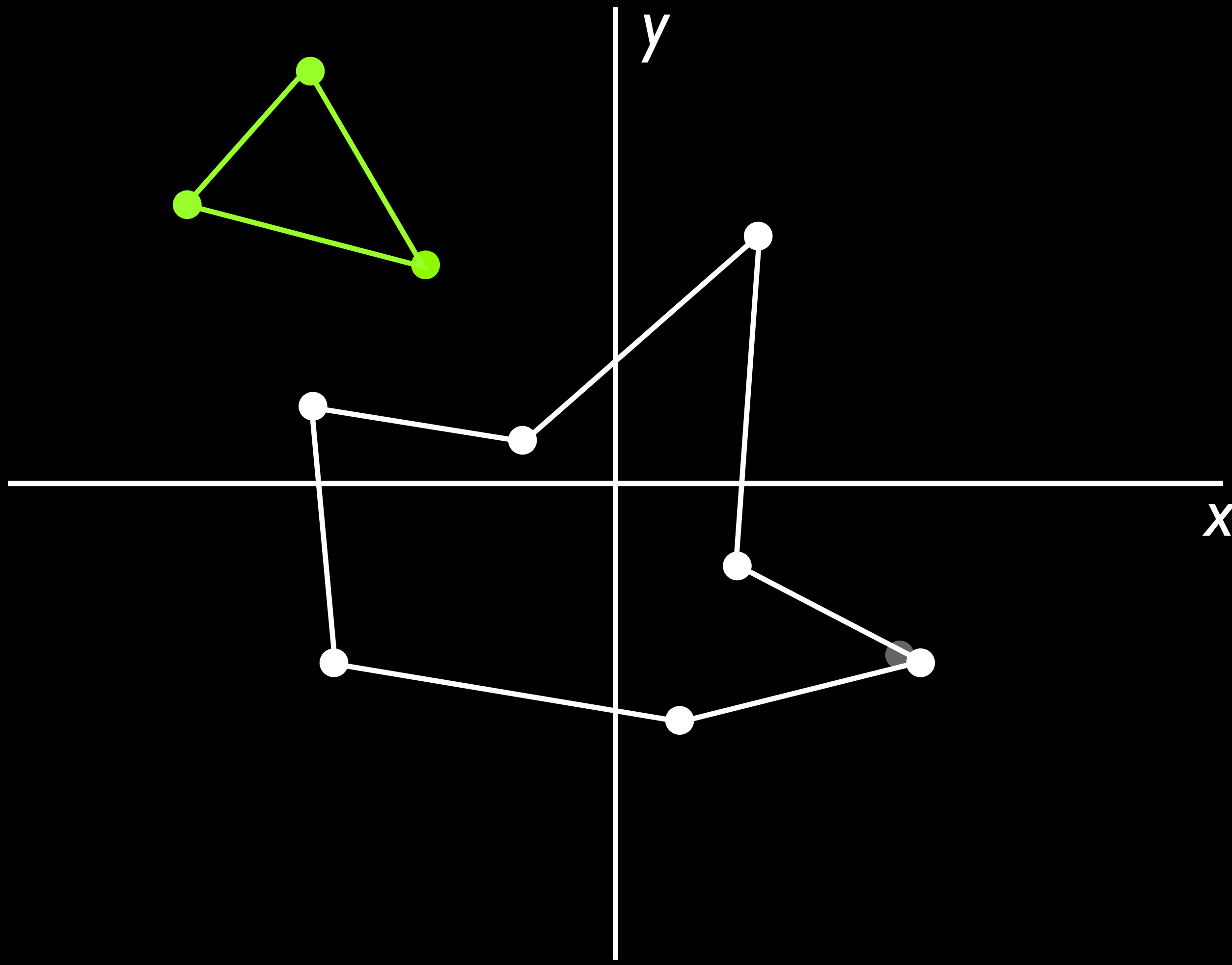
```
x_pts = [0.5, 1.2,
          1.1, ...]
y_pts = [0.5, 0.6,
         -0.9, ...]
```

Do things to the shape with a collection of functions:

```
def reflect_y(xs, ys):
    ...
    return (new_xs, new_ys)

x_pts, y_pts = reflect_y(x_pts,
                         y_pts)
```

Add another shape, if you have been careful, your functions still all work but you need more variables.

```
x_pts1 = [0.5, 1.2,
          1.1, ...]
y_pts1 = [0.5, 0.6,
         -0.9, ...]

x_pts2 = [-1.0,-1.5,
          -2.0]
y_pts2 = [1.1, 2.0,
          1.4]
```



Just remembering the details gets a little bit harder. And then a little bit harder. Eventually global state makes things very difficult indeed.

Objects are just a way
of organising data...

... which should make
code reuse easer and
enhance maintainability

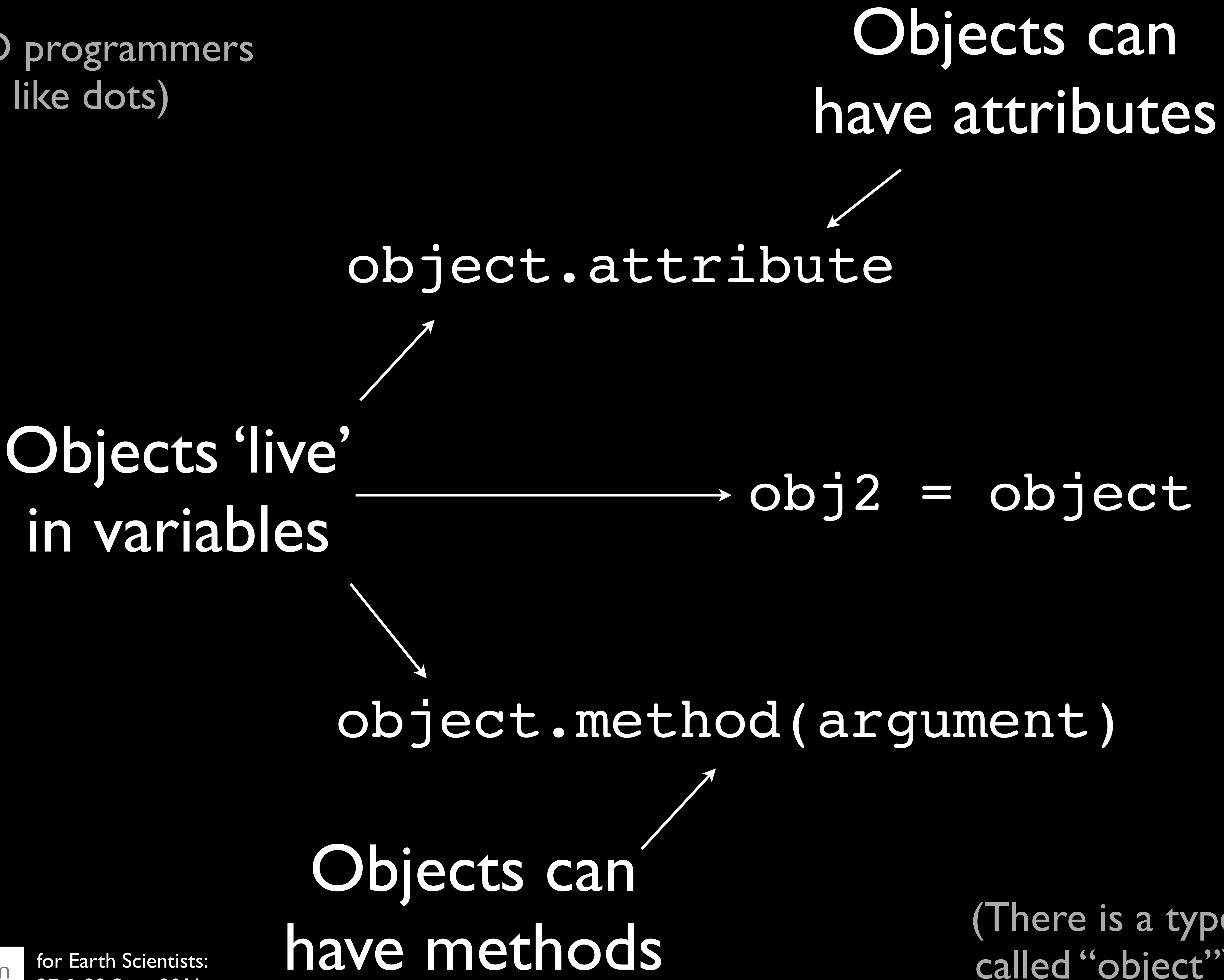
You already know what objects look like...

```
obj = open(“file”,‘r’)
for line in obj:
    ...
obj.close()
```

```
obj = 1+17j
obj.imag
```

... because in Python, everything is an object.

(OO programmers
like dots)



Objects can
have attributes

```
object.attribute
```

Objects 'live'
in variables

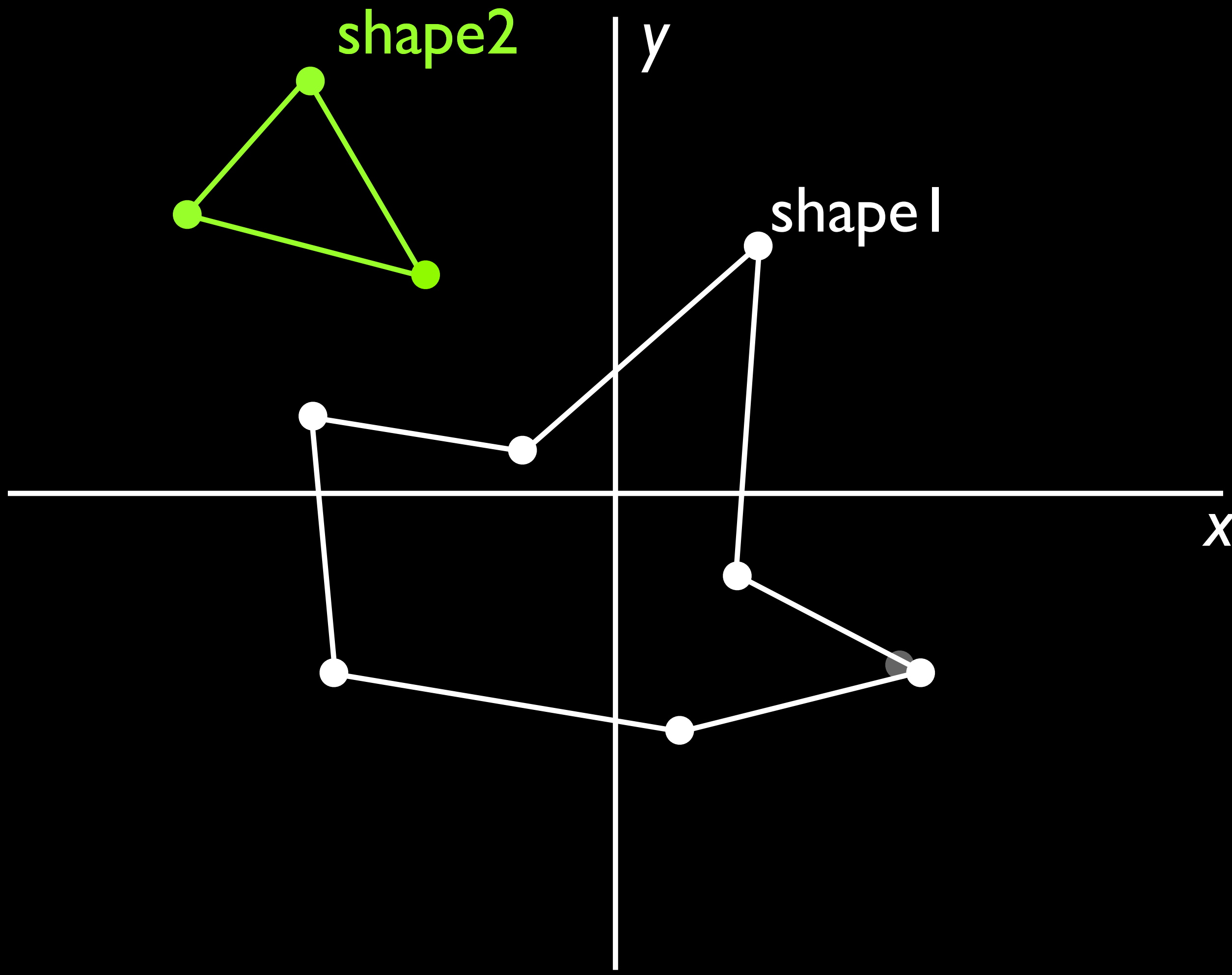
```
obj2 = object
```

```
object.method(argument)
```

Objects can
have methods

(There is a type
called "object")

Instead of keeping lists of points, make the shapes objects:

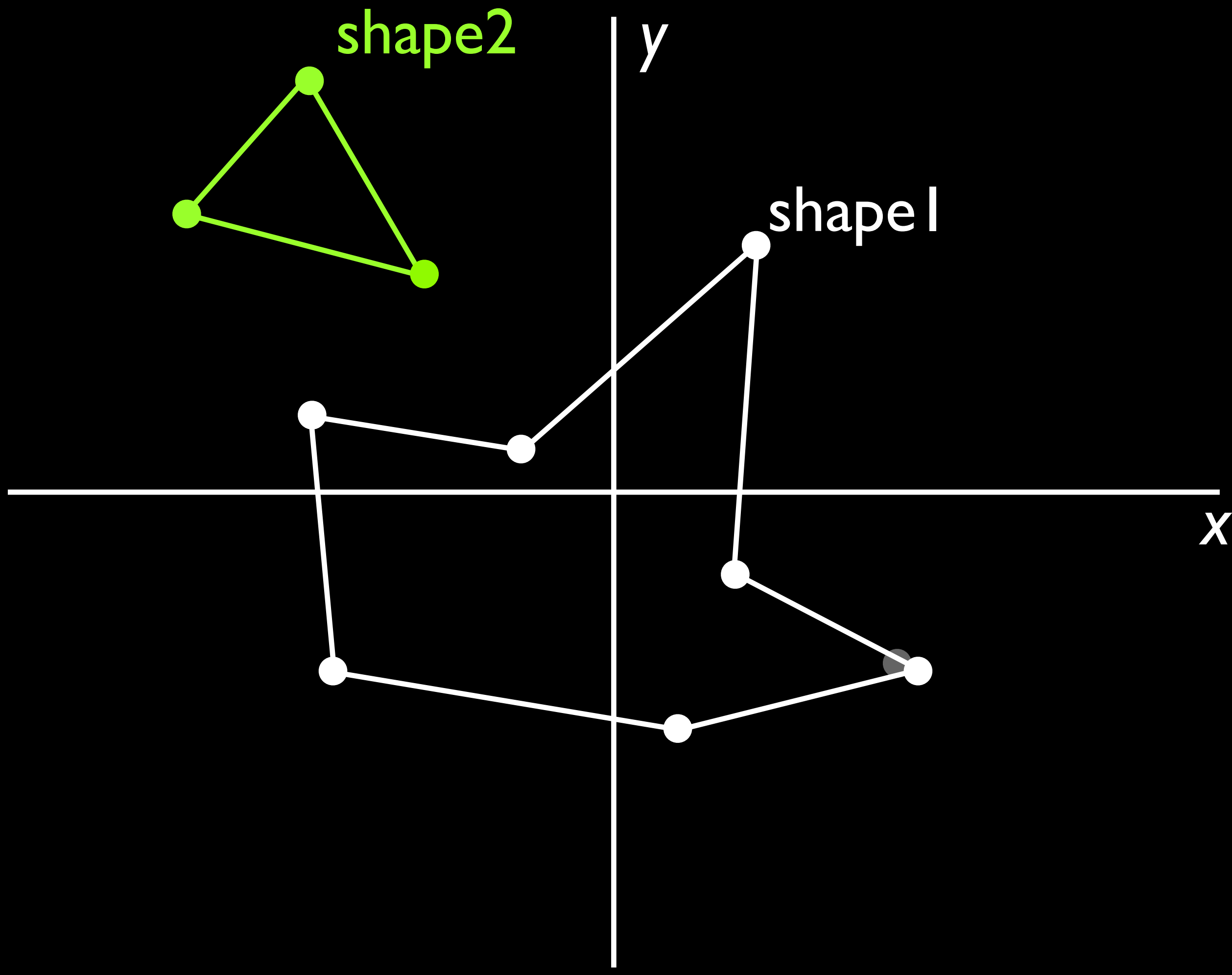


```
shape1 = Shape([0.5,
         1.2, ...],[0.5,
         0.6, ...])

shape2 = Shape([-1.0,
         -1.5,-2.0],[1.1,
         2.0,1.4])
```

Here `Shape` the name of a class of objects, `shape1` and `shape2` are instances of the class. We say `shape1` **is a** `Shape`. The **"is a"** relationship is key in OO design. The capitalisation of classes is a Python convention.

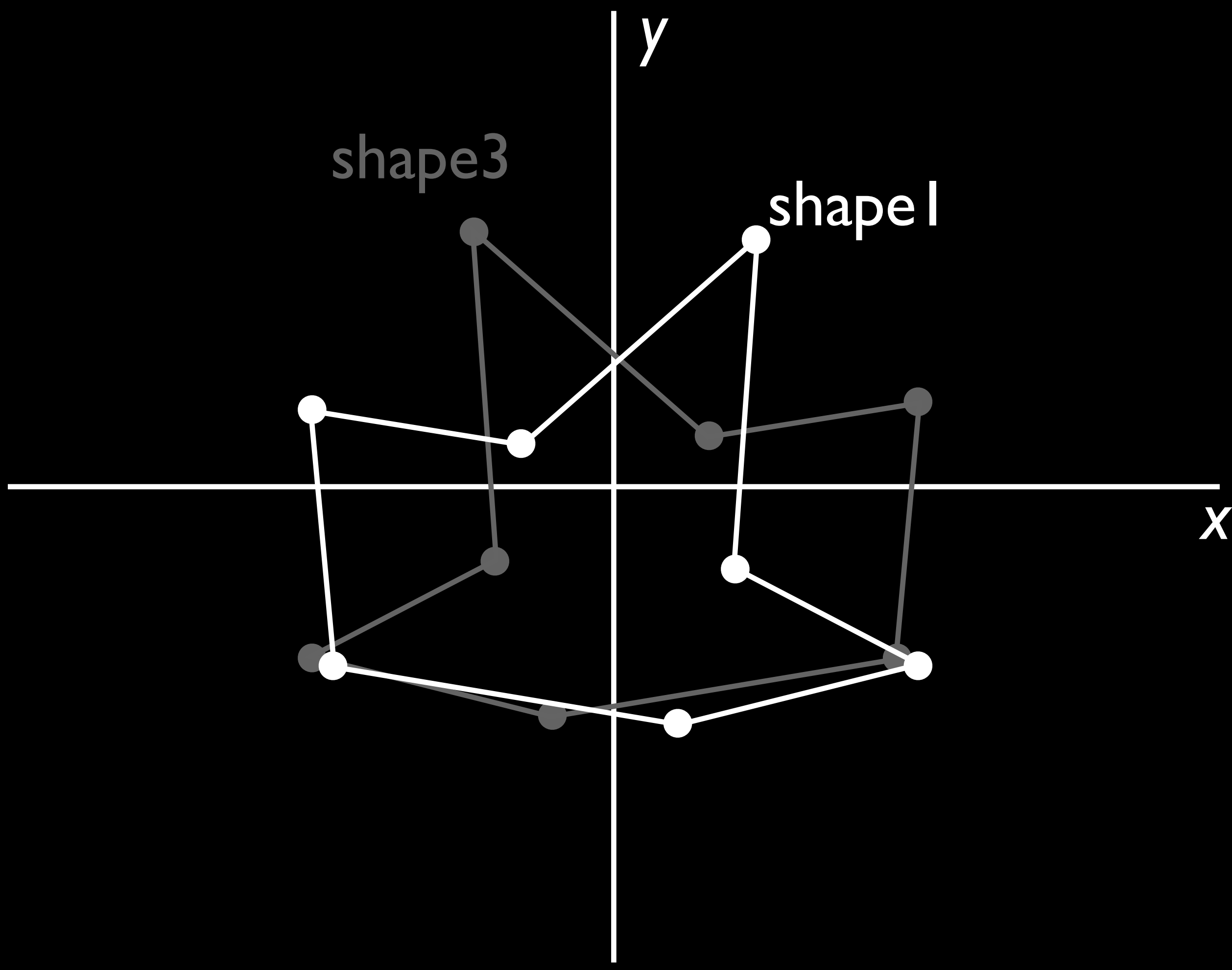
Points are then attributes. e.g.

```
shape1.x_pt # a list?
shape1.y_pt # a set?
```



Everything in Python is an object. This means we can use anything as an attribute, even other objects. Normally stick to the built in types.

Functions that operate on an object's data become methods.

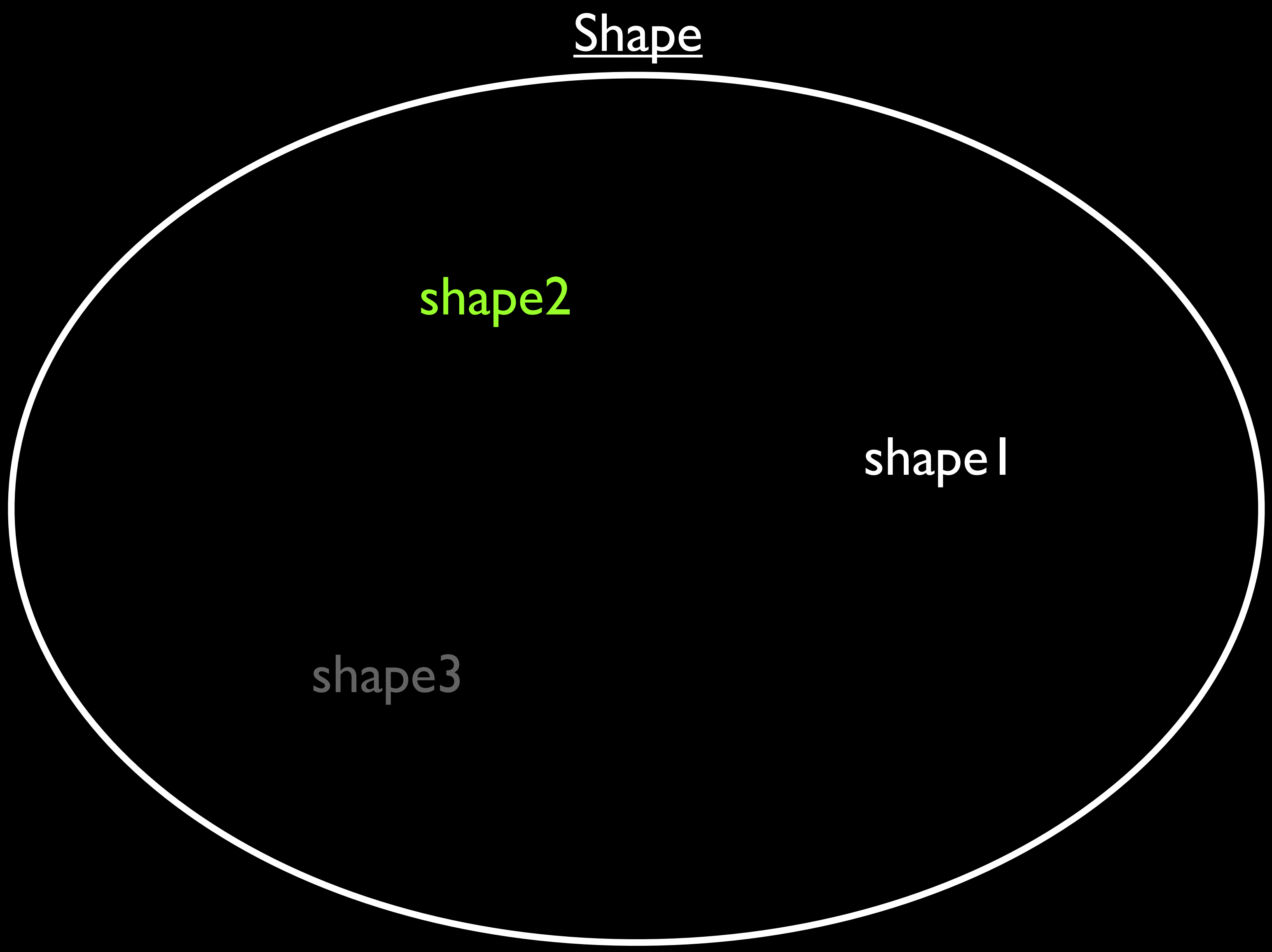


Methods are just functions connected to objects. They need brackets and can have arguments.

```
shape3 = shape1.reflect_y()
```

or

```
shape3 = shape1.reflect('yaxis')
```



Shape

shape2

shape1

shape3

Each object belongs to the class of shapes. They are instances of the class and have the same attributes and methods. They represent similar things and you can do the same sort of thing to them.

How to make a class

```
class Foo:

    def __init__(self, arg):
        self.attribute = arg*5
        self.count = 0

    def method(self, arg):
        self.count = self.count+1
        return self.attribute*self.count
```

Class definition

```
class Foo:

    def __init__(self, arg):
        self.attribute = arg*5
        self.count = 0

    def method(self, arg):
        self.count = self.count+arg
        return self.attribute*self.count
```

Class definition

```
instance = Foo(2)
print instance.method(2)
# 20
print instance.method(2)
# 40
```

Class use

Using the class “like a function” calls the __init__ method.

```
class Foo:

    def __init__(self, arg):
        self.attribute = arg*5
        self.count = 0

    def method(self, arg):
        self.count = self.count+arg
        return self.attribute*self.count
```

Class definition

```
instance = Foo(2)
print instance.method(2)
# 20
print instance.method(2)
# 40
```

Class use

Self (the first argument to any function in a class definition) represents this instance of the class.

```
class Foo:

    def __init__(self, arg):
        self.attribute = arg*5
        self.count = 0

    def method(self, arg):
        self.count = self.count+arg
        return self.attribute*self.count
```

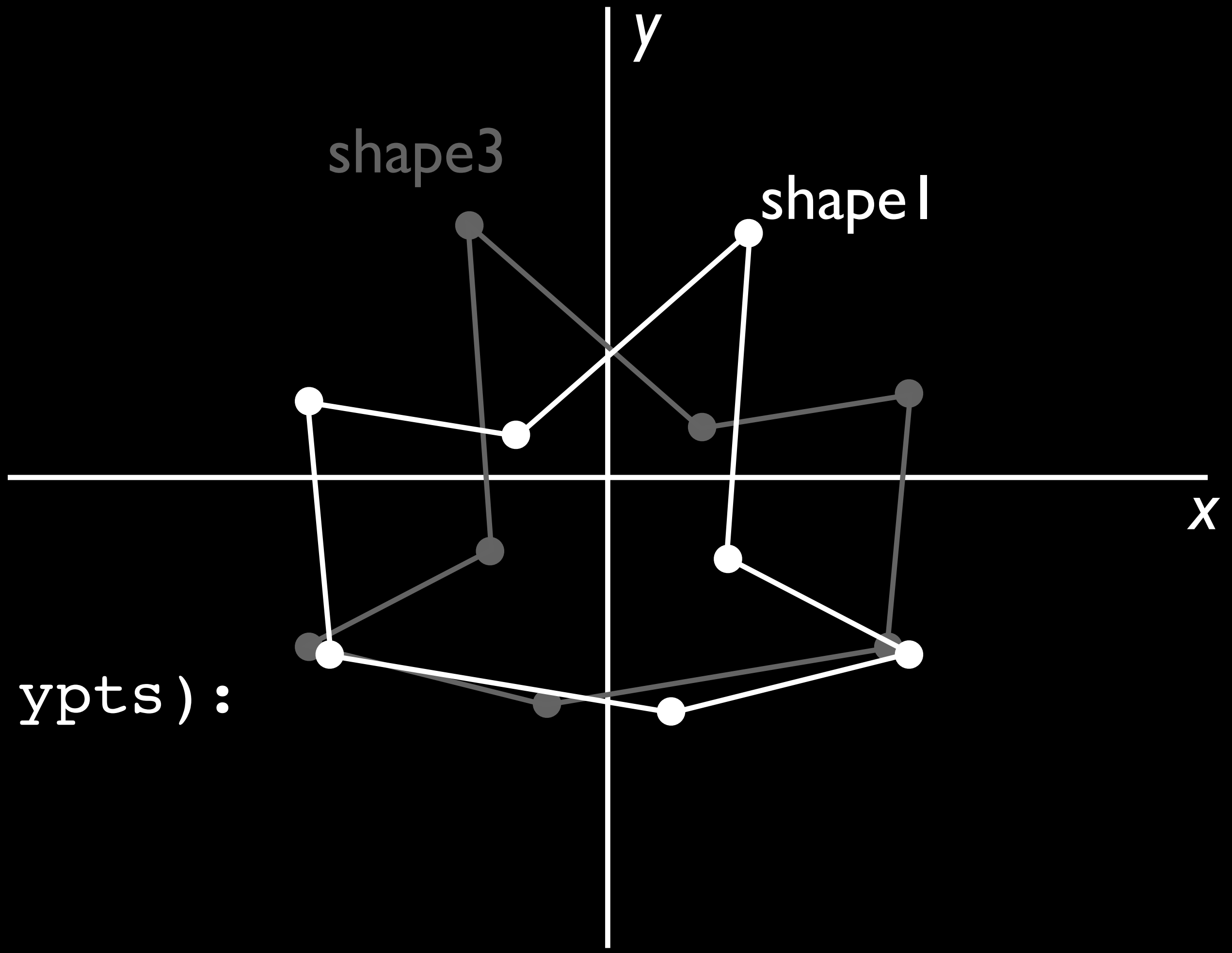
Class definition

```
instance = Foo(2)
print instance.method(2)
# 20
print instance.method(2)
# 40
```

Class use

Method and attribute names are used directly

You now know how to define the Shape class



```
class Shape:

    def __init__(self, xpts, ypts):
        self.xs = xpts
        self.ys = ypts

    def reflect_y(self):
       for x, y in zip(self.xs, self.ys):
            ...
       return Shape(new_xs, new_ys)
```

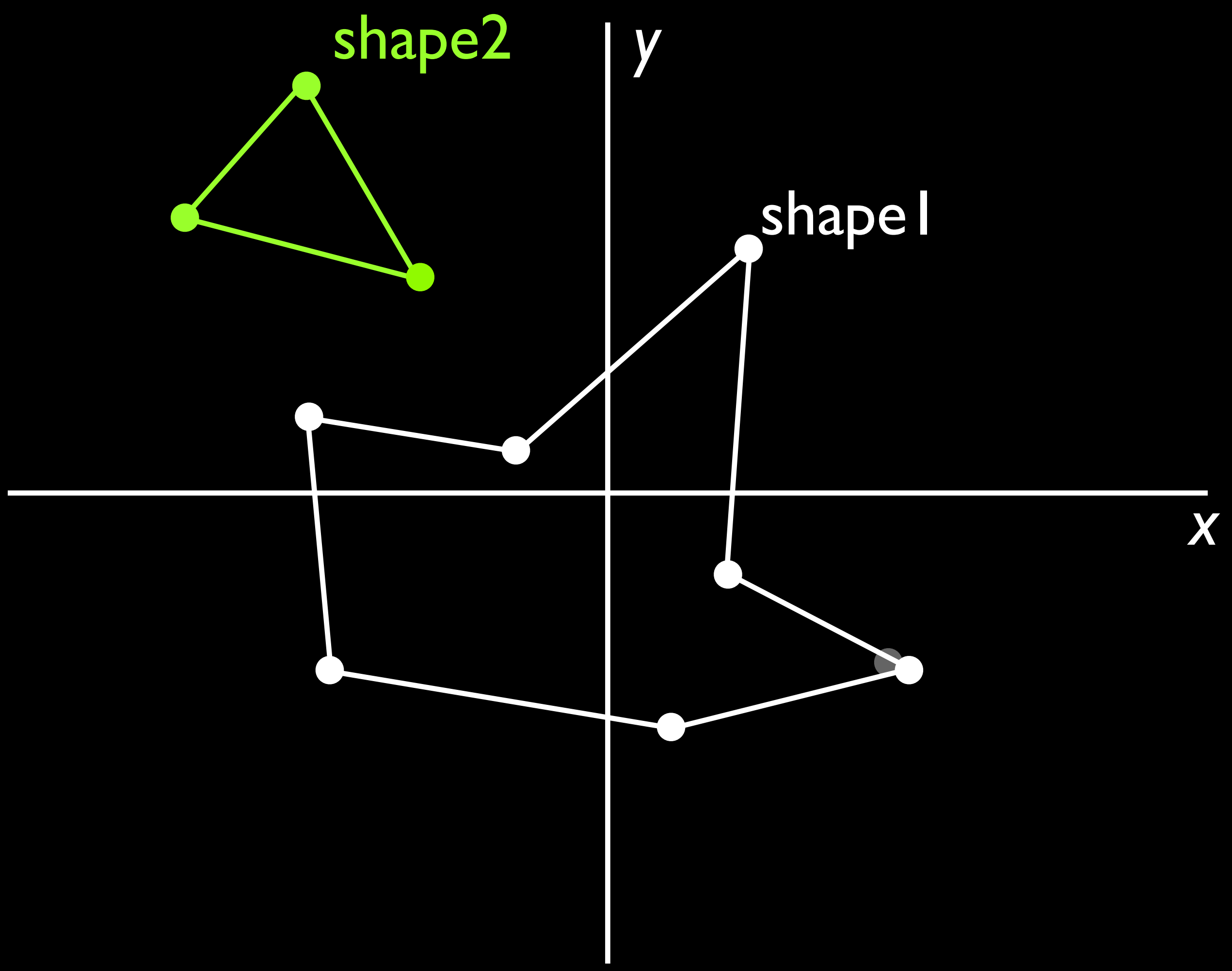
```
shape1 = Shape(....)
shape3 = shape1.reflect_y()
```

Objects are just a way of organising data...

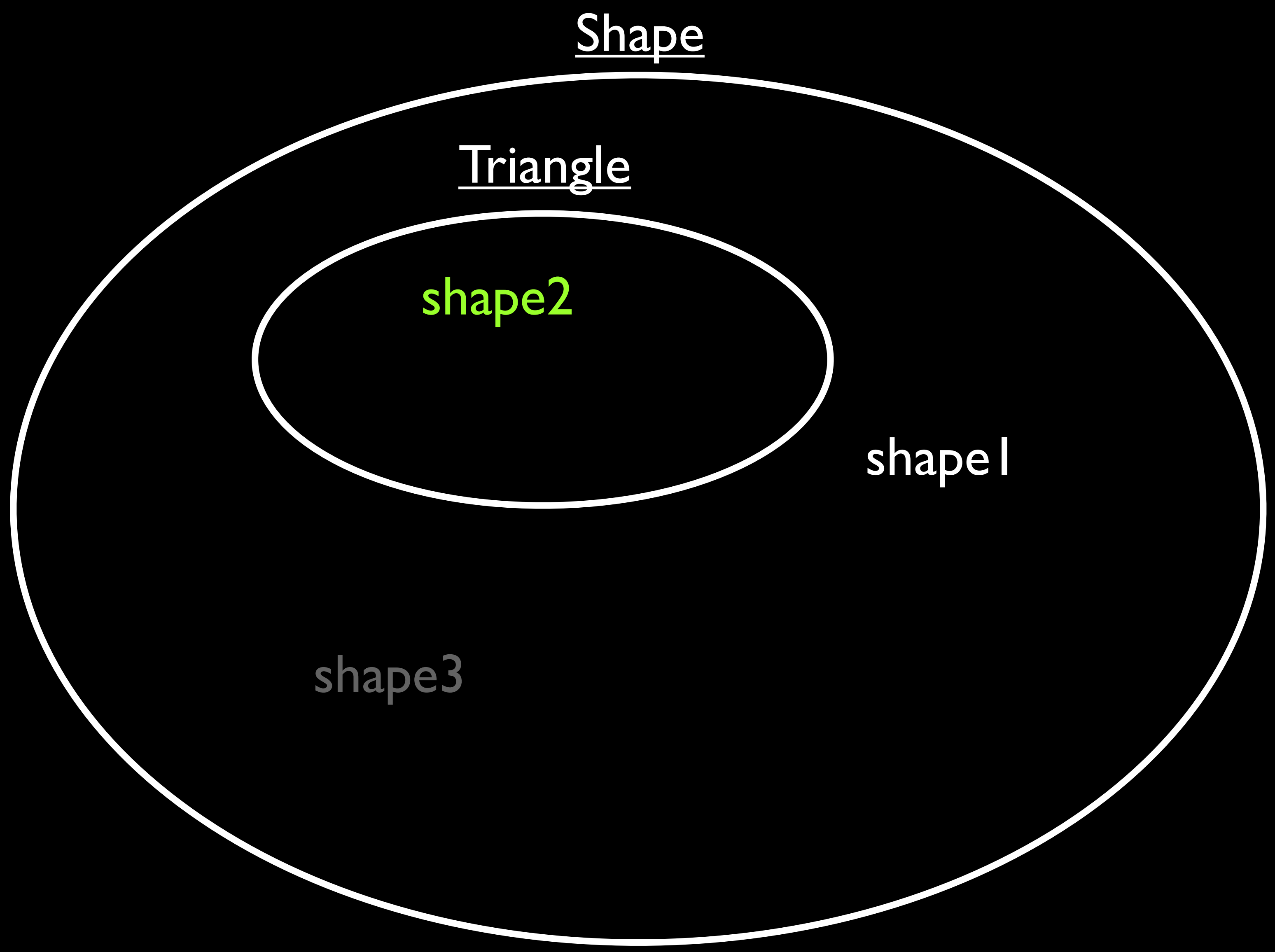
... which should make code reuse easer and enhance maintainability...

... and you know how they work

```
class Shape:
   ...
   def area(self):
      # Hard problem for
      # the general case


class Triangle:
   ...
   def area(self):
      # Not too difficult
      # just an equation
```

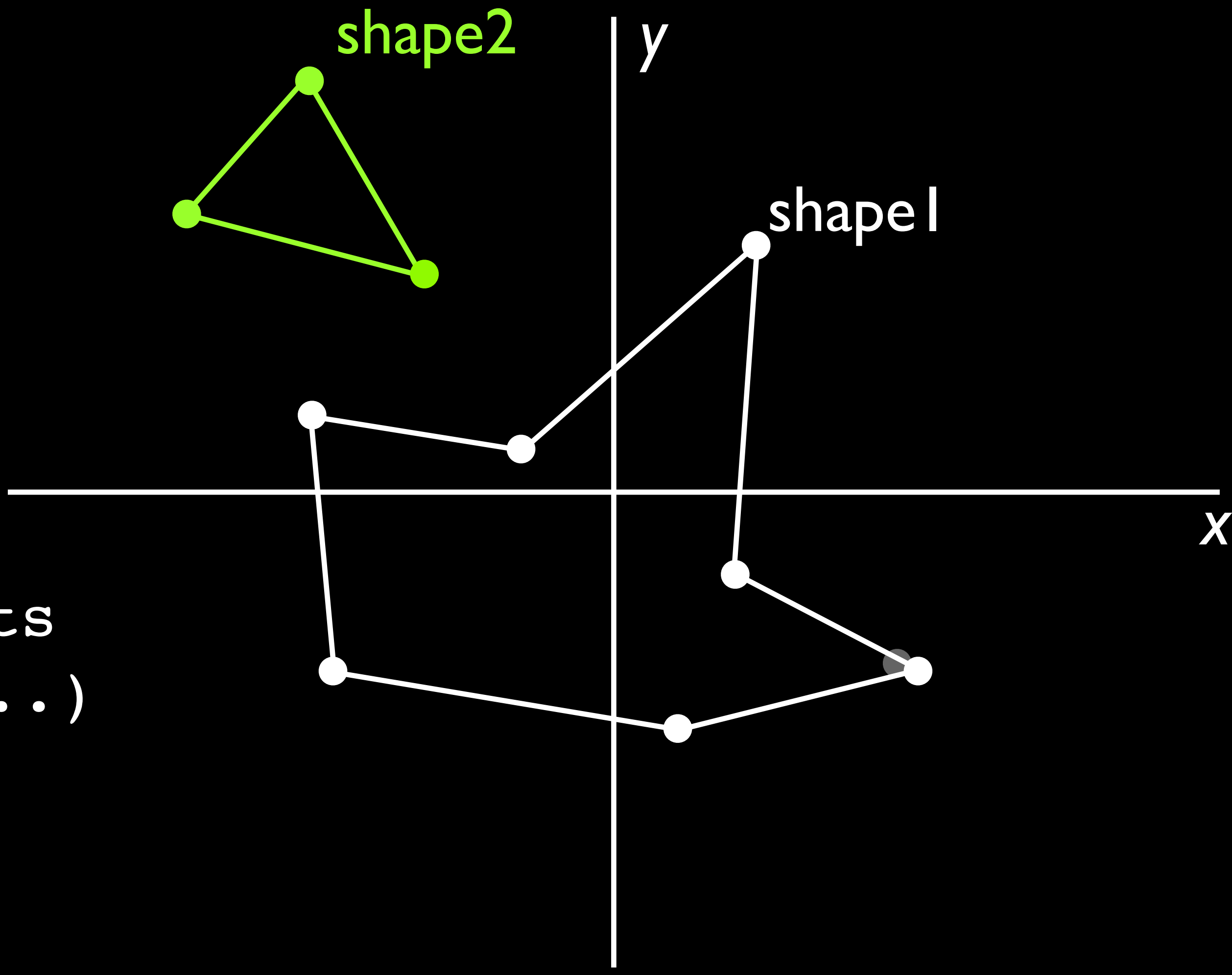


Think about calculating the area of our shapes. This is much easer for shape2 than shape1. Shape2 is a special kind of shape



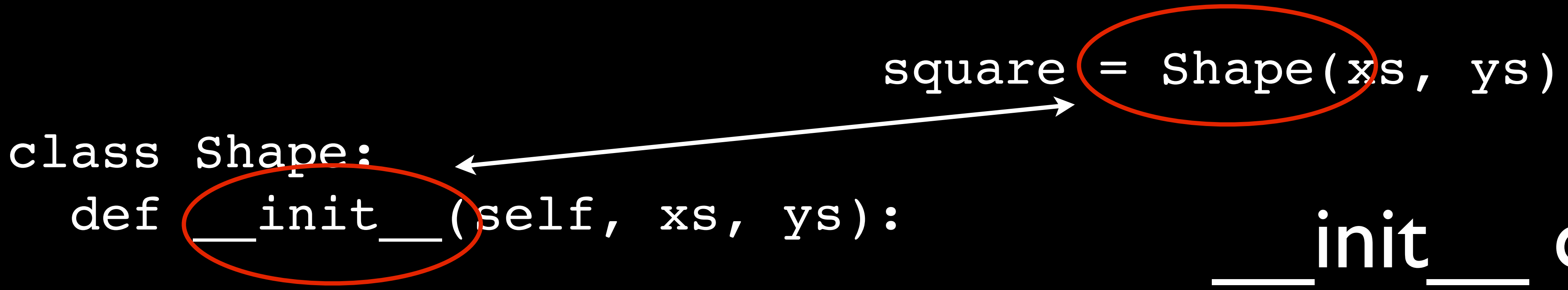
A Triangle **is a** Shape; shape2 **is a** Triangle; shape2 **is a** shape with “special” methods. We don’t want to have to rewrite all the shared code.

```
class Shape:
  def __init__(self, ...):


class Triangle(Shape):
  def __init__(self, ...):
      # Check we have 3 points
      Shape.__init__(self, ...)

  def area(self):
     # Not too difficult
     # just an equation
```



Triangle **inherits** from Shape. When an instance of Triangle calls a method the function defined in Triangle is used, if this does not exist, the one defined in Shape is used (and so on).

Special methods



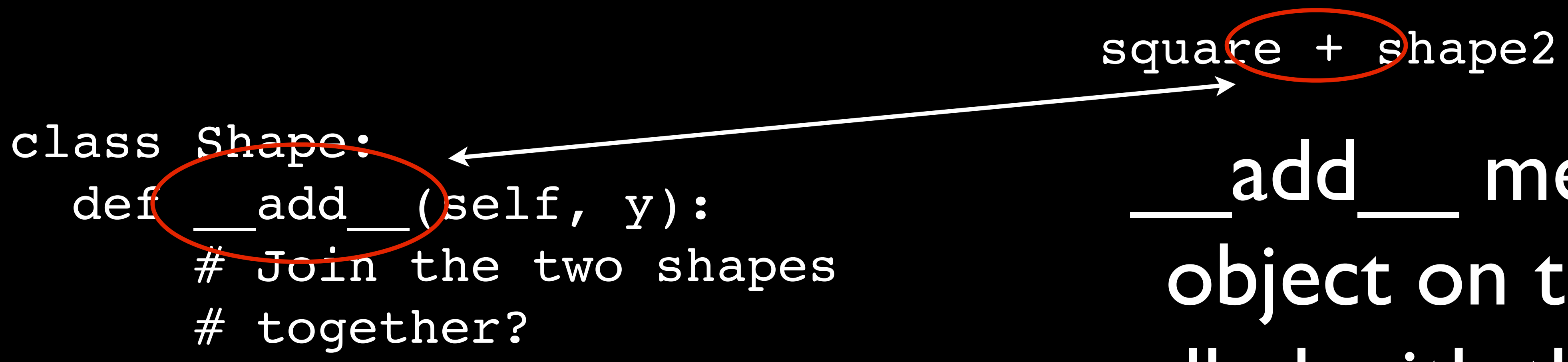
```
square = Shape(xs, ys)
```

```
class Shape:
  def __init__(self, xs, ys):
```

__init__ called on object creation

Everything is an object. We need a way to make our objects interact with the language.

Special methods



```
square + shape2
```

```
class Shape:
  def __add__(self, y):
      # Join the two shapes
      # together?
```

__add__ method of object on the LHS called with the object on the RHS as an argument

Everything is an object. We need a way to make our objects interact with the language.

Special methods

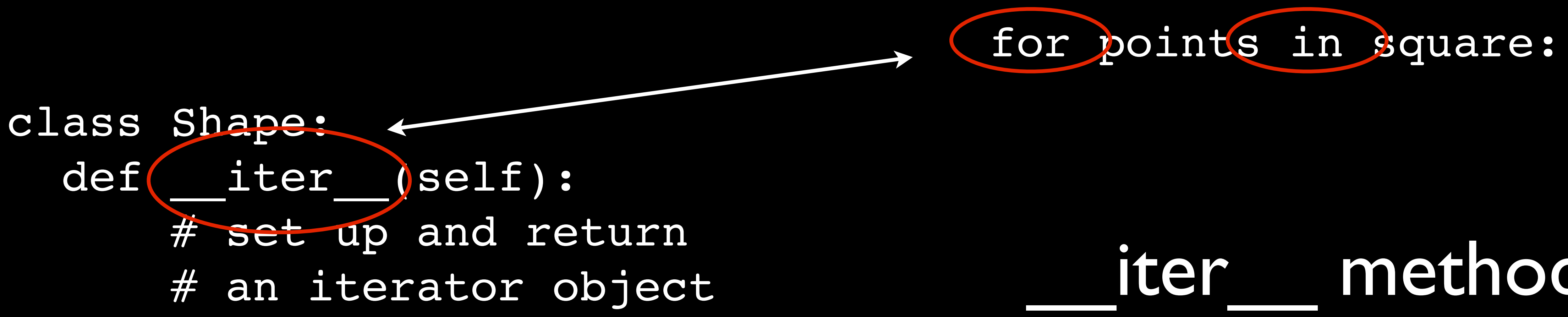
```
len(square)
```

```
class Shape:
  def __len__(self):
      # return the number of
      # points
```

__len__ method called when built in len() function used.

Everything is an object. We need a way to make our objects interact with the language.

Special methods

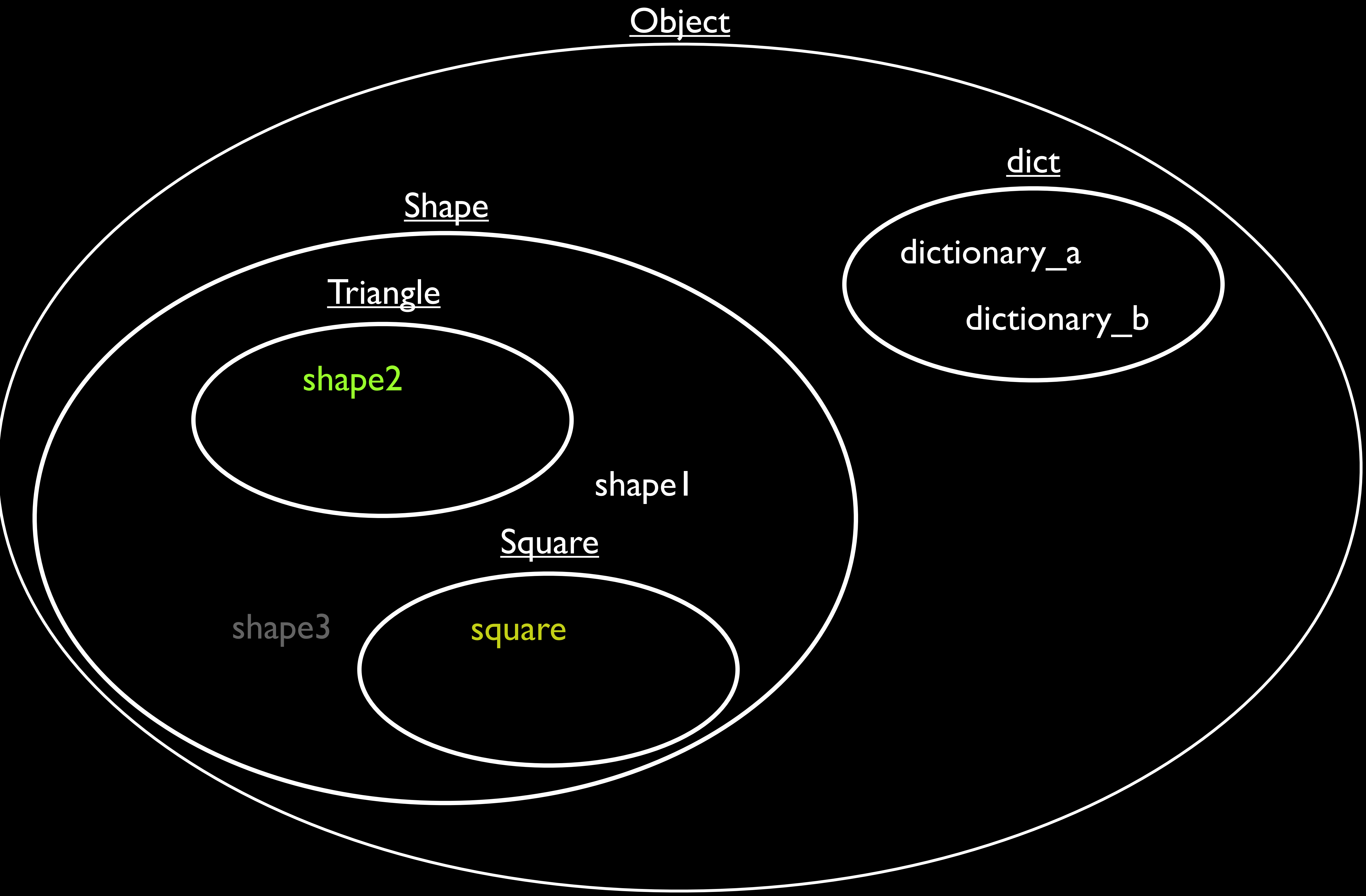


```
for points in square:
```

```
class Shape:
   def __iter__(self):
       # set up and return
       # an iterator object
```

__iter__ method
called when an object
is used with for.

Everything is an object. We need
a way to make our objects
interact with the language.



Everything is an object

Object orientated programming:

Encapsulation

Dynamic dispatch

Inheritance

When do you care... with Python

- Small Python programs - just sits at the back of your mind. You understand file.close(). Understand how stuff in the library works.
- Bigger programs - you may define one or two critical classes.
- Occasionally you need to make your classes interact with the wider program (__iter__ etc.). E.g. if you need a quaternion class.
- Python documentation authors assume you know about OO.