14.3 OBJECT-ORIENTED PROGRAMMING

The most common style of programming today is *object-oriented programming*. We're going to define in contrast with procedural programming that we've been doing up until now.

Back in the 1960's and 1970's, procedural programming was the dominant form of programming. People used *procedural abstraction* and defined lots of functions at high and low levels, and reused their functions wherever possible. This worked reasonably well—up to a point. As programs got really large and complex with many programmers working on them at the same time, procedural programming started to break down.

But programmers still ran into problems. People would write programs that would modify data in ways that other people wouldn't expect. They would use the same names for functions and find that their code couldn't be integrated into one large program.

There were also problems in *thinking* about programs and the tasks that the programs were supposed to perform. Procedures are about *verbs*—tell the computer to do this, tell the computer to do that to the data. It's not clear that that's the way that people think best about their problems.

Object-oriented programming is *noun-oriented programming*. Someone building an object-oriented program starts by thinking about what the nouns are in the *domain* of the problem—what are the people and things that are part of this problem and its solution? The process of identifying the objects, what each of them knows about (with respect to the problem), and what each of them has to do is called *object-oriented analysis*.

Programming in an object-oriented way means that you define variables (called *instance variables*) *for the objects* and you define functions (called *methods*) *for the objects*. You have very few or even *no* global functions or variables—things that are accessible everywhere. Instead, objects talk to one another by asking each other to do things via their methods. Adele Goldberg, one of the pioneers of object-oriented programming, calls this "Ask, don't touch." You can't just "touch" data and do whatever you want with it—instead, you "ask" objects to manipulate their data through their methods.

The term "object-oriented programming" was invented by Alan Kay. Kay is a brilliant, multidisciplinary character—he holds undergraduate degrees in Mathematics and Biology, a Ph.D. in computer science, and has been a professional jazz guitarist. He was awarded in 2004 the *ACM Turing Award*, the *Nobel Prize* of Computing. He saw object-oriented programming as a way of developing software that could truly scale to large systems. He described objects as being like biological *cells* that work together in well-defined ways to make the whole organism work. Like cells, objects would–

- Help manage *complexity* by distributing responsibility for tasks across many objects, rather than one big program.
- Support *robustness* by making the objects work relatively independently.
- Support *reuse* because each object would provide *services* to other ojects (tasks that the object would do for other objects, accessible through its

methods)—just as real world objects do.

The notion of starting from nouns is part of Kay's vision. Software, he said, is actually a *simulation* of the world. By making software *model* the world, it becomes clearer how to make software. You look at the world and how it works, then copy that into software. Things in the world *know* things–those become *instance variables*. Things in the world can *do* things–those become *methods*.

14.3.1 An Object-Oriented Slide Show

Let's use object-oriented techniques to build a slide show program. Let's say that we want to show a picture, then play a corresponding sound—and wait until the sound is done, before going on to the next picture. We'll use the function (mentioned many chapters ago) *blockingPlay()*.

Recipe 122: Slide show as one big function

```
def playslideshow():
    pic = makePicture(getMediaPath("barbara.jpg"))
    snd = makeSound(getMediaPath("bassoon-c4.wav"))
    show(pic)
    blockingPlay(snd)
    pic = makePicture(getMediaPath("beach.jpg"))
    snd = makeSound(getMediaPath("bassoon-e4.wav"))
    show(pic)
    blockingPlay(snd)
    pic = makePicture(getMediaPath("santa.jpg"))
    snd = makeSound(getMediaPath("bassoon-g4.wav"))
    show(pic)
    blockingPlay(snd)
    pic = makePicture(getMediaPath("jungle2.jpg"))
    snd = makeSound(getMediaPath("bassoon-c4.wav"))
    show(pic)
    blockingPlay(snd)
```

This isn't a very good program for any perspective. From a procedural programming perspective, there's an awful lot of duplicated code here. It would be nice to get rid of it. From an object-oriented programming perspective, we should have an *object*: A slide.

As we mentioned, objects have two parts to them. Objects *know* things—those become *instance variables*. Objects can *do* things—those become *methods*. We're going to access both of these using dot notation.

So, what does a slide know? It knows its *picture* and its *sound*. What can a slide do? It can *show* itself, by showing its picture and (block) playing its sound.

To define a slide object in Python (and many other object-oriented programming languages, including Java and C++), we must define a slide *class*. A class defines the instance variables and methods for a set of objects, that is, what each object of that class knows and can do. Each object of that class is called an *instance* of the class. We'll make multiple slides by making multiple instances of the slide class—just as our bodies might make multiple kidney cells or multiple heart cells, each of which knows how to do certain kinds of tasks.

To create a class in Python, we start with:

```
class slide:
```

What comes after that, indented, are the methods for creating new slides and playing slides. Let's add a *show()* method to our slide class.

```
class slide:
  def show(self):
      show(self.picture)
      blockingPlay(self.sound)
```

To create new instances, we call the class name like a function. We can define new instance variables by simply assigning them. So here's creating a slide and giving it a picture and sound.

```
>>> slide1=slide()
>>> slide1.picture = makePicture(getMediaPath("barbara.jpg"))
>>> slide1.sound = makeSound(getMediaPath("bassoon-c4.wav"))
```

We can now show our slide by saying `slide1.show()`.

What's this `self` stuff? When we execute `object.method()`, Python finds the method in the object's class, then calls it, using the instance object as an *input*. It's Python style to name that input variable `self` (because it is the object itself). Since we have the object in the variable `self`, we can then access its picture and sound by saying `self.picture` and `self.sound`.

But this is still pretty hard to use, if we have to set up all the variables from the command area. How could we make it easier? What if we could pass in the sound and picture for the slides as *inputs* to the `slide` class, as if the class were a real function? We can do this, by defining something called a *constructor*.

To create new instances with some inputs, we must define a function named `__init__`. That's "underscore-underscore-i-n-i-t-underscore-underscore." It's the predefined name in Python for a method that *initializes* new objects. Our `__init__` method needs three inputs: the instance itself (because all methods get that) and a picture and sound.

Recipe 123: A Slide Class

```
class slide:
  def __init__(self, pictureFile,soundFile):
      self.picture = makePicture(pictureFile)
      self.sound = makeSound(soundFile)
  def show(self):
      show(self.picture)
      blockingPlay(self.sound)
```

We can use our slide class to define a slide show like this.

Recipe 124: A slide show, using our Slide class

```
def playslideshow():
    slide1 = slide(getMediaPath("barbara.jpg"), getMediaPath("bassoon-c4.wav"))
    slide2 = slide(getMediaPath("beach.jpg"),getMediaPath("bassoon-e4.wav"))
    slide3 = slide(getMediaPath("santa.jpg"),getMediaPath("bassoon-g4.wav"))
    slide4 = slide(getMediaPath("jungle2.jpg"),getMediaPath("bassoon-c4.wav"))
    slide1.show()
    slide2.show()
    slide3.show()
    slide4.show()
```

One of the features of Python that make it so powerful is that we can mix object-oriented and functional programming styles. Slides are now objects that can easily be stored in lists, like any other kind of Python object. Here's an example of the same slide show where we use `map` to show the slide show.

Recipe 125: Slide show, in objects and functions

```
def showSlide(aslide):
  aslide.show()

def playslideshow():
    slide1 = slide(getMediaPath("barbara.jpg"), getMediaPath("bassoon-c4.wav"))
    slide2 = slide(getMediaPath("beach.jpg"),getMediaPath("bassoon-e4.wav"))
    slide3 = slide(getMediaPath("santa.jpg"),getMediaPath("bassoon-g4.wav"))
    slide4 = slide(getMediaPath("jungle2.jpg"),getMediaPath("bassoon-c4.wav"))
    map(showSlide,[slide1,slide2,slide3,slide4])
```

Is the object-oriented version of the slide show easier to write? It certainly has less replication of code. It features *encapsulation* in that the data and behavior of the object are defined in one and only one place, so any change to one is easily changed in the other. Being able to use lots of objects (like lists of objects) is called *aggregation* is a very powerful idea. We don't always have to define new classes—we can often use the powerful structures we know like lists with existing objects to great impact.

14.3.2 Joe the Box

The earliest example used to teach object-oriented programming wsa developed by Adele Goldberg and Alan Kay–it's called *Joe the Box*. Imagine that you have a class `Box` like the below:

```
class Box:
 def __init__(self):
   self.setDefaultColor()
   self.size=10
   self.position=(10,10)
 def setDefaultColor(self):
   self.color = red
 def draw(self,canvas):
   addRectFilled(canvas, self.position[0], self.position[1], self.size,
self.size, self.color)
```

What will you see if you execute the below?

```
>>> canvas = makeEmptyPicture(400,200)
>>> joe = Box()
>>> joe.draw(canvas)
>>> repaint(canvas)
```

Let's trace it out.

- Obviously, the first line just creates a black `canvas` that is 400 pixels wide and 200 pixels high.
- When we create `joe`, the `__init__` method is called. The method `setDefaultColor` is called on `joe`, so he gets a default color of red. When `self.color=red` is executed, the *instance variable* `color` is created for `joe` and gets a value of red. We return to `__init__` where joe is given a size of 10 and a position of (10,10) (`size` and `position` both become new instance variables).
- When `joe` is asked to `draw` himself on the `canvas`, he's drawn as a red, filled rectangle (`addRectFilled`), at x position 10 and y position 10, with a size of 10 pixels on each side.

We could add a method to `Box` that allows us to make `joe` change his size.

```
class Box:
 def __init__(self):
   self.setDefaultColor()
   self.size=10
   self.position=(10,10)
 def setDefaultColor(self):
   self.color = red
 def draw(self,canvas):
    addRectFilled(canvas, self.position[0], self.position[1], self.size,
self.size, self.color)
  def grow(self,size):
    self.size=self.size+size
```

Now we can tell `joe` to `grow`. A negative number, like -2, will cause `joe` to shrink. A positive number will cause `joe` to grow–though we'd have to add a `move` method if we wanted him to grow much and still fit on the canvas.

Now consider the below code added to the same Program Area.

```
class SadBox(Box):
  def setDefaultColor(self):
    self.color=blue
```

Notice that `SadBox` lists `Box` as a *superclass*. That means that `SadBox` *inherits* all the methods of Box. What will you see if you execute the below?

```
>>> jane = SadBox()
>>> jane.draw(canvas)
>>> repaint(canvas)
```

Let's trace it out:

- When `jane` is created as a `SadBox`, the method `__init__` is executed in class `Box`.
- The first thing that happens in `__init__` is that we call `setDefaultColor` *on the input object* `self`. That object is now `jane`. So, we call `jane`'s `setDefaultColor`. We say that `SadBox`'s `setDefaultColor` *overrides* `Box`'s.
- The `setDefaultColor` for `jane` sets the color to blue.
- We then return to executing the rest of `Box`'s `__init__`. We set `jane`'s size to 10 and position to (10,10).
- When we tell `jane` to draw, she appears as a 10x10 blue square at position (10,10). If we hadn't moved or grown `joe`, he'd disappear as `jane` is drawn on top of him.

Notice that both `joe` and `jane` are *kinds* of Boxes. They have the same instance variables (but different *values* for the same variables) and mostly know the same things. Because both understand `draw`, for example, we say that `draw` is *polymorphic*. Different objects of different classes do the same kind of thing when told to execute that method. A SadBox (`jane`) is slightly different in how it behaves when it gets created, so it knows some things differently. Joe and Jane highlight some of the basic ideas of object-oriented programming: Inheritance, specialization in sub-classes, and shared instance variables while having different instance variable values.

14.3.3 Object-oriented media

Of course, we've been using objects already. Pictures, sounds, samples, and colors are all objects. Our lists of pixels and samples are certainly examples of aggregation. The functions we've been using are actually just covering up the underlying methods. We certainly can just call the objects' methods directly.

```
>>> pic=makePicture(getMediaPath("barbara.jpg"))
>>> pic.show()
```

Here's how the *function* `show()` is defined. You can ignore `raise` and `__class__`. The key point is that the function is simply executing the existing method.

```
def show(picture):
    if not picture.__class__ == Picture:
        print "show(picture): Input is not a picture"
        raise ValueError
    picture.show()
```

Did you notice that we defined the method `show()` for slides, the same name that we have for showing pictures? First of all, we can clearly do that—objects can have their own methods with names that other objects use, too. Much more powerful is that each of those methods with the same name can achieve the same *goal*, but in different ways. For both slides and pictures, the method `show()` says, "Show the object." But what's really happening is different in each case: Pictures just show themselves, but slides show their pictures and play their sounds.

Computer Science Idea: Polymorphism
When the same name can be used to invoke different methods that achieve the same goal, we call that *polymorphism*. It's very powerful for the programmer. You simply tell an object `show()`—you don't have to care exactly what method is being executed, and you don't even have to know exactly what object it is that you're telling to show! You the programmer simply specify your *goal*, to show the object. The object-oriented program handles the rest.

There are several examples of polymorphism built into the methods that we're using in JES[1]. For example, both pixels and colors understand the methods `setRed`, `getRed`, `setBlue`, `getBlue`, `setGreen`, and `getGreen`. This allows us to manipulate the color of the pixels without pulling out the color object separately, or to manipulate colors. We could have defined the functions to take both kinds of inputs, or provide different functions for each kind of input, but both of those options get confusing. It's easy to do with methods.

[1]Recall that JES is an environment for programming in Jython, which is a particular kind of Python. The media supports are part of what JES provides—they're not part of the core of Python

```
>>> pic=makePicture(getMediaPath("barbara.jpg"))
>>> pic.show()
>>> pixel = getPixel(pic,100,200)
>>> print pixel.getRed()
73
>>> color = pixel.getColor()
>>> print color.getRed()
73
```

Another example is the method `writeTo()`. The method `writeTo(filename)` is defined for both pictures and sounds. Did you ever confuse `writePictureTo()` and `writeSoundTo()`? Isn't it easier to just always write `writeTo(filename)`? That's why that method is named the same in both classes, and why polymorphism is so powerful. (You may be wondering why we didn't introduce that in the first place. Were you ready in Chapter 2 to talk about dot notation and polymorphic methods? Didn't think so.)

Overall, there are actually many more methods defined in JES than functions. More specifically, there are a bunch of methods for drawing onto pictures that aren't available as functions.

- As you would expect, pictures understand `pic.addRect(color,x,y,width,height)`, `pic.addRectFilled(color,x,y,width,height)`, `pic.addOval(color,x,y,width,height)`, and `pic.addOvalFilled(color,x,y,width,height)`.

  See Figure 14.3 for examples of rectangles methods, drawn the below example.

  ```
  >>> pic=makePicture (getMediaPath("640x480.jpg"))
  >>> pic.addRectFilled (orange,10,10,100,100)
  >>> pic.addRect (blue,200,200,50,50)
  >>> pic.show()
  >>> pic.writeTo("newrects.jpg")
  ```

  Figure 14.4 for examples of ovals, drawn from the below example.

  ```
  >>> pic=makePicture (getMediaPath("640x480.jpg"))
  >>> pic.addOval (green,200,200,50,50)
  >>> pic.addOvalFilled (magenta,10,10,100,100)
  >>> pic.show()
  >>> pic.writeTo("ovals.jpg")
  ```

- Pictures also understand *arcs*. Arcs are literally parts of a circle. The two methods are `pic.addArc(color,x,y,width,height,startangle,arcangle)` and `pic.addArcFilled(color,x,y,width,height,startangle,arcangle)`. They draw arcs for `arcangle` degrees, where `startangle` is the starting point. 0 degrees is at 3 o'clock on the clock face. A positive arc is counter-clockwise, and negative is clockwise. The center of the circle is the middle of the rectangle defined by (x, y) with the given `width` and `height`.

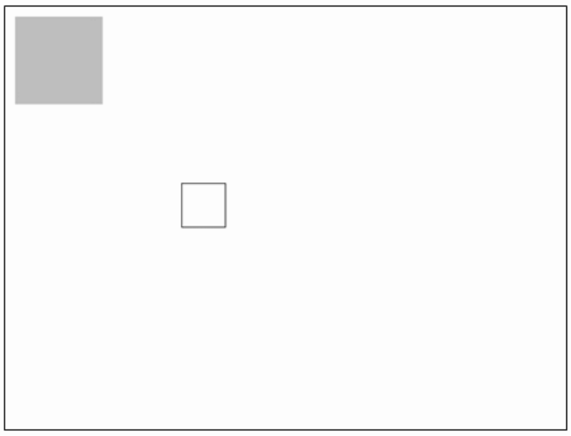

FIGURE 14.3: Examples of rectangle methods

- We can also now draw colored lines, using `pic.addLine(color,x1,y1,x2,y2)`.

  Figure 14.5 for examples of arcs and lines, drawn from the below example.

```
>>> pic=makePicture (getMediaPath("640x480.jpg"))
>>> pic.addArc(red,10,10,100,100,5,45)
>>> pic.show()
>>> pic.addArcFilled (green,200,100,200,100,1,90)
>>> pic.repaint()
>>> pic.addLine(blue,400,400,600,400)
>>> pic.repaint()
>>> pic.writeTo("arcs-lines.jpg")
```

- Text in Java can have styles, but only limited to make sure that all platforms can replicate those styles. `pic.addText(color,x,y,string)` is the one we would expect to see. There is also `pic.addTextWithStyle(color,x,y,string,style)` that takes a style created from `makeStyle(font,emphasis,size)`. The `font` is `sansSerif`,`serif`, or `mono`. The `emphasis` is `italic`, `bold`, or `plain`, or sum them to get combinations, e.g., `italic+bold`. `size` is a point size.

  Figure 14.6 for examples of text, drawn from the below example.

```
>>> pic=makePicture (getMediaPath("640x480.jpg"))
>>> pic.addText(red,10,100,"This is a red  string!")
>>> pic.addTextWithStyle (green,10,200,"This is a bold, italic,
green, large string", makeStyle(sansSerif,bold+italic,18))
>>> pic.addTextWithStyle (blue,10,300,"This is a blue, larger,
italic-only, serif string", makeStyle(serif,italic,24))
>>> pic.writeTo("text.jpg")
```

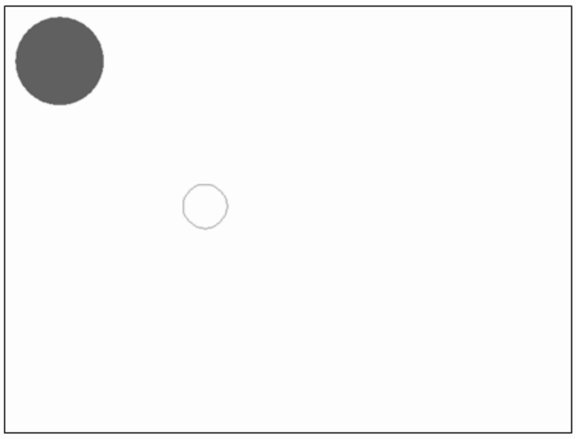

FIGURE 14.4: Examples of oval methods

Our older media functions that we wrote can be re-written in method form.

Recipe 126: Making a sunset using methods

```
def makeSunset(picture):
  for p in getPixels(picture):
    p.setBlue(p.getBlue()*0.7)
    p.setGreen(p.getGreen()*0.7)
```

The methods for accessing samples are `getSampleValue` and `getSampleObjectAt`.

Recipe 127: Turn a sound backwards using methods

```
def backwards(filename):
  source = makeSound(filename)
  target = makeSound(filename)

  sourceIndex = source.getLength()
  for targetIndex in range(1,target.getLength()+1):
    # The method is getSampleValue, not getSampleValueAt
    sourceValue =source.getSampleValue(sourceIndex)
    # The method is setSampleValue, not setSampleValueAt
    target.setSampleValue(targetIndex,sourceValue)
    sourceIndex = sourceIndex - 1

  return target
```

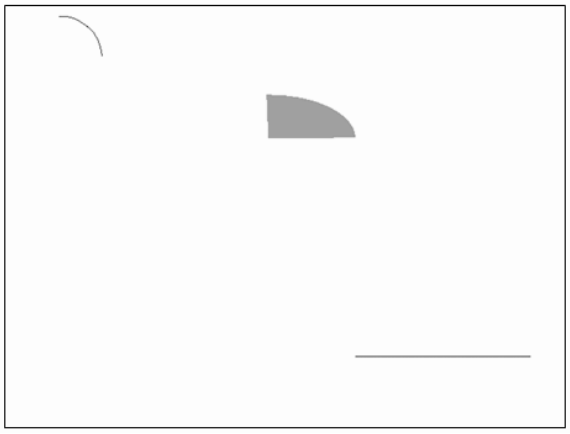
FIGURE 14.5: Examples of arc methods

14.3.4 Why objects?

A significant role for objects is to reduce the number of names that you have to remember. Through polymorphism, you only have to remember the name and the goal, not all the various global functions.

More importantly, though, objects encapsulate data and behavior. Imagine that you wanted to change the name of an instance variable, and then all the methods that use the variable. Changing them all in one place, together, is useful.

Aggregation is also a significant benefit of object systems. You can have lots of objects doing useful things. Want more? Just create them!

Python's objects are similar to many languages objects. One significant difference is in access to instance variables, though. In Python, any object can access and manipulate any other objects' instance variables. That's not true in languages like Java, C++, or Smalltalk. In these other languages, access to instance variables from other objects is limited and can even be eliminated entirely—then, you can only access objects' instance variables through methods called *getters* and *setters* (to get and set the instance variable).

Another big part of object systems that we haven't addressed is *inheritance*. We can declare one class (*parent class*) to be *inherited* by another class (*child class*). Inheritance provides for instant polymorphism—the instances of the child automatically have all the data and behavior of the parent class. The child can then add more behavior and data to what the parent class had. This is called making the child a *specialization* of the parent class. For example, a 3-D rectangle instance might know and do everything that a rectangle instance does by saying `class rectangle3D(rectangle)`.

Inheritance gets a lot of press in the object-oriented world, but it's a tradeoff. It does reduce even further duplication of code. But in actual practice, inheritance

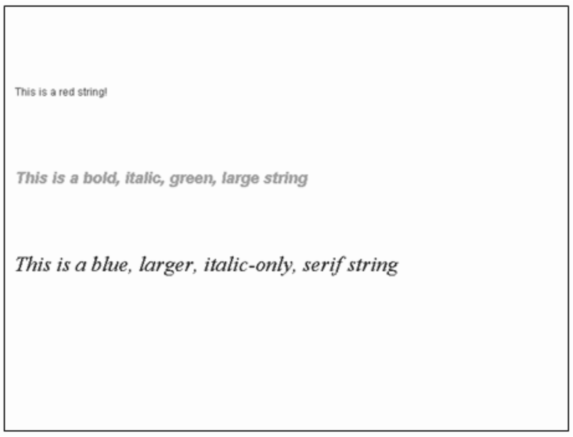
This is a red string!
This is a bold, italic, green, large string
This is a blue, larger, italic-only, serif string

FIGURE 14.6: Examples of text methods

isn't used all *that* much, and it can be confusing. Whose method is being executed when I type this?

So, when should use objects? You should define your own object classes when you have data in groups (like both pictures and sounds) and behavior that you want to define for all instances of that group. You should use existing objects *all the time*. They're very powerful. If you're not comfortable with dot notation and the ideas of objects, you can stick with functions—they work just fine. Objects just give you a leg up on more complex systems.

PROGRAMMING SUMMARY

Some of the functions and programming pieces that we met in this chapter.

Functional Programming