Object-Oriented Jython

This chapter is going to cover the basics of object-oriented programming. We'll start with covering the basic reasons why you would want to write object-oriented code in the first place, and then cover all the basic syntax, and finally we'll show you a non-trivial example. Object-oriented programming is a method of programming where you package your code up into bundles of data and behavior. In Jython, you can define a template for this bundle with a class definition. With this first class written, you can then create instances of that class that include instance-specific data, as well as bits of code called methods that you can call to do things based on that data. This helps you organize your code into smaller, more manageable bundles. With the release of Jython 2.5, the differences in syntax between the C version of Python and Jython are negligible. So, although everything here covers Jython, you can assume that all of the same code will run on the C implementation of Python, as well. Enough introduction though—let's take a look at some basic syntax to see what this is all about.

Basic Syntax

Writing a class is simple. It is fundamentally about managing some kind of "state" and exposing some functions to manipulate that state. In object jargon, we call those functions "methods." Let's start by creating a Car class. The goal is to create an object that will manage its own location on a two-dimensional plane. We want to be able to tell it to turn and move forward, and we want to be able to interrogate the object to find out where its current location is. Place the following code in a file named "car.py."

Listing 6-1.

```
class Car(object):

    NORTH = 0
    EAST = 1
    SOUTH = 2
    WEST = 3

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self.direction = self.NORTH

    def turn_right(self):
        self.direction += 1
        self.direction = self.direction % 4

    def turn_left(self):
        self.direction -= 1
        self.direction = self.direction % 4

    def move(self, distance):
        if self.direction == self.NORTH:
            self.y += distance
```

```
        elif self.direction == self.SOUTH:
            self.y -= distance
        elif self.direction == self.EAST:
            self.x += distance
        else:
            self.x -= distance

    def position(self):
        return (self.x, self.y)
```

We'll go over that class definition in detail but right now, let's just see how to create a car, move it around, and ask the car where it is.

Listing 6-2.

```
from car import Car

def test_car():
    c = Car()
    c.turn_right()
    c.move(5)
    assert (5, 0) ==  c.position()

    c.turn_left()
    c.move(3)
    assert (5, 3) == c.position()
```

In Jython there are things that are "callables." Functions are one kind of callable; classes are another. So one way to think of a class is that it's just a special kind of function, one that creates object instances. Once we've created the car instance, we can simply call functions that are attached to the Car class and the object will manage its own location. From the point of view of our test code, we do not need to manage the location of the car—nor do we need to manage the direction that the car is pointing in. We just tell it to move, and it does the right thing. Let's go over the syntax in detail to see exactly what's going on here. In Line 1 of car.py, we declare that our Car object is a subclass of the root "object" class. Jython, like many object-oriented languages, has a "root" object that all other objects are based off of. This "object" class defines basic behavior that all classes can reuse. Jython actually has two kinds of classes: "new style" and old style. The old way of declaring classes didn't require you to type "object;" you'll occasionally see the old-style class usage in some Jython code, but it's not considered good practice. Just subclass "object" for any of your base classes and your life will be simpler. Lines 3 to 6 declare class attributes for the direction that any car can point to. These are class attributes, so they can be shared across all object instances of the Car object. Class attributes can be referenced without having to create an object instance. Now for the good stuff. Lines 8-11 declare the object initializer method. This method is called immediately after your object is created and memory for it has been allocated. In some languages, you might be familiar with a constructor; in Jython, we have an initializer which is run after construction. Valid method names in Jython are similar to many other C style languages. Generally, use method names that start with a letter; you can use numbers in the rest of the method name if you really want, but don't

use any spaces. Jython classes have an assortment of special “magic” methods as well. These methods all start with a double underscore and end with a double underscore. These methods are reserved by the language and they have special meaning. So for our initializer “__init__,” the Jython runtime will automatically invoke that method once you’ve called your constructor with “Car().” There are other reserved method names to let you customize your classes further, and we’ll get into those later. In our initializer, we are setting the initial position of the car to (0, 0) on a two-dimensional plane, and then the direction of the car is initialized to pointing north. When we initialize the object, we don’t have to pass in the position explicitly. The function signature uses Jython’s default argument list feature, so we don’t have to explicitly set the initial location to (0,0). Default arguments for methods work just the same as the default function arguments that were covered in Chapter 4. When the method is created, Jython binds the default values into the method so that, if nothing is passed in, the signature’s values will be used. There’s also a new argument introduced called “self.” This is a reference to the current object, the Car object. If you’re familiar with other C style languages, you might have called the reference “this.” Remember, your class definition is creating instances of objects. Once your object is created, it has its own set of internal variables to manage. Your object will inevitably need to access these, as well as any of the class internal methods. Jython will pass a reference to the current object as the first argument to all your instance methods. If you’re coming from some other object-oriented language, you’re probably familiar with the “this” variable. Unlike C++ or Java, Jython doesn’t magically introduce the reference into the namespace of accessible variables, but this is consistent with Jython’s philosophy of making things explicit for clarity. When we want to assign the initial x, y position, we just need to assign values on to the name “x”, and “y” on the object. Binding the values of x and y to self makes the position values accessible to any code that has access to self; namely, the other methods of the object. One minor detail here: in Jython, you can technically name the arguments however you want. There’s nothing stopping you from calling the first argument “this” instead of “self,” but the community standard is to use “self.” One of Jython’s strengths is its legibility and community standards around style. Lines 13 to 19 declare two methods to turn the vehicle in different directions. Notice how the direction is never directly manipulated by the caller of the Car object. We just asked the car to turn, and the car changed its own internal “direction” state. In Jython, you can specify private attributes by using a preceding double underscore, so self.direction would change to self.__direction. Once your object is instantiated, your methods can continue to access private attributes using the double underscore name, but external callers would not be able to easily access those private attributes. The attribute name will be mangled for external callers into “obj._Car__direction”. In practice, we don’t suggest using private attributes, because you cannot possibly know all the use cases your code may have to satisfy. If you want to provide a hint to other programmers that an attribute should be considered private, you can use a single underscore. Lines 21 to 29 define where the car should move to when we move the car forward. The internal direction variable informs the car how it should manipulate the x and y position. Notice how the caller of the Car object never needs to know precisely what direction the car is pointing in. The caller only needs to tell the object to turn and move forward. The particular details of how that message is used is abstracted away. That’s not too bad for a couple dozen lines of code. This concept of hiding internal details is called encapsulation. This is a core concept in object-oriented programming. As you can see from even this simple example, it allows you to structure your code so that you can provide a simplified interface to the

users of your code. Having a simplified interface means that we could have all kinds of behavior happening behind the function calls to turn and move, but the caller can ignore all those details and concentrate on using the car instead of managing the car. As long as the method signatures don't change, the caller really doesn't need to care about any of that. Let's extend the class definition now to add persistence so we can save and load the car's state to disk. The goal here is to add it without breaking the existing interface to our class. First, pull in the pickle module. Pickle will let us convert Jython objects into byte strings that can be restored to full objects later.

Import pickle

Now, just add two new methods to load and save the state of the object.

Listing 6-3.

```
def save(self, filename):
    state = (self.direction, self.x, self.y)
    pickle.dump(state, open(filename,'wb'))

def load(self, filename):
    state = pickle.load(open(filename,'rb'))
    (self.direction, self.x, self.y) = state
```

Simply add calls to save() at the end of the turn and move methods and the object will automatically save all the relevant internal values to disk. There's a slight problem here: we need to have different files for each of our cars; our load and save methods have explicit filename arguments but our objects themselves don't have any notion of a name. Let's modify the intializer so that we always have a name bound into the object. Change __init__ to accept a name argument.

Listing 6-4.

```
def __init__(self, name, x=0, y=0):
    self.name = name
    self.x = x
    self.y = y
    self.direction = self.NORTH
```

People who use the Car object don't even need to know that it's saving to disk, because the car object handles it behind the scenes.

Listing 6-5.

```
def turn_right(self):
    self.direction += 1
    self.direction = self.direction % 4
    self.save(self.name)
```

```
def turn_left(self):
    self.direction -= 1
    self.direction = self.direction % 4
    self.save(self.name)

def move(self, distance):
    if self.direction == self.NORTH:
        self.y += distance
    elif self.direction == self.SOUTH:
        self.y -= distance
    elif self.direction == self.EAST:
        self.x += distance
    else:
        self.x -= distance
    self.save(self.name)
```

Now, when you call the turn, or move methods, the car will automatically save itself to disk. If you want to reconstruct the car object's state from a previously saved pickle file, you can simply call the load() method and pass in the string name of your car.

Object Attribute Lookups

If you've been paying attention, you're probably wondering how the NORTH, SOUTH, EAST and WEST variables got bound to self. We never actually assigned them to the self variable during object initialization—so what's going on when we call move()? How is Jython actually resolving the value of those four variables? Now seems like a good time to show how Jython resolves name lookups. The direction names actually got bound to the car class. The Jython object system does a little bit of magic when you try accessing any name against an object, it first searches for anything that was bound to "self." If Jython can't resolve any attribute on self with that name, it goes up the object graph to the class definition. The direction attributes NORTH, SOUTH, EAST, WEST were bound to the class definition, so the name resolution succeeds and we get the value of the class attribute. A very short example will help clarify this.

Listing 6-6.

```
>>> class Foobar(object):
...     def __init__(self):
...         self.somevar = 42
...     class_attr = 99
...
>>>
>>> obj = Foobar()
>>> obj.somevar
42
>>> obj.class_attr
99
>>> obj.not_there
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foobar' object has no attribute 'not_there'
```

```
>>>
```

So the key difference here is what you bind a value to. The values you bind to self are available only to a single object. Values you bind to the class definition are available to all instances of the class. The sharing of class attributes among all instances is a critical distinction, because mutating a class attribute will affect all instances. This may cause unintended side effects if you're not paying attention as a variable may change value on you when you aren't expecting it to.

Listing 6-7.

```
>>> other = Foobar()
>>> other.somevar
42
>>> other.class_attr
99
>>> # obj and other can have different values for somevar
>>> obj.somevar = 77
>>> obj.somevar
77
>>> other.somevar
42
>>> # If we assign to other.class_attr, that makes an instance attribute of
other called class_attr.
>>> other.class_attr = 66
>>> other.class_attr
66
>>> # And doesn't change the class_attribute class_attr for other objects
>>> obj.class_attr
99
>>> # You can still get at the class attribute from other by looking at
other.  class  .class attr
>>> other.__class__.class_attr
99
>>> # and if you remove the instance attribute other.class_attr,
>>> then other.class_attr goes back to referring to the class attribute
>>> del other.class_attr

>>> other.class_attr

99

>>> # But if the class_attribute is mutable, when you change it, you change
it for every instance
>>> Foobar.class_list = []

>>> obj.class_list

[]

>>> other.class_list

[]
```

```
>>> obj.class_list.append(1)

>>> obj.class_list

[1]

>>> other.class_list

[1]
```

We think it's important to stress just how transparent Jython's object system really is. Object attributes are just stored in a plain Jython dictionary. You can directly access this dictionary by looking at the __dict__ attribute.

Listing 6-8.

```
>>> obj = Foobar()
>>> obj.__dict__
{'somevar': 42}
```

Notice that there are no references to the methods of the class (in this case, just our initializer), or the class attribute 'class_attr'. The __dict__ only shows the local attributes and methods of the object. We'll cover inheritance shortly, and you'll see how attributes and methods are looked up in the case where you specialize classes through subclassing. The same trick can be used to inspect all the attributes of the class, just look into the __dict__ attribute of the class definition and you'll find your class attributes and all the methods that are attached to your class definition:

Listing 6-9.

```
>>> Foobar.__dict__
{'__module__': '__main__',
    'class_attr': 99,
    '__dict__': <attribute '__dict__' of 'Foobar' objects>,
    '__init__': <function __init__ at 1>}
```

This transparency can be leveraged with dynamic programming techniques using closures and binding new functions into your class definition at runtime. We'll revisit this later in the chapter when we look at generating functions dynamically and finally with a short introduction to metaprogramming.

Inheritance and Overloading

In the car example, we subclass from the root object type. You can also subclass your own classes to specialize the behavior of your objects. You may want to do this if you notice that your code naturally has a structure where you have many different classes that all share some common behavior. With objects, you can write one class, and then reuse it using inheritance to automatically gain access to the pre-existing behavior and attributes of the parent class. Your "base" objects will inherit behavior from the root "object" class, but any subsequent subclasses will inherit from your own classes. Let's take a simple example of using some animal classes to see how this works. Define a module "animals.py" with the following code:

Listing 6-10.

```
class Animal(object):
    def sound(self):
        return "I don't make any sounds"
class Goat(Animal):
    def sound(self):
        return "Bleeattt!"
class Rabbit(Animal):
    def jump(self):
        return "hippity hop hippity hop"
class Jackalope(Goat, Rabbit):
    pass
```

Now you should be able to explore that module with the jython interpreter:

Listing 6-11.

```
>>> from animals import *
>>> animal = Animal()
>>> goat = Goat()
>>> rabbit = Rabbit()
>>> jack = Jackalope()
>>> animal.sound()
"I don't make any sounds"
>>> animal.jump()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Animal' object has no attribute 'jump'
>>> rabbit.sound()
"I don't make any sounds"
>>> rabbit.jump()
'hippity hop hippity hop'
>>> goat.sound()
'Bleeattt!'
>>> goat.jump()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Goat' object has no attribute 'jump'
>>> jack.jump()
```

```
'hippity hop hippity hop'
>>> jack.sound()
'Bleeattt!'
```

Inheritance is a very simple concept, when you declare your class, you simply specify which parent classes you would like to reuse. Your new class can then automatically access all the methods and attributes of the super class. In this example, the Goat object has no method jump, and its super class Animal has no method jump, so the attempt to invoke the jump method fails. Invoking the sound method on the rabbit actually calls the super class's sound method. This is the key idea: if an attribute lookup fails on the local object instance, the lookup is then propagated up the inheritance tree to the super class. Notice how the Jackalope had access to methods from both the rabbit and the goat because it can use two super classes to resolve methods. With single inheritance—when your class simply inherits from one parent class—the rules for resolving where to find an attribute or a method are very straightforward. Jython just looks up to the parent if the current object doesn't have a matching attribute. It's important to point out now that the Rabbit class is a type of Animal: the Jython runtime can tell you that programmatically by using the isinstance function:

Listing 6-12.

```
>>> isinstance(bunny, Rabbit)
True
>>> isinstance(bunny, Animal)
True
>>> isinstance(bunny, Goat)
False
```

For many classes, you may want to extend the behavior of the parent class instead of just completely overriding it. For this, you'll want to use the super() function. Let's specialize the Rabbit class like this:

Listing 6-13.

```
class EasterBunny(Rabbit):
    def sound(self):
        orig = super(EasterBunny, self).sound()
        return "%s - but I have eggs!" % orig
```

If you now try making this rabbit speak, it will extend the original sound() method from the base Rabbit class. Calling the super() function lets you access the super class's implementation of the sound method. In this example, it's useful because the EasterBunny class is reusing and extending the basic Rabbit class's sound() method.

Listing 6-14.

```
>>> bunny = EasterBunny()
>>> bunny.sound()
```

```
"I don't make any sounds - but I have eggs!"
```

That wasn't so bad. For these examples, we only demonstrated that inherited methods can be invoked, but you can do exactly the same thing with attributes that are bound to the self. For multiple inheritance, things get complicated quickly. Jython uses "left first, depth first" search to resolve attribute lookups. In a nutshell, if you were to draw your inheritance diagram, Jython would look down the left side of your graph looking for attributes going from the bottom up, left to right. If any super class is inherited by two or more subclasses, then the super class is used for lookup only after all attribute lookups have been exhausted on the subclasses.

Underscore Methods

Abstraction using plain classes is wonderful and all, but it's even better if your code seems to naturally fit into the syntax of the language. Jython supports a variety of underscore methods: methods that start and end with double "_" signs that let you overload the behavior of your objects. This means that your objects will seem to integrate more tightly with the language itself. You have already seen one such method: __init__. With the underscore methods, you can give you objects behavior for logical and mathematical operations. You can even make your objects behave more like standard builtin types like lists, sets or dictionaries. Let's start with adding simple unicode extensions to a SimpleObject to see the most simple example of this. Then we'll move on to building customized container classes.

Listing 6-15.

```
from __future__ import with_statement
from contextlib import closing
with closing(open('simplefile','w')) as fout:
    fout.writelines(["blah"])
with closing(open('simplefile','r')) as fin:
    print fin.readlines()
```

This snippet of code just opens a file, writes a little bit of text, and then we read the contents out. Not terribly exciting. Most objects in Jython are serializable to strings using the pickle module. The pickle module lets us convert our live Jython objects into byte streams that can be saved to disk and later restored into objects. Let's see the functional version of this:

Listing 6-16.

```
from __future__ import with_statement
from contextlib import closing
from pickle import dumps, loads

def write_object(fout, obj):
    data = dumps(obj)
    fout.write("%020d" % len(data))
    fout.write(data)
```

```
def read_object(fin):
    length = int(fin.read(20))
    obj = loads(fin.read(length))
    return obj

class Simple(object):
    def __init__(self, value):
        self.value = value
    def __unicode__(self):
        return "Simple[%s]" % self.value

with closing(open('simplefile','wb')) as fout:
    for i in range(10):
        obj = Simple(i)
        write_object(fout, obj)

print "Loading objects from disk!"
print '=' * 20

with closing(open('simplefile','rb')) as fin:
    for i in range(10):
        print read_object(fin)
```

This should output something like this:

Listing 6-17.

```
Loading objects from disk!
====================
Simple[0]
Simple[1]
Simple[2]
Simple[3]
Simple[4]
Simple[5]
Simple[6]
Simple[7]
Simple[8]
Simple[9]
```

So now we're doing something interesting. Let's look at exactly what happening here. First, you'll notice that the Simple object is rendering nicely: the Simple object can render itself using the __unicode__ method. This is clearly an improvement over the earlier rendering of the object with angle brackets and a hex code. The write_object function is fairly straightforward, we're just converting our objects into strings using the pickle module, computing the length of the string and then writing the length and the actual serialized object to disk. This is fine, but the read side is a bit clunky. We don't really know when to stop reading. We can fix this using the iteration protocol. Which bring us to one of my favorite reasons to use objects at all in Jython.

Protocols

In Jython, we have “duck typing.” If it walks like a duck, quacks like a duck, and looks like a duck, it’s a duck. This is in stark contrast to more rigid languages like C# or Java which have formal interface definitions. One of the nice benefits of having duck typing is that Jython has the notion of object protocols. If you happen to implement the right methods, Jython will recognize your object as a certain type of ‘thing’. Iterators are objects that look like lists that let you read the next object. Implementing an iterator protocol is straightforward: just implement a next() method and a __iter__ method, and you’re ready to rock and roll. Let’s see this in action:

Listing 6-18.

```
class PickleStream(object):
    """
    This stream can be used to stream objects off of a raw file stream
    """
    def __init__(self, file):
        self.file = file

    def write(self, obj):
        data = dumps(obj)
        length = len(data)
        self.file.write("%020d" % length)
        self.file.write(data)

    def __iter__(self):
        return self

    def next(self):
        data = self.file.read(20)
        if len(data) == 0:
            raise StopIteration
        length = int(data)
        return loads(self.file.read(length))

    def close(self):
        self.file.close()
```

This class will let you wrap a simple file object and you can now send it raw Jython objects to write to a file, or you can read objects out as if the stream was just a list of objects. Writing and reading becomes much simpler:

Listing 6-19.

```
with closing(PickleStream(open('simplefile','wb'))) as stream:
    for i in range(10):
        obj = Simple(i)
        stream.write(obj)

with closing(PickleStream(open('simplefile','rb'))) as stream:
    for obj in stream:
        print obj
```

Abstracting out the details of serialization into the PickleStream lets us “forget” about the details of how we are writing to disk. All we care about is that the object will do the right thing when we call the write() method. The iteration protocol can be used for much more advanced purposes, but even with this example, it should be obvious how useful it is. While you could implement the reading behavior with a read() method, just using the stream as something you can loop over makes the code much easier to understand. Let’s step back now and look at some of the other underscore methods. Two of the most common uses of underscore methods are to implement proxies and to implement your own container-like classes. Proxies are very useful in many programming problems. You use a proxy to act as an intermediary between a caller and a callee. The proxy class can add in extra behavior in a manner that is transparent to the caller. In Jython, you can use the __getattr__ method to implement attribute lookups if a method or attribute does not seem to exist.

Listing 6-20.

```
class SimpleProxy(object):
    def __init__(self, parent):
        self._parent = parent

    def __getattr__(self, key):
        return getattr(self._parent, key)
```

That represents the simplest (and not very useful) proxy. Any lookup for an attribute that doesn’t exist on SimpleProxy will automatically invoke the __getattr__ method, and the lookup will then be delegated to the parent object. Note that this works for attributes that are not underscore attributes. Let’s look at a simple example to make this clearer.

Listing 6-21.

```
>>> class TownCrier(object):
...     def __init__(self, parent):
...         self._parent = parent
...     def __getattr__(self, key):
...         print "Accessing : [%s]" % key
...         return getattr(self._parent, key)
...
>>> class Calc(object):
...     def add(self, x, y):
...         return x + y
...     def sub(self, x, y):
...         return x - y
...
>>> calc = Calc()
>>> crier = TownCrier(calc)
>>> crier.add(5,6)
Accessing : [add]
11
>>> crier.sub(3,6)
Accessing : [sub]
-3
```

Here, we can see that our TownCrier class is delegating control to the Calculator object whenever a method is invoked, but we are also adding in some debug messages along the way. Unlike a language like Java where you would need to implement a specific interface (if one even exists), in Jython, creating a proxy is nearly free. The __getattr__ method is automatically invoked if attribute lookups fail using the normal lookup mechanism. Proxies provide a way for you to inject new behavior by using a delegation pattern. The advantage here is that you can add new behavior without having to know anything about the delegate's implementation; something that you'd have to deal with if you used subclassing. The second common use of underscore methods we'll cover is implementing your own container class. We'll take a look at implementing a small dictionary-like class. Suppose we have class that behaves like a regular dictionary, but it logs all read access to a file. To get the basic behavior of a dictionary, we need to be able to get, set and delete key/value pairs, check for key existence and count the number of records in the dictionary. To get all of that behavior, we will need to implement the following methods:

Listing 6-22.

```
__getitem__(self, key)
__setitem__(self, key, value)
__delitem__(self, key)
__contains__(self, item)
__len__(self)
```

The method names are fairly self-explanatory. __gettiem__, __setitem__ and __delitem__ all manipulate key/value pairs in our dictionary. We can implement this behavior on top of a regular list object to get a naïve implementation. Put the following code into a file named "foo.py."

Listing 6-23.

```
class  SimpleDict(object):
    def __init__(self):
        self._items = []

    def __getitem__(self, key):
       # do a brute force key lookup and return the value
        for k, v in self._items:
            if k == key:
                return v
        raise LookupError, "can't find key: [%s]" % key

    def __setitem__(self, key, value):
        # do a brute force search and replace
        # for the key if it exists. Otherwise append
        # a new key/value pair.
        for i, (k , v) in enumerate(self._items):
            if k == key:
                self._items[i][1] = v
                return
        self._items.append((key, value))
```

```
    def __delitem__(self, key):
        # do a brute force search and delete
        for i, (k , v) in enumerate(self._items):
            if k == key:
                del self._items[i]
                return
       raise LookupError, "Can't find [%s] to delete" % key
```

The implementations listed previously are naïve, but they should illustrate the basic pattern of usage. Once you have just those three methods implemented, you can start using dictionary style attribute access.

Listing 6-24.

```
>>> from foo import *
>>> x = SimpleDict()
>>> x[0] = 5
>>> x[15]= 32
>>> print x[0]
5
>>> print x[15]
32
```

To get two remaining behaviors, key existence and dictionary size, we fill in the __contains__ and __len__ methods.

Listing 6-25.

```
def __contains__(self, key):
    return key in [k for (k, v) in self._items]

def __len__(self):
    return len(self._items)
```

Now this implementation of a dictionary will behave almost identically to a standard dictionary, we're still missing some "regular" methods like items(), keys() and values(), but accessing the SimpleDict using square brackets will work the way you expect a dictionary to work. While this implementation was intentionally made to be simple, it is easy to see that we could have saved our items into a text file, a database backend or any other backing storage. The caller would be blind to these changes; all they would interact with is the dictionary interface.

Default Arguments

One particular snag that seems to catch every Jython programmer is when you use default values in a method signature.

Listing 6-26.

```
>>> class Tricky(object):
...     def mutate(self, x=[]):
...         x.append(1)
...         return x
...
>>> obj = Tricky()
>>> obj.mutate()
[1]
>>> obj.mutate()
[1, 1]
>>> obj.mutate()
[1, 1, 1]
```

What's happening here is that the instance method "mutate" is an object. The method object stores the default value for "x" in an attribute inside the method object. To complicate things further, method objects are bound to your class definition. So when you go and mutate the list, you're actually changing the value of an attribute of the method itself. Each of your object instances point to the same class definition, and the same method. Your default arguments will change for all of your instances!

Runtime Binding of Methods

One interesting feature of Jython is that instance methods are actually just attributes hanging off of the class definition; the functions are just attributes like any other variable, except that they happen to be "callable." It's even possible to create and bind in functions to a class definition at runtime using the new module to create instance methods. In the following example, you can see that it's possible to define a class with nothing in it, and then bind methods to the class definition at runtime.

Listing 6-27.

```
>>> def some_func(self, x, y):
...     print "I'm in object: %s" % self
...     return x * y
...
>>> import new
>>> class Foo(object): pass
...
>>> f = Foo()
>>> f
<__main__.Foo object at 0x1>
>>> Foo.mymethod = new.instancemethod(some_func, f, Foo)
>>> f.mymethod(6,3)
I'm in object: <__main__.Foo object at 0x1>
```

```
18
```

When you invoke the mymethod method, the same attribute lookup machinery is being invoked. Jython looks up the name against the “self” object. When it can’t find anything there, it goes to the class definition. When it finds it there, the instancemethod object is returned. The function is then called with two arguments and you get to see the final result. The special function new.instancemethod is doing some magic so that when some_func is invoked, the Jython runtime will automatically pass in the object instance as the first argument. That’s the self attribute we saw earlier in this chapter. Functions that are bound to an object in this manner are appropriately called “bound methods.” Without this binding behavior, the object instance will not be passed in as the first argument. In this case, the method would be called an “unbound method.” This kind of dynamism in Jython is extremely powerful. You can write code that generates functions at program runtime, and then bind those functions to objects. You can do all of this because in Jython, classes and functions are what are known as “first-class objects.” The class definition itself is an actual object, just like any other object. Manipulating classes is as easy as manipulating any other object. The practical use of this kind of technique is when you are building tools that generate code. Instead of statically code generating functions and methods, you can “grow” your methods depending on runtime features of your objects. This is how most of the Python database toolkits work. You define classes that represent objects in your database, and the toolkit will inspect your objects and enhance the classes with persistence behavior. Using dynamic programming techniques, like creating new methods at runtime, opens up the possibility of literally post-processing your classes.

Caching Attribute Access

Suppose we have some method that requires intensive computational resources to run, but the results do not vary much over time. Wouldn’t it be nice if we could cache the results so that the computation wouldn’t have to run each and every time? We can leverage the decorator pattern in Chapter 4 and add write the results of our computations as new attributes of our objects. Here’s our class with a slow computation method. The slow_compute() method really doesn’t do anything interesting; it just sleeps and eats up one second of time. We’re going to wrap the method up with a caching decorator so that we don’t have to wait the one second every time we invoke the method.

Listing 6-28.

```
import time
class Foobar(object):
    def slow_compute(self, *args, **kwargs):
        time.sleep(1)
        return args, kwargs, 42
```

Now let’s cache the value using a decorator function. Our strategy is that for any function named X with some argument list, we want to create a unique name and save the final computed value to that name. We want our cached value to have a human readable name, we

want to reuse the original function name, as well as the arguments that were passed in the first time. Let's get to some code!

Listing 6-29.

```
import hashlib
def cache(func):
    """
    This decorator will add a  cache functionName HEXDIGEST
    attribute after the first invocation of an instance method to
    store cached values.
    """
    # Obtain the function's name
    func_name = func.func_name
    # Compute a unique value for the unnamed and named arguments
    arghash = hashlib.sha1(str(args) + str(kwargs)).hexdigest()
    cache_name = '_cache_%s_%s' % (func_name, arghash)

    def inner(self, *args, **kwargs):
        if hasattr(self, cache_name):
            # If we have a cached value, just use it
            print "Fetching cached value from : %s" % cache_name
            return getattr(self, cache_name)
        result = func(self, *args, **kwargs)
        setattr(self, cache_name, result)
        return result
    return inner
```

There are only two new tricks that are in this code.

1.We're using the hashlib module to convert the arguments to the function into a unique single string. 2.We're using getattr, hasattr, and setattr to manipulate the cached value on the instance object.

The three functions getattr, setattr, and hasattr allow you to get, set, and test for attributes on an object by using string names instead of symbols. So accessing foo.bar is equivalent to invoking getattr(foo, 'bar'). In the previous case, we're using the attribute functions to bind the result of the slow calculation function into an attribute of an instance of Foobar. The next time the decorated method is invoked, the hasattr test will find the cached value and we return the precomputed value. Now, if we want to cache the slow method, we just throw on a @cache line above the method declaration.

Listing 6-30.

```
@cache
def slow_compute(self, *args, **kwargs):
    time.sleep(1)
    return args, kwargs, 42
```

Fantastic! We can reuse this cache decorator for any method we want now. Let's suppose now that we want our cache to invalidate itself after every N number of calls. This practical use of currying is only a slight modification to the original caching code. The goal is the same; we are going to store the computed result of a method as an attribute of an object. The name of the attribute is determined based on the actual function name, and is concatenated with a hash string computed by using the arguments to the method. In the code sample, we'll save the function name into the variable "func_name" and we'll save the argument hash value into "arghash." Those two variables will also be used to compute the name of a counter attribute. When the counter reaches N, we'll clear out the precomputed value so that the calculation can run again.

Listing 6-31.

```
import hashlib
def cache(loop_iter):
    def function_closure(func):
        func_name = func.func_name

        def closure(self, loop_iter, *args, **kwargs):
            arghash = hashlib.sha1(str(args) + str(kwargs)).hexdigest()
            cache_name = '_cache_%s_%s' % (func_name, arghash)
            counter_name = '_counter_%s_%s' % (func_name, arghash)

            if hasattr(self, cache_name):
                # If we have a cached value, just use it
                print "Fetching cached value from : %s" % cache_name
                loop_iter -= 1
                setattr(self, counter_name, loop_iter)
                result = getattr(self, cache_name)

                if loop_iter == 0:
                    delattr(self, counter_name)
                    delattr(self, cache_name)
                    print "Cleared cached value"
                return result

            result = func(self, *args, **kwargs)
            setattr(self, cache_name, result)
            setattr(self, counter_name, loop_iter)
            return result

        return closure

    return function_closure
```

Now we're free to use @cache for any slow method and caching will come in for free, including automatic invalidation of the cached value. Just use it like this:

Listing 6-32.

```
@cache(10)
```

```
def slow_compute(self, *args, **kwargs):
    # TODO: stuff goes here...
    pass
```

Summary

Now, we're going to ask you to use your imagination a little. We've covered quite a bit of ground really quickly. We can: look up attributes in an object (use the __dict__ attribute); check if an object belongs to a particular class hierarchy (use the isinstance function); build functions out of other functions using currying and even bind those functions to arbitrary names. This is fantastic. We now have all the basic building blocks we need to generate complex methods based on the attributes of our class. Imagine a simplified addressbook application with a simple contact.

Listing 6-33.

```
class Contact(object):
    first_name = str
    last_name = str
    date_of_birth = datetime.Date
```

Assuming we know how to save and load to a database, we can use the function generation techniques to automatically generate load() and save() methods and bind them into our Contact class. We can use our introspection techniques to determine what attributes need to be saved to our database. We could even grow special methods onto our Contact class so that we could iterate over all of the class attributes and magically grow 'searchby_first_name' and 'searchby_last_name' methods. Jython's flexible object system allows you to write code that has a deep ability to introspect itself by simply looking up information in dictionaries like __dict__. You also have the ability to rewrite parts of your classes using decorators and even creating new instance methods at runtime. These techniques can be combined together to write code that effectively rewrites itself. This technique is called 'metaprogramming'. This technique is very powerful: we can write extremely minimal code, and we can code generate all of our specialized behavior. In the case of our contact, it would "magically" know how to save itself, load itself, and delete itself from a database. This is precisely how the database mappers in Django and SQLAlchemy work: they rewrite parts of your program to talk to a database.

Source: http://www.jython.org/jythonbook/en/1.0/ObjectOrientedJython.html