Object Oriented Programming with PYTHON (part two)

By icarus

Table of Contents

Back To Work

Last time out, I gave you a crash course in object−oriented programming in the context of Python development. I explained what classes and instances were, showed you how to construct a class and create instances from it, demonstrated how to access class methods and properties, and tossed in a few real and not−so−real examples.

In this concluding article, I'll be deliving deeper into the topic, with a look at inheritance, destructors and overrides. This may sound complex at first glance, but I've done my best to make it easy to understand and apply − so come on in and tell me what you think!

The Family Tree

One of the main virtues of object−oriented programming is that it allows you to re−use existing objects, and add new capabilities to them − a feature referred to as "inheritance". By creating a new object which inherits the properties and methods of an existing object, developers can build on existing Python classes, thereby reducing both development and testing time.

Python allows you to derive a new class from an existing class by specifying the name of the base class within parentheses while defining the new class. So, if I wanted to derive a new class named evenBiggerSnake() from the base class veryBigSnake(), my class definition would look something like this:

```
class evenBiggerSnake(veryBigSnake):

# method and property definitions
```

You can inherit from more than one base class as well.

```
class evenBiggerSnake(veryBigSnake, veryBigBird, veryBigFish):

# method and property definitions
```

A derived class functions in exactly the same manner as any other class, with one minor change: in the event that a method or property accessed by an object is not found in the derived class, Python will automatically search the base class (and the base class's ancestors, if any exist) for that particular method or property.

As an example, let's create the new evenBiggerSnake() class, which inherits from the base class veryBigSnake().

```
class veryBigSnake:

# constructor
# now accepts name and type as arguments
def __init__(self, name="Peter Python", type="python"):
self.name = name
self.type = type
print "New snake in da house!"


# function to set snake name
def set_snake_name(self, name):
self.name = name

# function to set snake type
```

```
def set_snake_type(self, type):
self.type = type

# function to display name and type
def who_am_i(self):
print "My name is " + self.name + ", I'm a " + self.type + "
and I'm
perfect for you! Take me home today!"

class evenBiggerSnake(veryBigSnake):
pass
```

At this point, you should be able to do this

```
>>> alpha = evenBiggerSnake()
New snake in da house!
>>> alpha.who_am_i()
My name is Peter Python, I'm a python and I'm perfect for you!
Take me home
today!
>>> alpha.set_snake_name("Roger Rattler")
>>> alpha.set_snake_type("rattlesnake")
>>> alpha.who_am_i()
My name is Roger Rattler, I'm a rattlesnake and I'm perfect
for you! Take
me home today!
>>>
```

and have the code work exactly as before, despite the fact that you are now using the evenBiggerSnake() class. This indicates that the class evenBiggerSnake() has successfully inherited the properties and methods of the base class veryBigSnake().

This is sometimes referred to as the "empty sub−class test" − essentially, a new class which functions exactly like the parent class, and can be used as a replacement for it.

Note also that the derived class automatically inherits the base class's constructor if it doesn't have one of its own. However, if I did explicitly define a constructor for the derived class, this new constructor would override the base class's constructor.

```
class evenBiggerSnake(veryBigSnake):

# constructor
# accepts name, age and type as arguments
def __init__(self, name="Paul Python", type="python",
age="2"):
```

```
self.name = name
self.age = age
self.type = type
print "A new, improved snake has just been born"
```

Look what happens when I create an instance of the class now:

```
>>> alpha = evenBiggerSnake()
A new, improved snake has just been born
>>> alpha.name
'Paul Python'
>>> alpha.age
'2'
>>> alpha.who_am_i()
My name is Paul Python, I'm a python and I'm perfect for you!
Take me home
today!
>>>
```

This is true of other methods too − look what happens when I define a new who_am_i() method for the evenBiggerSnake() class:

```
class evenBiggerSnake(veryBigSnake):

# constructor
# accepts name, age and type as arguments
def __init__(self, name="Paul Python", type="python",
age="2"):
self.name = name
self.age = age
self.type = type
print "A new, improved snake has just been born"

# modified function to display name, age and type
def who_am_i(self):
print "My name is " + self.name + ", I'm a " + self.type +
" and I'm just " + self.age + " years old"
```

Here's the output:

```
>>> alpha = evenBiggerSnake()
A new, improved snake has just been born
>>> alpha.who_am_i()
```

```
My name is Paul Python, I'm a python and I'm just 2 years old
>>>
```

Alarm Bells

So that's the theory − now let's see it in action. The first order of business is to create a new AlarmClock() class, derived from the base class Clock(). You may remember this from the first part of this article − if not, here's a reminder:

```
# a simple clock class
# each Clock object is initialized with offsets (hours and
minutes)
# indicating the difference between GMT and local time

class Clock:

# constructor
def __init__(self, offsetSign, offsetH, offsetM, city):

# set variables to store timezone offset
# from GMT, in hours and minutes, and city name
self.offsetSign = offsetSign
self.offsetH = offsetH
self.offsetM = offsetM
self.city = city

# print message
print "Clock created"


# method to display current time, given offsets
def display(self):

# use the gmtime() function, used to convert local time to GMT
# import required methods from the time module
# returns an array
from time import time, gmtime

self.GMTTime = gmtime(time())

self.seconds = self.GMTTime[5]
self.minutes = self.GMTTime[4]
self.hours = self.GMTTime[3]

# calculate time
if(self.offsetSign == '+'):
# city time is ahead of GMT
self.minutes = self.minutes + self.offsetM

if (self.minutes > 60):
```

```
self.minutes = self.minutes − 60
self.hours = self.hours + 1

self.hours = self.hours + self.offsetH

if (self.hours >= 24):
self.hours = self.hours − 24

else:
# city time is behind GMT
self.seconds = 60 − self.seconds
self.minutes = self.minutes − self.offsetM

if (self.minutes < 0):
self.minutes = self.minutes + 60
self.hours = self.hours − 1

self.hours = self.hours − self.offsetH

if (self.hours < 0):
self.hours = 24 + self.hours


# make it look pretty and display it
self.localTime = str(self.hours) + ":" + str(self.minutes) +
":" +
str(self.seconds)
print "Local time in " + self.city + " is " + self.localTime

# that's all, folks!
```

And here's the derived class:

```
# a derived clock class
# each AlarmClock object is initialized with offsets (hours
and minutes)
# indicating the difference between GMT and local time

class AlarmClock(Clock):
pass

# that's all, folks!
```

Let's just verify that the new class has inherited all the methods and properties of the base class correctly.

```
>>> london = AlarmClock("+", 0, 00, "London")
Clock created
>>> london.display()
Local time in London is 8:52:21
>>>
```

Great! Next, let's add a new method to our derived class.

```
class AlarmClock(Clock):

# resets clock to display GMT
def reset_to_gmt(self):
self.offsetSign = "+"
self.offsetH = 0
self.offsetM = 0
self.city = "London"
print "Clock reset to GMT!"
```

And now, when I use it, here's what I'll see:

```
>>> bombay = AlarmClock("+", 5, 30, "Bombay")
Clock created
>>> bombay.display()
Local time in Bombay is 16:45:32
>>> bombay.reset_to_gmt()
Clock reset to GMT!
>>> bombay.display()
Local time in London is 11:15:39
>>>
```

So we have an AlarmClock() class which inherits methods from a base Clock() class while simultaneously adding its own specialized methods. Ain't that just dandy?

Under The Microscope

A number of built−in functions are available to help you navigate Python's classes and objects.

The most basic task involves distinguishing between classes and instances − and the type() function can help here. Take a look:

```
>>> type(veryBigSnake)
<type 'class'>
>>> beta = veryBigSnake("Vanessa Viper", "viper")
New snake in da house!
>>> type(beta)
<type 'instance'>
>>>
```

You may already be familiar with the dir() function, which returns a list of object properties and methods − look what it says when I run it on a class

```
>>> dir(veryBigSnake)
['__del__', '__doc__', '__init__', '__module__',
'set_snake_name',
'set_snake_type', 'who_am_i']
>>>
```

and on an object of that class.

```
>>> dir(beta)
['name', 'type']
>>>
```

Every class also exposes the __bases__ property, which holds the name(s) of the class(es) from which this particular class has been derived. Most of the time, this property does not contain a value; it's only useful if you're working with classes which inherit methods and properties from each other.

```
>>> # base class − has no ancestors
>>> veryBigSnake.__bases__
()
>>> # derived class − has base class
>>> evenBiggerSnake.__bases__
(<class snake.veryBigSnake at 80d5c08>,)
>>>
```

If you'd like to see the values of a specific instance's properties, you can use the instance's __dict__ property, which returns a dictionary of name−value pairs,

```
>>> beta.__dict__
{'name': 'Vanessa Viper', 'type': 'viper'}
>>>
```

while the corresponding __class__ property identifies the class from which this instance was spawned.

```
>>> beta.__class__
<class snake.veryBigSnake at 80cda20>
>>>
```

Chaos And Destruction

In Python, an object is automatically destroyed once the references to it are no longer in use, or when the Python script completes execution. A destructor is a special function which allows you to execute commands immediately prior to the destruction of an object.

You do not usually need to define a destructor − but if you want to see what it looks like, take a look at this:

```
class veryBigSnake:

# constructor
# now accepts name and type as arguments
def __init__(self, name="Peter Python", type="python"):
self.name = name
self.type = type
print "New snake in da house!"


# function to set snake name
def set_snake_name(self, name):
self.name = name

# function to set snake type
def set_snake_type(self, type):
self.type = type

# function to display name and type
def who_am_i(self):
print "My name is " + self.name + ", I'm a " + self.type + "
and I'm
perfect for you! Take me home today!"

# destructor
def __del__(self):
print "Just killed the snake named " + self.name + "!"
```

Note that a destructor must always be called __del__()

Here's a demonstration of how to use it:

```
>>> alpha = veryBigSnake("Bobby Boa", "boa constrictor")
New snake in da house!
>>> beta = veryBigSnake("Alan Adder", "harmless green adder")
New snake in da house!
>>> del beta
```

```
Just killed the snake named Alan Adder!
>>> del alpha
Just killed the snake named Bobby Boa!
>>>
```

And with multiple murder on my hands, it's now time to bid you goodbye. If you're interested in the more arcane aspects of Python's OO capabilities − operator overloading, private and public variables, and so on − you should consider visiting the following sites:

The official Python tutorial, at http://www.python.org/doc/current/tut/node11.html

The Python Cookbook, at http://aspn.activestate.com/ASPN/Cookbook/Python

The Vaults of Parnassus, at http://www.vex.net/parnassus/

Python HOWTOs, at http://py−howto.sourceforge.net/

The Python FAQ, at http://www.python.org/doc/FAQ.html

Until next time...stay healthy!

Note: All examples in this article have been tested on Linux/i586 with Python 1.5.2. Examples are illustrative only, and are not meant for a production environment. YMMV!\n