Object Oriented Programming with PYTHON (part one)

By icarus

Table of Contents

Treating Pythons As Objects

You're probably tired of hearing me say it, but indulge me one more time − when it comes to object−oriented programming, very few languages have Python's capabilities.

This is primarily because Python was built from the ground up as an object−oriented language, and so provides built−in constructs that make it simple for developers to structure code for maximum reusability. Python's "dynamic typing", which automatically recognizes objects like numbers, strings and lists, and negates the need to declare variable types and sizes, offers an advantage not found in languages like C or Java, while automatic memory allocation and management, together with a vast array of pluggable libraries and high−level abstractions, complete the picture.

Over the course of this two−part article, I'm going to take an in−depth look at Python's OO capabilities, together with examples and explanations to demonstrate just how powerful it really is. I'll be covering most of the basics − classes, objects, attributes and methods − and a couple of the advanced concepts − constructors, destructors and inheritance. And if you're new to object−oriented programming, or just apprehensive about what lies ahead, don't worry − I promise this will be a lot easier than you think.

Let's get going!

The Basics

Before we begin, let's just go over the basics quickly:

In Python, a "class" is simply a set of program statements which perform a specific task. A typical class definition contains both variables and functions, and serves as the template from which to spawn specific instances of that class.

These specific instances of a class are referred to as "objects". Every object has certain characteristics, or "properties", and certain pre−defined functions, or "methods". These properties and methods of the object correspond directly with the variables and functions within the class definition.

Once a class has been defined, Python allows you to spawn as many instances of the class as you like. Each of these instances is a completely independent object, with its own properties and methods, and can thus be manipulated independently of other objects. This comes in handy in situations where you need to spawn more than one instance of an object − for example, two simultaneous database links for two simultaneous queries, or two shopping carts.

Classes also help you keep your code modular − you can define a class in a separate file, and include that file only in the pages where you plan to use the class − and simplify code changes, since you only need to edit a single file to add new functionality to all your spawned objects.

A Very Big Snake

A class definition typically looks like this:

```
class veryBigSnake:

# some useful methods
def eat(self):
# code goes here

def sleep(self):
# code goes here

def squeeze_to_death(self):
# code goes here
```

Once the class has been defined, you can instantiate a new object by assigning it to a Python variable,

```
Python 1.5.2 (#1, Feb 1 2000, 16:32:16) [GCC egcs−2.91.66
19990314/Linux
(egcs− on linux−i386
Copyright 1991−1995 Stichting Mathematisch Centrum, Amsterdam
>>> python = veryBigSnake()
>>>
```

which can then be used to access all object methods and properties.

```
>>> python.eat()
>>>
>>> python.sleep()
>>>
```

As stated above, each instance of a class is independent of the others − which means that, if I was a snake fancier, I could create more than one instance of veryBigSnake(), and call the methods of each instance independently.

```
>>> alpha = veryBigSnake()
>>> beta = veryBigSnake()
>>> # make the alpha snake eat
>>> alpha.eat()
>>>
```

```
>>> # make the beta snake eat and sleep
>>> beta.eat()
>>>
>>> beta.sleep()
>>>
```

What's In A Name?

Let's now add a few properties to the mix, by modifying the class definition to support some additional characteristics:

```
class veryBigSnake:

# some useful methods
def eat(self):
# code goes here

def sleep(self):
# code goes here

def squeeze_to_death(self):
# code goes here

def set_snake_name(self, name):
# code goes here
self.name = name
```

The class definition now contains one additional method, set_snake_name(), which modifies the value of the property "name". Let's see how this works:

```
>>> alpha = veryBigSnake()
>>> beta = veryBigSnake()
>>>
>>> # name the alpha snake
>>> alpha.set_snake_name("Peter Python")
>>> alpha.name
'Peter Python'
>>>
>>> # name the beta snake
>>> beta.set_snake_name("Bobby Boa")
>>> beta.name
'Bobby Boa'
>>>
>>> # rename the alpha snake
>>> alpha.set_snake_name("Sally Snake")
>>> alpha.name
'Sally Snake'
>>>
```

As the illustration above shows, once new objects are defined, their individual methods and properties can be accessed and modified independent of each other. This comes in very handy, as the next few pages will show.

It's also important to note the manner in which object methods and properties are accessed − by prefixing the method or property name with the name of the specific object instance.

Digging Deep

Now that you've got the concepts straight, let's take a look at the nitty−gritty of a class definition.

```
class veryBigSnake:

# method and property definitions
```

Every class definition begins with the keyword "class", followed by a class name. You can give your class any name that strikes your fancy, so long as it doesn't collide with a reserved word. All class variables and functions are indented within this block, and are written as you would normally code them.

In order to create a new instance of a class, you need to simply create a new variable referencing the class.

```
>>> alpha = veryBigSnake()
>>>
```

In English, the above would mean "create a new object of class veryBigSnake and assign it to the variable 'alpha'".

You can now access all the methods and properties of the class via this variable.

```
>>> # accessing a method
>>> alpha.set_snake_name("Peter Python")
>>>
>>> # accessing a property
>>> alpha.name
>>>
```

Again, in English,

```
>>> alpha.set_snake_name("Peter Python")
>>>
```

would mean

"execute the method set_snake_name() with parameter 'Peter Python' of this specific instance of the class veryBigSnake".

Self−Involved

You'll have noticed, in the examples above, a curious little thingummy called "self". And you're probably wondering what exactly it has to do with anything.

The "self" variable has a lot to do with the way class methods work in Python. When a class method is called, it requires to be passed a reference to the instance that is calling it. This reference is passed as the first argument to the method, and is represented by the "self" variable.

```
class veryBigSnake:

# some useful methods

def set_snake_name(self, name):
self.name = name
```

An example of this can be seen in the set_snake_name() function above. When an instance of the class calls this method, a reference to the instance is automatically passed to the method, together with the additional "name" parameter. This reference is then used to update the "name" variable belonging to that specific instance of the class.

Here's an example of how this works:

```
>>> alpha = veryBigSnake()
>>> alpha.set_snake_name("Peter Python")
>>> alpha.name
'Peter Python'
>>>
```

Note that when you call a class method from an instance, you do not need to explicitly pass this reference to the method − Python takes care of this for you automatically.

With that in mind, consider this:

```
>>> alpha = test.veryBigSnake()
>>> veryBigSnake.set_snake_name(alpha, "Peter Python")
>>> alpha.name
'Peter Python'
>>>
```

In this case, I'm calling the class method directly (not via an instance), but passing it a reference to the instance in the method call − which makes this snippet equivalent to the one above it.

Under Construction

It's also possible to automatically execute a function when the class is called to create a new object. This is referred to in geek lingo as a "constructor" and, in order to use it, your class definition must contain a method named __init()__

Typically, the __init__() constructor is used to initialize variables or execute class methods when the object is first created. With that in mind, let's make a few changes to the veryBigSnake class:

```
class veryBigSnake:

# constructor
def __init__(self):
# initialize properties
self.name = "Peter Python"
self.type = "python"
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!"
```

In this case, the constructor sets up a couple of variables with default values and prints a short message indicating that a new object has been successfully created.

The set_snake_type() and set_snake_name() functions alter the object's properties, while the who_am_i() function provides a handy way to see the current values of the object's properties.

Here's an example of how it could be used:

```
>>> alpha = veryBigSnake()
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("Alan Adder")
>>> alpha.set_snake_type("harmless green adder")
>>> alpha.who_am_i()
My name is Alan Adder, I'm a harmless green adder and I'm
perfect for you!
Take
me home today!
>>>
```

Since class methods work in exactly the same way as regular Python functions, you can set up the constructor to accept arguments, and also specify default values for these arguments. This makes it possible to simplify the class definition above to:

```
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!"
```

And here's how you could use it:

```
>>> # create two snakes
>>> alpha = veryBigSnake("Roger Rattler", "rattlesnake")
New snake in da house!
>>> beta = veryBigSnake()
```

```
New snake in da house!
>>>
>>> # view snake information
>>> alpha.who_am_i()
My name is Roger Rattler, I'm a rattlesnake and I'm perfect
for you! Take
me home today!
>>>
>>> # notice that the beta snake has been created with default
properties!
>>> beta.who_am_i()
My name is Peter Python, I'm a python and I'm perfect for you!
Take me home
today!
>>> alpha.name
'Roger Rattler'
>>> alpha.type
'rattlesnake'
>>> beta.name
'Peter Python'
>>> beta.type
'python'
>>>
```

It should be noted here that it is also possible to directly set instance properties, bypassing the exposed methods of the class. For example, the line

```
>>> beta.set_snake_name("Vanessa Viper")
>>>
```

is technically equivalent to

```
>>> beta.name="Vanessa Viper"
>>>
```

Notice I said "technically". It is generally not advisable to do this, as it would violate the integrity of the object; the preferred technique is always to use the methods exposed by the object to change object properties.

Just as an illustration − consider what would happen if the author of the veryBigSnake() class decided to change the variable "name" to "snake_name".You would need to rework your test script as well, since you were directly referencing the variable "name". If, on the other hand, you had used the exposed set_snake_name() method, the changes to the variable name would be reflected in the set_snake_name() method by the author and your code would require no changes whatsoever.

By limiting yourself to exposed methods, you are provided with a level of protection which ensures that changes in the class code do not have repercussions on your code.

Tick, Tick

Now that you've (hopefully) understood the fundamental principles here, let's move on to something slightly more practical. This next package allows you to create different Clock objects, and initialize each one to a specific time zone. You could create a clock which displays local time, another which displays the time in New York, and a third which displays the time in London − all through the power of Python objects. Let's take a look:

```
# 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!
```

As you can see, the class's constructor initializes an object with four variables: the name of the city, an indicator as to whether the city time is ahead of, or behind, Greenwich Mean Time, and the difference between the local time in that city and the standard GMT, in hours and minutes.

Once these attributes have been set, the display() method takes over and performs some simple calculations to obtain the local time in that city.

And here's how you could use it:

```
>>> london = Clock("+", 0, 00, "London")
Clock created
>>> london.display()
Local time in London is 8:52:21
>>> bombay = Clock("+", 5, 30, "Bombay")
```

```
Clock created
>>>
>>> bombay.display()
Local time in Bombay is 14:23:5
>>> us_ct = Clock("-", 6, 00, "US/Central")
Clock created
>>> us_ct.display()
Local time in US/Central is 2:53:11
>>>
```

And that's just about it for the moment. You now know the basics of Python's object−oriented programming paradigm, and should be able to construct and use your own objects. While you're doing that, remember that I'll be back next week with some more dope on the topic − the second part of this article will cover inheritance, overrides, destructors and the built−in Python functions that help you navigate between classes and class instances. Don't miss it!

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!