

Object Oriented programming in Python

This is a short tutorial about object oriented programming in Python. It will show some basic features and the most important things to know about object in Python. Enjoy!!!

When it comes to object-oriented programming, very few languages have Python's capabilities.

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.

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 class definition typically looks like this:
(save this file to e.g. Snake.py)

```
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,

Instead of using the interactive interpreter, save the code into e.g. class_try.py and use

`$python class_try.py`

To try everything out.

Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam

```
>>> import Snake ##needed to get access to the classes
>>> 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()
>>>
```

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. 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". 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. 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. {mospagebreak title=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
>>>
```

Inheritance

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
>>>
```

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? 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>
>>>
```

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!  
>>>
```

More info can be found at:

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>