Taming Python

Python's flexibility lets you change the rules when necessary

April 05, 2010
URL:http://www.drdobbs.com/tools/taming-python/224201320

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems. He is currently trying to build intelligent machines inspired by the brain at Numenta (www.numenta.com).

Python is a great dynamic language. In what way? For one thing, it allows you to add attributes to objects anytime. Consider the following class **A**:

```
class A(object):
  def __init__(self):
    self.x = 5

  def set_y(self, value):
    self.y = value
```

When you instantiate it as in:

```
a = A()
```

The **__init__()** is executed and the **x** attribute is set to 5. However, it has no **y** attribute just yet:

```
print 'a.x:', a.x
```

So try:

```
  print 'a.y:', a.y
except AttributeError:
  print "a has no 'y' attribute"
```

Output:

```
a.x: 5
a.y: a has no 'y' attribute
```

Now, if you call the **set_y()** method suddenly the **a** object grows a **'y'** attribute. You can even add a **'z'** attribute willy nilly from outside the class:

```
a.set_y(8)
print 'a.y:', a.y

a.z = 12
print 'a.z:', a.z
```

Output:

```
a.y: 8
a.z: 12
```

What's going on here? It's actually pretty simple. Each Python object keeps all its attributes in a special dictionary (a collection of key-value pairs) called **__dict__** and anyone can access the **__dict__** of any object and add/delete/modify items. Here is the **__dict__** after **y** and **z** were added:

```
print a.__dict__
{'y': 8, 'x': 5, 'z': 12}
```

Note that **__dict__** is not ordered by the insertion order. Python lets you remove attributes using the **del** statement and check for the existence of attributes using the **hasattr()** function. Here I verify that **a** has a **'z'** attribute, remove it, and verify it doesn't have a **'z'** attribute anymore:

```
>>> assert hasattr(a, 'z')
>>> del a.z
>>> assert not hasattr(a, 'z')
```

But, you can delete it directly by working with the **__dict__**. Here I check that **'y'** is in the **__dict__**, remove it from the **__dict__**, and check it's not in the **__dict__**.

```
>>> assert 'y' in a.__dict__
>>> del a.__dict__['y']
>>> assert 'y' not in a.__dict__
```

It looks like **hasattr()** is not really needed, but that's not the case. Python objects have many attributes that are not in their **__dict__**. For example, the **__dict__** itself is not in the **__dict__**. Python objects get many attributes from their types. For example, the **__init__()** and **set_y()** methods are also attributes that the **'a'** object gets from its class **A**:

```
>>> a.set_y
<bound method A.set_y of <code.A object at 0x411730>>

>>> hasattr(a, 'set_y')
True

>>> 'set_y' in a.__dict__
False
```

The bottom line is that it's good to know about the **__dict__**, but you should use **hasattr()** if you want to test for the exsitence of an attribute. If you want to see all the attributes try the **'dir'** function:

```
>>> dir(a)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__',
'__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'set_y', 'x']
```

In some situations this enormous flexibility becomes a burden. At Numenta where I work, we use Python objects as plugins to a runtime engine (server) that communicates with a client that may potentially be on another machine. This means that the communication goes through a special protocol and operations on the plugins such as initialization, execution and persistence are orchestrated carefully. The runtime engine itself is implemented in C++ and interacts with the Python object indirectly through a C++ wrapper. A common scenario is that the Python object creates new attributes during its execution.

But this also caused a lot of confusion and subtle bugs. Some attributes weren't available at the right time. Typos couldn't be detected easily (setting an attribute with the wrong name just created a new attribute). To make things even worse, a few generations of changes led to various attributes becoming obsolete and/or deprecated. These attributes were created on-the-fly when loading an old object from a file.

This aspect of Python became a real obstacle. The solution that emerged was that new attributes should be created only in the **__init__()** and **__setstate__()** methods. The **__init()__** method is the method that's being called when a new object is instantiated for the first time. The **__setstate__()** method is called when an object is loaded from a "pickle" file. A pickle file is simply a built-in serialization scheme that Python supports. That allows persisting Python objects across multiple sessions of the application.

Okay, so this policy means that methods like **set_y**() are invalid (unless called directly or indirectly from **__init__**() or **__setstate__**()) because they create a new attribute. Also external code that tries to create new attributes like **'a.z = 12'** is forbidden. If you have an attribute that you don't know what to set it to at **__init__**() time, just set it to **'None'**.

Well, that sounds great, but just telling programmers about this magnificent new policy is not enough. Programmers are notorious for being a confused and/or belligerent bunch and even the most obedient programers can't escape typos. The Numenta software has many Python plugins, which are pretty big objects (some has 20-30 attributes) and it's not an easy task even to detect violations of policy, not to mention fixing them. What is needed is an automated approach that can enforce the policy without excpetion. This is were the lock attributes pattern comes into play. It is a pretty sophisticated solution that involves intercepting attribute setting and checking for violations. When a violation is detected an exception is raised.

Implementing, Deploying, Managing

Let's break it down to digestable pieces and see how to implement, deploy, and manage it:

1. Intercept somehow every attribute setting on every object of interest.
2. Determine if the attribute is being set by code in **__init__()/__setstate__**() or called indirectly from **__init__()/__setstate__**()
3. If the policy is violated raise an exception
4. Find an easy way to attach the policy checking code to every target object
5. Performance. All these checks can take a serious toll. The user should be able to turn it off.

Intercept Every Attribute Setting. Python provides the answer through the **__setattr__**() magic method. This method is defined on the base object and performs the assignment operation for all objects. If you override it in your class then every attribute assignment goes to this method. Here is a simple example: Class **X** defines a **__setattr__** method that just prints a message when an attribute is set on an instance of **X**. This happens when a method of **X** itself like **__init__**() is trying to set an attribute called **'a'** and also when external code is trying to set a **'b'** attribute on an instance of **X**:

```
class X(object):
  def __init__(self):
    self.a = 3
  def __setattr__(self, name, value):
    print 'Trying to set', name, 'to', value

x = X()
x.b = 5

assert not hasattr(x, 'a')
assert not hasattr(x, 'b')
```

Output:

```
Trying to set a to 3
Trying to set b to 5
```

If you actually want the attribute to be set, you need to call **object.__setattr__(self, nsme, value)**.

Determine the Origin of set Attribute Attempts. The goal is to allow setting attributes only in code originating from **__init__**() or **__setstate__**(). You can do it in several ways. For example, you can examine the call stack and verify that it contains **__init__**()/**__setstate__**(). But, this is complicated, expansive, and potentially brittle (e.g., you can set an attribute of object **x** from the **__init__**() of object **y**, which will violate the rules). A better way is to turn on a flag at the beginning of **__init__**()/**__setstate__**() and turn it off at the end. The special flag is going to be just an attribute called **_can_add_attributes**. The **__setattr__**() method will check if the object has that attribute and will allow adding new attributes only if the attribute exists.

Raise an Exception If the Policy Is Violated. That's pretty easy. Whenever a new attribute is set, the **__setattr__** will check if the object already has the attribute. If not, it will check if **_can_add_attributes** is present; if not, it means the policy has been violated and just raise an exception.

Find an Easy Way to Attach the Policy Checking Code to Every Target Object. This isn't easy. The idea is to do it as unobstrusively as possile. If you require people to add a special code in every method they will rebel, forget or more like both. The trick is to modify their classes dynamically. In Python 2.6 and up class decorators are the ticket. In Python 2.2 though 2.5, meta-class will get you there. I will show both techniques. Class decorators are easy to apply. meta-classes can be applied just by deriving from a base class or even better -- a mixin.

Make Sure Performance Is Not Hindered By All This Uber-sophisticated Mechanism. Python is slow. That's not new. There are various ways to speed it up. The best way is always to make it do less. You may consider running all your tests with the the lock attribute checking turned on and then in production turn if off. In Numenta, I use an environment variable to control the lock attribues mechanism. If the environment variable NTA_DONT_USE_LOCK_ATTRIBUTES is defined then the lock attributes mechanism is not engaged and you don't pay for it (other than the one-time check if the environment variable is defined). This way, by default all developers can be confident that they don't violate the lock attributes principle by accident, but in the production environment the environment variable is defined and there is no overhead.

Demo Time

Here is a little class **A** that that uses the lock attributes mechanism. It simply subclsses a mysterious class called **LockAttributesMixin** imported from the **lockattributes** module (which I'll talk about soon):

```
from lockattributes import LockAttributesMixin

class A(LockAttributesMixin):
  def __init__(self):
    self.x = 777

  def foo(self):
    self.y = 888
```

The **A** class defines an **'x'** attribute in its **__init__**() method, which is fine and a **'y'** attribute in its **foo()** method, which violates the lock attributes principle. Let's see what happens if you try to invoke **foo()**.

```
a = A()

print 'a.x = ', a.x

try:
  a.foo()
except Exception, e:
  print 'a.foo() failed:', str(e)
```

Output:

```
a.x =  777
a.foo() failed: Attempting to set a new attribute: y
```

The result of invoking **a.foo()** was an exception with a very informative mesage that says exactly what went wrong (Attempting to set a new attribute: **y**).

If you try to set a new attribute on a from the outside you will get the same result, but setting **'x'** that was defined in **__init__**() is okay:

```
try:
  a.z = 999
except Exception, e:
  print 'a.z = 999 failed:', str(e)

print 'Setting a.x to 444'
a.x = 444
print 'a.x = ', a.x
```

Output:

```
a.z = 999 failed: Attempting to set a new attribute: z
Setting a.x to 444
a.x =  444
```

This is exactly how lock attributed should behave -- applied easilly just by subclassing some base class, detect attempts to set new attributes outside of **__init__()/__setstate__()**, and raise informative exception when a violation occurs.

Let's see how it works. All the magic is implemented in a single file -- lockattributes.py. This file and a test program that uses it are available for download here. The lockattributes.py module can be used as is (no external dependencies) and the lockattributes_test.py that contains the last example with the **Circle** class (described later in this article).

The _allow_new_attributes() Decorator

Decorators are a great Python feature. They let you replace or augment a function or a method with a different function or method (and much more). Very often decorators are used to run some code before and after calling the original method (think "aspect-oriented programming"). Here is a trivial example that just prints **'before'** and **'after'**. The **decorator** function called **before_after()** takes the original function **f** as argument and returns a new (nested) function called internally **'decorated'** that prints before/after calling the original function **f**. This decorated function will quietly replace the original function:

```
def before_after(f):
  def decorated(*args, **kwargs):
    print 'before'
    f(*args, **kwargs)
    print 'after'
  return decorated
```

You apply a decorator to a function/method by writing **@<decorator name>** before the definition. Here I apply the **before_after** decorator to two simple functions **a()** and **b()** that print **'a'** and **'b'**, respectively:

```
@before_after
def a():
  print 'a'

@before_after
def b():
  print 'b'
```

Calling **a()** and **b()** results in the following output:

```
a()
print
b()
```

In practice, the @ decorator syntax is equivalent to applying the decorator function directly:

```
def c():
  print 'c'

c = before_after(c)
c()
```

Output:

```
before
c
after
```

But, it's nicer to use the decorator syntax and you don't need to repeat the function name. If you want to know more about decorators, I recommend Python 2.4 Decorators.

The lockattributes module defines such a decorator called **_allow_new_attributes()**. This decorator will be used to decorate the **__init__()** and **__setstate__()** methods of target classes to set a special attribute called **_CanAddAttributes**.

```
def _allow_new_attributes(f):
  """A decorator that maintains the attribute lock state of an object
```

It coperates with the **LockAttributesMetaclass** (see below) that replaces the **__setattr__** method with a custom one that checks the **_canAddAttributes** counter and allows setting new attributes only if **_canAddAttributes > 0**. New attributes can be set only from methods decorated with this decorator (should be only **__init__** and **__setstate__** normally). The decorator is reentrant (e.g. if from inside a decorated function another decorated function is invoked). Before invoking the target function it increments the counter (or sets it to 1). After invoking the target function it decrements the counter. When the counter reaches 0, it is removed.

```
  """
  def decorated(self, *args, **kw):
    """The decorated function that replaces __init__() or __setstate__()
    """
    if not hasattr(self, '_canAddAttributes'):
      self._canAddAttributes = 1
    else:
      self._canAddAttributes += 1
    assert self._canAddAttributes >= 1

    # Save add attribute counter
    count = self._canAddAttributes

    # Run the original function
    f(self, *args, **kw)

    # Restore _canAddAttributes if deleted from dict (can happen in __setstate__)
    if hasattr(self, '_canAddAttributes'):
      self._canAddAttributes -= 1
    else:
      self._canAddAttributes = count - 1

    assert self._canAddAttributes >= 0
    if self._canAddAttributes == 0:
      del self._canAddAttributes

  return decorated
```

That's a lot of non-trivial code so analyze it piece by piece. The general structure of the **_allow_new_attributes()** decorator is just like the before_**after()**: It takes the original function **f** as an argument and returns another function called **decorated** that does some stuff before calling the original function f and then does some stuff after calling. The only difference is what kind of stuff happens before and after. The main idea is to have the **'_canAddAttributes'** attribute available on the target object (accessed through **'self'**) during the time **__init__()** and **__setstate__()** are called and not exist after they are finished. The reason **_canAddAttribute** is an integer that is incremented and decremented instead of a simple True/False boolean is that there could be nested calls to base classes. For example, consider the following two classes:

```
from lockattributes import LockAttributeMixin

class A(LockAttributesMixin):
  def __init__(self):
    ...

class B(A):
  def __init__(self):
    A.__init__(self)
    self.x = 5

b = B()
```

When a new **b** object is created the following calls are executed:

```
B.__init__() (because a new B object is created)
  A.__init__() (because B.__init__() calls A.__init__())
  self.x = 5
```

Both **A** and **B** are monitored for locked attributes. Class **A** direcly subclasses the **LockAttributesMixin** and class **B** indirectrly subclasses it by sublassing **A**. Now, suppose **_canAddAttributes** was just a boolean attributes. Before **B.__init__()** is executed will be set to **True. B.__init__()** will call **A.__init__()** and before **a.__init__()** starts **_canAddAttributes** will be set to True again. **A.__init__()** will execute and when it is done **_canAddAttribute** will be set to False! But, now the statement **self.x = 5** (comes after the call to **A.__init__()** inside **B.__init__()**) will fail. The solution is to maintain a count and increase/decrease it appropriately. As long as the count is greater than 0, it is okay to add new attributes.

Python Meta-classes

This is going to be confusing. A meta-class is the class of a class. In Python every object has a class and you can find it out by querying its **__class__** attribute:

```
>>> x = 5
>>> x.__class__
<type 'int'>

>>> a = A()
>>> a.__class__
<class '__main__.A'>
```

Now, classes are objects too. Yes, in Python classes are also objects. If they are objects, then they must have a class. Indeed, they have. The class of every class is by default **'type'**:

```
>>> int.__class__
<type 'type'>
>>> int.__class__
<type 'type'>
>>> A.__class__
<type 'type'>
>>> type.__class__
<type 'type'>
```

Yes, **'type'** is also a class (and an object) and it is also its own class. Is that cool or what? Before you try to figure out all the relationships between objects, classes and meta-classes in Python and rip the fabric of the universe let's move on to class instantiation. Normally, this is done implicitly when the interpreter sees a class definition in a module for the first time. The **'type'** class is able to instantiate any class. Python allows you to attach a different meta-class to a class. That buys you the capability to modify almost anything about the target class. For example, you can add new methods to a class or decorate an existing method. The meta-class called **M** adds a new method called **hello()** that prints 'hello' and in addition it decorates every method in the class with the **before_after()** decorator. All the work is done in the **__init__()** method of the meta-class **M**. The method accepts as first argument the class instance (called **'cls'** instead of the conventional **'self'** to remind you that the instances of the meta-class are classes), then the name of the target class, its bases and finally and most importantly a dictionary of all its attributes, which include its methods. To add or modify functions you can either add them directly (see the **'hello'** method) or use **setattr()**. Don't try to modify the dict directly because it doesn't modify the class itself. The **M** meta-class checks the type of each attribute in the dict and if it is a function (**types.FunctionType**) then it decorates with with the **before_after** decorator:

```
import types

def hello(self):
  print 'hello'

class M(type):
  def __init__(cls, name, bases, d):
    cls.hello = hello
    for name, value in d.items():
      if isinstance(value, types.FunctionType):
        setattr(cls, name, before_after(value))
```

Now, that we have a meta-class let's attach it to some class. This is done by setting the **__meta-class__** attribute of the target class or by subclassing a class that has an attached meta-class. All subclasses of a class with attached meta-class are modified by the meta-class.

```
class X(object):
  __meta-class__ = M

  def foo(self):
    print 'foo'

  def bar(self):
    print 'bar'
```

Instances of **x** gain an additional **'hello()'** method and all their original methods are decorated with **before_after**:

```
x = X()
x.foo()
print '-' * 10
x.bar()
print '-' * 10
x.hello()
```

Output:

```
before
foo
after
----------
before
bar
after
----------
hello
```

That's how meta-classes work their magic and can completely modify the behavior of their target classes. Now, let's look at the actual **LockAttributesMetaclass**

The LockAttributesMetaclass

This meta-class has three goals:

- If the lock attributes machinery is deactivated just bail out early
- Check each attribute setting to verify that the lock atributes principle is not violated.
- Decorate the **__init__()** and **__setstate__()** methods with the **_allow_new_attributes** decorator

It accomplishes the first goal simply by checking if a special environment variable, the **deactivation_key** (DONT_USE_LOCK_ATTRIBUTES) is in the environment:

```
class LockAttributesMetaclass(type):
  def __init__(cls, name, bases, dict):

    # Bail out if not active. Zero overhead other than this one-time check
    # at class definition time
    if deactivation_key in os.environ:
      return
```

It accomplishes the second goal by setting the class **__setattr__()** method to a custom one that checks for lock attributes violations (described later) and also keeping the original **__setattr__ in** an attribute **called _original_setattr** (that will be used later):

```
    # Initialize the super-class
    type.__init__(cls, name, bases, dict)

    # Store and replace the __setattr__ with the custom one (if needed)
    if not hasattr(cls, '_original_setattr'):
      cls._original_setattr = cls.__setattr__
      cls.__setattr__ = custom_setattr
```

It accomplishes the third goal by looking up the **__init__()** and **__setstate__()** methods and decorating them with the **_allow_new_attributes** decorator. There is a subtle point here, which is that the target class might not have **__init__()** or **__setstate__()** methods. Well, if there is no **__setstate__()** it's okay because even if an instance is loaded from a pickle the default **__setstate__** will not add any new attributes. But, if there is **no __init__()** method it can be a problem. The reason is that the lock attributes checking supports inheritance. If a subclass of an attribute locked class is trying to create new attributes in its **__init__()** method it should be able to do it. So, if there is no **__init__()** method a trivial **__init__()** method is added (described later):

```
    # Get the __init__ and __setstate__ from the target class's dict
    # If there is no __init__ use _simple_init (it's Ok if there is no
    #__setstate__)
    methods = [('__init__', dict.get('__init__', _simple_init)),
               ('__setstate__', dict.get('__setstate__', None))]

    # Wrap the methods with _allow_new_attributes decorator
    for name, method in methods:
      if method is not None:
        setattr(cls, name, _allow_new_attributes(method))
```

Here is the **_simple_init()** function:

```
def _simple_init(self, *args, **kw):
  """Trivial init method that just calls the base class' __init__()

  This method is attached to classes that don't define __init__(). It is needed
  because LockAttributesMetaclass must decorate the __init__() method of
  its target class.
  """
  type(self).__base__.__init__(self, *args, **kw)
```

The custom_setattr Function

This function allows setting only existing attributes. It is designed to work with the **_allow_new_attributes** decorator. It works is by checking if the requested attribute is already in the **__dict__** or if the **_canAddAttributes counter > 0**; otherwise it raises an exception.

If all is well it calls the original **__setattr__()**. This means it can work also with classes that already have **custom __setattr__**:

```
    def custom_setattr(self, name, value):
      if (name == '_canAddAttributes' or
         (hasattr(self, '_canAddAttributes') and self._canAddAttributes > 0) or
         hasattr(self, name)):
        return self._original_setattr(name, value)
      else:
        raise Exception('Attempting to set a new attribute: ' + name)
```

The LockAttributeMixin

The **LockAttributesMixin** just attaches the **LockAttributesMetaclass** to the target class. The term "mixin" means a class that is intended to be used as a base class and provide some functionality to classes that derive from it, but it is not useful by itself, so it should never be instantiated directly. The mixin often cooperates with the derived class or other mixin classes:

```
class LockAttributesMixin(object):
  """This class serves as a base (or mixin) for classes that want to enforce
  the locked attributes pattern (all attributes should be defined in __init__()
  or __setstate__().

  All the target class has to do add LockAttributesMixin as one of its bases
  (inherit from it).

  The meta-class will be activated when the application class is created
  and the lock attributes machinery will be injected (unless the
  deactivation_key is defined in the environment)
  """
  __meta-class__ = LockAttributesMetaclass
```

Complete Example with Inheritance and Persistance

The following example demonstrates the lock attributes mechanism in the context of a class hierarchy where both the base class and the derived class implements **__getstate__()/__setstate__()** for persistence into pickle files. The pickle module allows storing and loading Python objects in files. When the object is saved using **pickle.dump()**, the **__getstate__()** method is called and its return value is stored in the file. When the object is loaded using

pickle.load(), the **__setstate__**() method is called with the same state that **__getstate__**() returned and the object can initialize itself from this state.

Here is the base class **Shape** with the required import statements. Note that the state the **Shape** class manages is simply the center point. It subclasses the **LockAttributesMixin**, which means this class and all its subclasses will automatically benefit from locked attributes enforcement.

```
from lockattributes import LockAttributesMixin
import math
import cPickle as pickle

class Shape(LockAttributesMixin):
  def __init__(self, center):
    self.center = center

  def __getstate__(self):
    return self.center

  def __setstate__(self, state):
    self.center = state
```

The **Circle** class subclasses the **Shape** class and has two more attributes --**radius** and **area** -- that are set in **__init__**() and **__setstate__**(). Note that the **area** is NOT stored in the pickle file to save space because it can be calculated from the **radius**. The **Circle** subclass plays nicely with its base class and makes sure to call its **__init__**(), , and **__setstate__**() at the right time. The state the **Circle** manages is a tuple that contains the state of the **Shape** class as first member and the **radius** as the second member. This is a pretty robust scheme because the base class may add or change its persistent state and the subclass persistence code will not need to change:

```
class Circle(Shape):
  def __init__(self, center, radius):
    Shape.__init__(self, center)
    self.radius = radius
    self.calculateArea()

  def calculateArea(self):
    self.area = math.pi * math.pi * self.radius

  def calculatePerimeter(self):
    self.perimeter = 2 * math.pi * self.radius

  def __getstate__(self):
    return (Shape.__getstate__(self), self.radius)

  def __setstate__(self, state):
    Shape.__setstate__(self, state[0])
    self.radius = state[1]
    self.calculateArea()
```

Here is some code that creates a **Circle** object, displays its attributes and then calls the forbidden **calculatePerimeter()** method that attempts to create a new **'perimeter'** attribute:

```
# Create a new circle
x = Circle((3, 4), 5)

# Display its attributes
print 'center:', x.center
print 'radius:', x.radius
print 'area:', x.area
# Call the calculatePerimeter method that creates a new attribute
try:
  x.calculatePerimeter()
except Exception as e:
  print str(e)
```

The next snippet is similar except that the **x** object is stored in a pickle file and then loaded into a new instance **x2**:

```
# Store the circle in a pickle file
with open('circle.pkl', 'w') as f:
  pickle.dump(x, f)

# Load the circle from the .pkl file
with open('circle.pkl') as f:
  x2 = pickle.load(f)
# Display its attributes
print 'center:', x2.center
print 'radius:', x2.radius
print 'area:', x2.area
# Call the calculatePerimeter method that creates a new attribute
try:
  x2.calculatePerimeter()
except Exception as e:
  print str(e)
```

Conclusion

In this article, I have demonstrated the amazing flexibility of Python and the tools it provides to developers to alter its own rules when necessary. As you know power corrupts and ultimate power corrupts ultimately. Don't let the power Python puts in your hands corrupt your code. Used judiciously, you can do great things with Python. Used indiscriminately, you end up with a bad case of obfuscated spaggheti code and collegues hiding under their desks when they hear your footfalls.