The History of Python

A series of articles on the history of the Python programming language and its community.

Monday, June 21, 2010

The Inside Story on New-Style Classes

[Warning, this post is long and gets very technical.]

On the surface, new-style classes appear very similar to the original class implementation. However, new-style classes also introduced a number of new concepts:

- low-level constructors named __new__()
- descriptors, a generalized way to customize attribute access
- static methods and class methods
- properties (computed attributes)
- decorators (introduced in Python 2.4)
- slots
- a new Method Resolution Order (MRO)

In the next few sections, I will try to shine some light on these concepts.

Low-level constructors and __new__()

Traditionally, classes defined an __init__() method which defined how new instances are initialized after their creation. However, in some cases, the author of a class may want to customize how instances are created—for example, if an object was being restored from a persistent database. Old-style classes never really provided a hook for customizing the creation of objects although there were library modules allowed certain kinds of objects to be created in non-standard ways (e.g., the "new" module).

New-style classes introduced a new class method __new__() that lets the class author customize how new class instances are created. By overriding __new__() a class author can implement patterns like the Singleton Pattern, return a previously created instance (e.g., from a free list), or to return an instance of a different class (e.g., a subclass). However, the use of __new__ has other important applications. For example, in the pickle module, __new__ is used to create instances when unserializing objects. In this case, instances are created, but the __init__ method is not invoked.

Another use of __new__ is to help with the subclassing of immutable types. By the nature of their immutability, these kinds of objects can not be initialized through a standard __init__() method. Instead, any kind of special initialization must be performed as the object is created; for instance, if the class wanted to modify the

About Me

Guido van Rossum

Python's BDFL

View my complete profile

value being stored in the immutable object, the __new__ method can do this by passing the modified value to the base class __new__ method.

Descriptors

Descriptors are a generalization of the concept of bound methods, which was central to the implementation of classic classes. In classic classes, when an instance attribute is not found in the instance dictionary, the search continues with the class dictionary, then the dictionaries of its base classes, and so on recursively. When the attribute is found in a class dictionary (as opposed to in the instance dictionary), the interpreter checks if the object found is a Python function object. If so, the returned value is not the object found, but a wrapper object that acts as a currying function. When the wrapper is called, it calls the original function object after inserting the instance in front of the argument list.

For example, consider an instance x of a class C. Now, suppose that one makes a method call x.f(0). This operation looks up the attribute named "f" on x and calls it with an argument of 0. If "f" corresponds to a method defined in the class, the attribute request returns a wrapper function that behaves approximately like the function in this Python pseudocode:

```
def bound_f(arg):
    return f(x, arg)
```

When the wrapper is called with an argument of 0, it calls "f" with two arguments: x and 0. This is the fundamental mechanism by which methods in classes obtain their "self" argument.

Another way to access the function object f (without wrapping it) is to ask the class C for an attribute named “f”. This kind of search does not return a wrapper but simply returns the function f. In other words, x.f(0) is equivalent to C.f(x, 0). This is a pretty fundamental equivalence in Python.

For classic classes, if the attribute search finds any other kind of object, no wrapper is created and the value found in the class dictionary is returned unchanged. This makes it possible to use class attributes as “default” values for instance variables. For example, in the above example, if class C has an attribute named “a” whose value is 1, and there is no key “a” in x’s instance dictionary, then x.a equals 1. Assignment to x.a will create an key “a” in x’s instance dictionary whose value will shadow the class attribute (by virtue of the attribute search order). Deleting x.a will reveal the shadowed value (1) again.

Unfortunately, some Python developers were discovering the limitation of this scheme. One limitation was that it was prevented the creation of “hybrid” classes that had some methods implemented in Python and others in C, because only Python functions were being wrapped in such a way as to provide the method with access to the instance, and this behavior was hard-coded in the language. There was also no obvious way to define different kinds of methods such as a static member functions familiar to C++ and Java programmers.

To address this issue, Python 2.2 introduced a straightforward generalization of the above wrapping behavior. Instead of hard-coding the behavior that Python function objects are wrapped and other objects aren't, the wrapping is now left up to the object found by the attribute search (the function f in the above example). If the object found by an attribute search happens to have a special method named __get__, it is considered to be a "descriptor" object. The __get__ method is then called and whatever is returned is used to produce the result of the attribute search. If the object has no __get__ method, it is returned unchanged. To obtain the original behavior (wrapping function objects) without special-casing function objects in the instance attribute lookup code, function objects now have a __get__ method that returns a wrapper as before. However, users are free to define other classes with methods named __get__, and their instances, when found in a class dictionary during an instance attribute lookup, can also wrap themselves in any way they like.

In addition to generalizing the concept of attribute lookup, it also made sense to extend this idea for the operations of setting or deleting an attribute. Thus, a similar scheme is used for assignment operations such as x.a = 1 or del x.a. In these cases, if the attribute "a" is found in the instance's class dictionary (not in the instance dictionary), the object stored in the class dictionary is checked to see if it has a __set__ and __delete__ special method respectively. (Remember that __del__ has a completely different meaning already.) Thus, by redefining these methods, a descriptor object can have complete control over what it means to get, set, and delete an attribute. However, it's important emphasize that this customization only applies when a descriptor instance appears in a class dictionary—not the instance dictionary of an object.

staticmethod, classmethod, and property

Python 2.2 added three predefined classes: classmethod, staticmethod, and property, that utilized the new descriptor machinery. classmethod and staticmethod were simply wrappers for function objects, implementing different __get__ methods to return different kinds of wrappers for calling the underlying function. For instance, the staticmethod wrapper calls the function without modifying the argument list at all. The classmethod wrapper calls the function with the instance's class object set as the first argument instead of the instance itself. Both can be called via an instance or via the class and the arguments will be the same.

The property class is a wrapper that turned a pair of methods for getting and setting a value into an "attribute." For example, if you have a class like this,

```
class C(object):
    def set_x(self, value):
        self.__x = value
    def get_x(self):
        return self.__x
```

a property wrapper could be used to make an attribute "x" that when accessed, would implicitly call the get_x and set_x methods.

When first introduced, there was no special syntax for using the classmethod, staticmethod, and property descriptors. At the time,

it was deemed too controversial to simultaneously introduce a major new feature along with new syntax (which always leads to a heated debate). Thus, to use these features, you would define your class and methods normally, but add extra statements that would "wrap" the methods. For example:

```
class C:
    def foo(cls, arg):
        ...
    foo = classmethod(foo)
    def bar(arg):
        ...
    bar = staticmethod(bar)
```

For properties, a similar scheme was used:

```
class C:
 def set_x(self, value):
     self.__x = value
 def get_x(self):
     return self.__x
 x = property(get_x, set_x)
```

Decorators

A downside of this approach is that the reader of a class had to read all the way till the end of a method declaration before finding out whether it was a class or static method (or some user-defined variation). In Python 2.4, new syntax was finally introduced, allowing one to write the following instead:

```
class C:
  @classmethod
  def foo(cls, arg):
     ...
  @staticmethod
  def bar(arg):
     ...
```

The construct @expression, on a line by itself before a function declaration, is called a decorator. (Not to be confused with descriptor, which refers to a wrapper implementing __get__; see above.) The particular choice of decorator syntax (derived from Java's annotations) was debated endlessly before it was decided by "BDFL pronouncement". (David Beazley wrote a piece about the history of the term BDFL that I'll publish separately.)

The decorator feature has become one of the more successful language features, and the use of custom decorators has exceeded my widest expectations. Especially web frameworks have found lots of uses for them. Based on this success, in Python 2.6, the decorator syntax was extended from function definitions to include class definitions.

Slots

Another enhancement made possible with descriptors was the introduction of the __slots__ attribute on classes. For example, a class could be defined like this:

```
class C:
  __slots__ = ['x','y']
```

...

The presence of __slots__ does several things. First, it restricts the valid set of attribute names on an object to exactly those names listed. Second, since the attributes are now fixed, it is no longer necessary to store attributes in an instance dictionary, so the __dict__ attribute is removed (unless a base class already has it; it can also be added back by a subclass that doesn't use __slots__). Instead, the attributes can be stored in predetermined locations within an array. Thus, every slot attribute is actually a descriptor object that knows how to set/get each attribute using an array index. Underneath the covers, the implementation of this feature is done entirely in C and is highly efficient.

Some people mistakenly assume that the intended purpose of __slots__ is to increase code safety (by restricting the attribute names). In reality, my ultimate goal was performance. Not only was __slots__ an interesting application of descriptors, I feared that all of the changes in the class system were going to have a negative impact on performance. In particular, in order to make data descriptors work properly, any manipulation of an object's attributes first involved a check of the class dictionary to see if that attribute was, in fact, a data descriptor. If so, the descriptor was used to handle the attribute access instead of manually manipulating the instance dictionary as is normally the case. However, this extra check also meant that an extra lookup would be performed prior to inspecting the dictionary of each instance. Thus the use of __slots__ was a way to optimize the lookup of data attributes—a fallback, if you will, in case people were disappointed with the performance impact of the new class system. This turned out unnecessary, but by that time it was of course too late to remove __slots__. Of course, used properly, slots really can increase performance, especially by reducing memory footprint when many small objects are created.

I'll leave the history of Python's Method Resolution Order (MRO) to the next post.

Posted by Guido van Rossum at 11:18 AM

---

10 comments:

Rakesh T. June 23, 2010 at 11:19 AM

Wow... That's too much technical for me! I must admit that when I was searching internet for a good programming language, I found Python, and when I tried it, I fell in love with it.

Python is so very easy and fun to learn and code... Python rocks!

Reply

carl June 23, 2010 at 11:31 AM

Great article, thanks!

Indentation error in the first example of pre-decorator function wrapping: foo = classmethod(foo) is indented one level further than it should be.

Reply

Guido van Rossum June 23, 2010 at 2:22 PM

Thanks, indent error fixed (blogger's editor really sucks for code samples).

Reply

andrewbadr June 23, 2010 at 9:40 PM

This is great stuff. Thanks, Guido.

Reply

hcarvalhoalves June 23, 2010 at 9:45 PM

Interesting, now I can see how __slots__ can be a lot more efficient! I never really understood the advantages of it, as I don't consider having fixed class attributes exactly an "advantage".

Reply

Jayesh June 23, 2010 at 10:58 PM

Great article

Reply

itsadok June 23, 2010 at 11:54 PM

I was unclear about how the descriptor's __get__ method would know which object to operate on, but a quick web search revealed the obvious answer - it is passed as a parameter.

Anyway, for those confused like me, here's a python implementation of staticmethod, classmethod and "normalmethod":

```
class normalmethod(object):
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, type_=None):
        def wrapper(*args):
            return self.f(obj, *args)
    return wrapper

class mystaticmethod(object):
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, type_=None):
        return self.f

class classmethod(object):
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, type_=None):
        if type_ is None:
            type_ = type(obj)
        def wrapper(*args):
            return self.f(type_, *args)
        return wrapper
```

Reply

systempuntoout June 28, 2010 at 6:29 AM

great article, for code sample you could try http://www.hilite.me/ hosted on App Engine.

Reply

Aaron Meurer January 23, 2013 at 10:17 PM

Excellent description of descriptors! I made the mistake of trying to understand them from the documentation, which is great as a reference, but not so hot if you just want to understand what the feature is and why it exists.

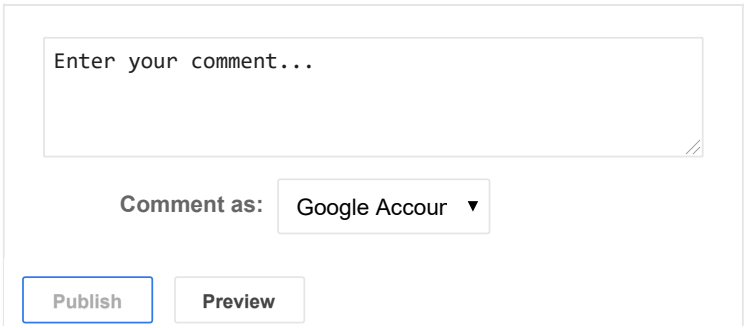
This whole blog has been very well written. Any plans to pick it up again?

Reply

Andrew Pontzen October 13, 2014 at 2:15 AM

Thanks, this cleared up a whole load of confusion I had. However, shouldn't all the example classes be derived from object to make them new-style, or have I misunderstood something there?

Reply



Note: Only a member of this blog may post a comment.

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)