The new class model in Python 2.2

CLOTHES MAKETH THE MAN

The new-style classes introduced with Python 2.2 allow cleaner programming. Andreas Jung investigates how these innovations help the programmer to avoid having to resort to dirty tricks

Just before last Christmas PythonLabs, under the leadership of Guido van Rossum, brought us Python 2.2. As mentioned in a previous article, the class model has changed substantially in Python 2.2, which seems like reason enough to take a detailed look at the changes.

What's new?

Until now there has been a strict separation between built-in data types, such as lists, dictionaries and tuples, and user-defined classes. Some classes could not derive from built-in data types. There were wrapper modules like *UserDict* and *UserList* for emulating built-in data types, but these were more of a work-around.

Python 2.2 introduces something called new-style classes (NS). This type of class is characterised by being derived from *object*.

```
class X(object):
    ...
```

In Python 2.2 the following built-in data types are now also NS classes and therefore derive from *object*:

```
* int
* long
* float
* complex
* str
* unicode
* tuple
* list
* dict
```

They can now also be used as factories; for instance, *d=dict()* is identical to *d={}*.

In Listing 1 we are going to explain the application of NS classes using the implementation of the class *sortedDict*. *sortedDict* is intended to behave like a dictionary, the difference being that the sequence of

Listing 1: Dictionary with sequence

"sortedDict.py"

```
    class sortedDict(dict):

        def __init__(self):
             dict.__init__(self)
             self.lst = list()

        def __setitem__(self,k,v):
             dict.__setitem__(self,k,v)
             if not k in self.lst:
                self.lst.append(k)

        def __delitem__(self,k):
            dict.__delitem__(self,k)
            self.lst.remove(k)

        def keys(self):
            return self.lst

        def values(self):
            return [ dict.__getitem__(self,x) for x in self.lst]

        def items(self):
            return [ (x,dict.__getitem__(self,x)) for x in self.lst]

    if __name__ == "__main__":

        d            = sortedDict()
        d['linux']   = 'magazine'
        d[17]        = 42
        d[ (2,3) ]   = (1,2)

        print 'keys():',d.keys()
        print 'values():',d.values()
        print 'items():',d.items()
```

Listing 2: Extended *sortedDict.py*

```
1  class sortedDict(dict):
2        ..
3
4       def items(self):
5           return [ (x,dict.__getitem__(self,x)) for x in self.lst]
6
7
8       class sortedDictIterator:
9
10          def __init__(self,lst):
11             self.lst = lst
12             self.__num = 0
13
14          def next(self):
15
16             if self.__num < len(self.lst):
17                self.__num+=1
18                return  self.lst[self.__num-1]
19             else:
20                raise StopIteration
21
22      def __iter__(self):
23          return self.sortedDictIterator(self.lst)
24
25
26 if __name__ == "__main__":
27
28      d            = sortedDict()
29      d['linux']   = 'magazine'
30      d[17]        = 42
31      d[ (2,3) ]   = (1,2)
32
33      for key in d:
34          print   'Key=%s, value=%s' % (key, d[key])
```

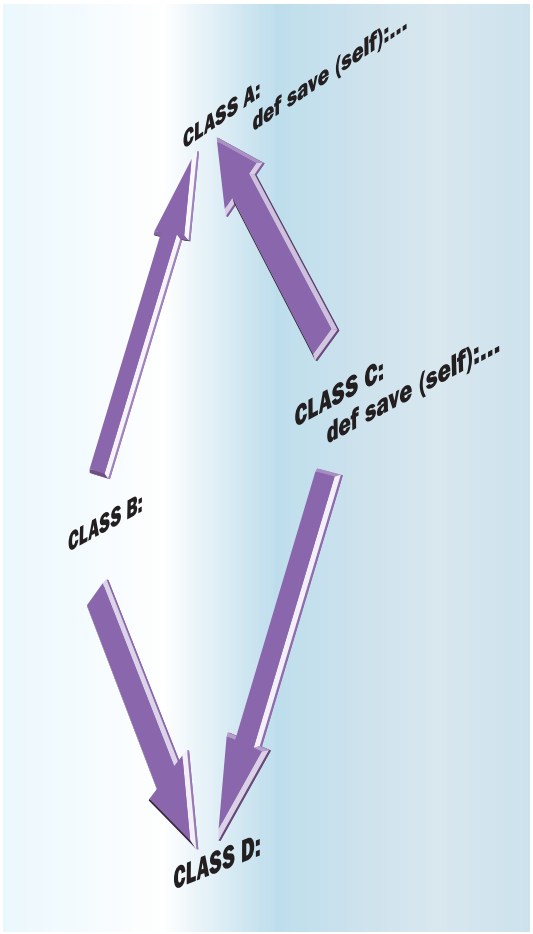


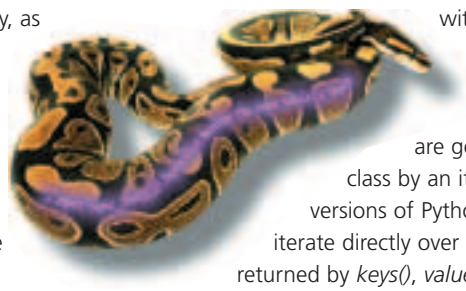
Figure 1: The "diamond rule" illustrates inefficient name resolution

the keys is maintained when reading from the dictionary after new elements are added. Just as a reminder, Python dictionaries do not define a fixed key sequence, that means *keys()* does not necessarily return the keys in the order in which they were added to the dictionary.

The script in Listing 1 provides the following output:

```
keys(): ['linux', 17, (2, 3)]
values(): ['magazine', 42, (1, 2)]
items(): [('linux', 'magazine'),
(17, 42), ((2, 3), (1, 2))]
```

Line 1 defines the new class *sortedDict* as being derived from the built-in dictionary class *dict*. It is not necessary to specify *object* explicitly, as *dict* itself already derives from *object*. The constructor in lines 3 to 5 initialises the dictionary and also defines the internal variable *lst*, which is going to store the keys in sequence. *lst* is needed later to obtain the sequence of the keys when reading from the dictionary. *__setitem__()* and *__delitem()__* are required if new keys are added or deleted. When setitem is used a new key is appended to *lst*, while use of delitem deletes a key. Our example overrides the *keys()* method so that the list of keys is returned. The same is true for the methods *values()* and *items()*.



Iterator extension

Until now you had to use *__getitem__()* in Python classes in order to iterate objects within a *for* loop. This approach can be ambiguous and prone to errors. Instead, Python 2.2 allows you to define *__iter__()* within a class and to return an iterator object. Python invokes *__iter__()* once when entering the *for* loop. Essentially an iterator implements a *next()* method that Python calls at each iteration within the *for* loop and which returns the next element of the object.

To demonstrate this, we are going to extend our *sortedDict* class by an iterator interface. In previous versions of Python it was not possible to iterate directly over a dictionary, only over the list returned by *keys()*, *values()* or *items()*.

For our example we are going to introduce the new class *sortedDictIterator*. *__iter__()* returns an instance of this class as soon as a *for* loop is used to iterate over an instance of *sortedDict*. A reference to the dictionary's keys initialises the constructor of the iterator object. At each iteration of the loop *next()* returns the next element of the keys until the end of the list is reached. The implementation returns the keys similarly to the implementation of the iterator object for dictionaries in Python 2.2. When a *StopIteration* exception is raised this indicates the end of the iteration.

The extended program in Listing 2 provides the following output:

```
Key=linux, value=magazine
Key=17, value=42
Key=(2, 3), value=(1, 2)
```

Multiple inheritance

Up until now name resolution during multiple inheritance has often led to unexpected results. This becomes particularly obvious when looking at the example of the famous diamond rule (Figure 1).

When invoking *save()* on an instance of D, the base classes B and A are searched first, followed only then by C (depth-first rule). That means the old resolution algorithm calls *A.save()*, even though *C.save()* would be more logical. NS classes use a new resolution method, based largely on Common Lisp:

- All base classes are listed according to the depth-first rule, with base classes that are used several times getting multiple listings: [D B A C A]
- Duplicates are removed from the list, leaving only the last occurrence of the element: [D B C A]
- The remaining list is searched from left to right in order to find the method (i.e. in the above example C.save() would be found)

Attribute access

Until now, access to the attributes of an object took place in two stages. First a check whether the required attribute existed in the instance's dictionary

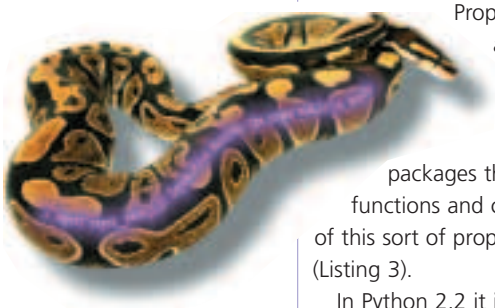
Listing 3: *properties.py*

```
class Test(object):
   def set_number(self,n):
        self.n = n
   def get_number(self):
       return self.n*self.n
    def del_number(self):
       del self.n
   number = property(get_number,set_number,\
                     del_number,"Number")

T = Test()
T.number = 5
print T.number
del T.number
```

__dict__. If it did not, *__getattr__()* was called. This approach is very popular for calculating attributes on the fly (also known as *computed attributes*), but it can easily lead to infinite recursion. For NS classes there is a new *__getattribute(attr)__* method, which is called every time an attribute is accessed.

Properties and slots



Properties are a special type of attribute. They behave like attributes but have their own read, modify and delete functions. The new *property()* function packages the relevant *get*, *set* and *del* functions and creates a property object. The use of this sort of property is externally transparent (Listing 3).

In Python 2.2 it is possible to limit the number of attribute names permitted for an object using the *__slot__* attribute. This makes access to other attributes impossible (Listing 4).

Static methods

Static methods are methods that are not tied to any instance of an object but are instead called directly through the class. In contrast to normal class methods *self* is omitted as the first argument and the method itself is defined as a static method by a *staticmethod()* call.

Listing 5 creates a static method called *foo()*. Calling this through the *Demo* class or one of its instances returns "a b" in both cases.

Listing 4: *slots.py*

```
class Demo(object):
   __slots__ = ['x','y']

Output
>> from slots import Demo
>> D = Demo()
>> D.x = 2
>> D.y = 4
>> D.z = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'Demo' object has no ➥
attribute 'z'
```

Listing 5: static.py

```
class Demo:

   def foo(x,y):
       print x,y

   foo = staticmethod(foo)

Demo.foo('a','b')
demo = Demo()
demo.foo('a','b')
```

Backwards compatible?

Old-style and new-style classes co-exist peacefully in Python 2.2. Any semantic changes relate solely to new-style classes (and these are easily identifiable by being derived from *object*).

Conclusion

The unification of built-in data types and user-defined classes makes programming under Python easier, by avoiding redundant code and through clear structures such as properties. Many important details can't be explained here due to the limitations of space. Guido van Rossum has described all the changes at length at *python.org*; a shorter summary can be found at *amk.ca*. Some of the innovations are hard to understand at times, but use of trial and error and Python's interactive mode should allow you to familiarise yourself with the new concepts quite quickly.

Info:

Python 2.2: *http://www.python.org/2.2*
Guido von Rossum: "Unifying types and classes in Python 2.2":
http://www.python.org/2.2/descrintro.html
AMK: "What's New in Python 2.2":
http://www.amk.ca/python/2.2

The author

Andreas Jung lives near Washington D.C. and works for Zope Corporation in the Zope core team. Email:
andreas@andreas-jung.com