Subject: Re: The metaclass saga using Python
From: Vladimir Marangozov <Vladimir.Marangozov@imag.fr>
To: tim_one@email.msn.com (Tim Peters)
Cc: python-list@cwi.nl
Date: Wed, 5 Aug 1998 15:59:06 +0200 (DFT)

[Tim]
>
> building-on-examples-tends-to-prevent-abstract-thrashing-ly y'rs - tim
>

OK, I stand corrected. I understand that anybody's interpretation of
the meta-class concept is likely to be difficult to digest by others.

Here's another try, expressing the same thing, but using the Python
programming model, examples and, perhaps, more popular terms.

1. Classes.

   This is pure Python of today. Sorry about the tutorial, but it is
   meant to illustrate the second part, which is the one we're
   interested in and which will follow the same development scenario.
   Besides, newbies are likely to understand that the discussion is
   affordable even for them :-)

   a) Class definition

      A class is meant to define the common properties of a set of objects.
      A class is a "package" of properties. The assembly of properties
      in a class package is sometimes called a class structure (which isn't
      always appropriate).

```
      >>> class A:
              attr1 = "Hello"                  # an attribute of A
              def method1(self, *args): pass   # method1 of A
              def method2(self, *args): pass   # method2 of A
      >>>
```

      So far, we defined the structure of the class A. The class A is
      of type <class>. We can check this by asking Python: "what is A?"

```
      >>> A                                   # What is A?
      <class __main__.A at 2023e360>
```

   b) Class instantiation

      Creating an object with the properties defined in the class A is
      called instantiation of the class A. After an instantiation of A, we
      obtain a new object, called an instance, which has the properties
      packaged in the class A.

```
      >>> a = A()                             # 'a' is the 1st instance of A
      >>> a                                   # What is 'a'?
      <__main__.A instance at 2022b9d0>

      >>> b = A()                             # 'b' is another instance of A
      >>> b                                   # What is 'b'?
      <__main__.A instance at 2022b9c0>
```

      The objects, 'a' and 'b', are of type <instance> and they both have
      the same properties. Note, that 'a' and 'b' are different objects.
      (their adresses differ). This is a bit hard to see, so let's ask Python:

```
      >>> a == b                              # Is 'a' the same object as 'b'?
      0                                       # No.
```

      Instance objects have one more special property, indicating the class
      they are an instance of. This property is named __class__.

```
      >>> a.__class__                         # What is the class of 'a'?
      <class __main__.A at 2023e360>          # 'a' is an instance of A
      >>> b.__class__                         # What is the class of 'b'?
      <class __main__.A at 2023e360>          # 'b' is an instance of A
      >>> a.__class__ == b.__class__          # Is it really the same class A?
      1                                       # Yes.
```

   c) Class inheritance (class composition and specialization)

      Classes can be defined in terms of other existing classes (and only
      classes! -- don't bug me on this now). Thus, we can compose property
      packages and create new ones. We reuse the property set defined
      in a class by defining a new class, which "inherits" from the former.
      In other words, a class B which inherits from the class A, inherits
      the properties defined in A, or, B inherits the structure of A.

      In the same time, at the definition of the new class B, we can enrich
      the inherited set of properties by adding new ones and/or modify some
      of the inherited properties.

```
      >>> class B(A):                             # B inherits A's properties
              attr2 = "World"                    # additional attr2
              def method2(self, arg1): pass      # method2 is redefined
              def method3(self, *args): pass     # additional method3

      >>> B                                  # What is B?
      <class __main__.B at 2023e500>
      >>> B == A                             # Is B the same class as A?
      0                                      # No.
```

Classes define one special property, indicating whether a class inherits the properties of another class. This property is called __bases__ and it contains a list (a tuple) of the classes the new class inherits from. The classes from which a class is inheriting the properties are called superclasses (in Python, we call them also -- base classes).

```
      >>> A.__bases__                        # Does A have any superclasses?
      ()                                     # No.
      >>> B.__bases__                        # Does B have any superclasses?
      (<class __main__.A at 2023e360>,)      # Yes. It has one superclass.
      >>> B.__bases__[0] == A                # Is it really the class A?
      1                                      # Yes, it is.
```

Congratulations on getting this far! This was the hard part.
Now, let's continue with the easy one.

---

2. Meta-classes

You have to admit, that an anonymous group of Python wizards are not satisfied with the property packaging facilities presented above. They say, that the Real-World bugs them with problems that cannot be modelled successfully with classes. Or, that the way classes are implemented in Python and the way classes and instances behave at runtime isn't always appropriate for reproducing the Real-World's behavior in a way that satisfies them.

Hence, what they want is the following:

a) leave objects as they are (instances of classes)
b) leave classes as they are (property packages and object creators)

BUT, at the same time:

c) consider classes as being instances of mysterious objects.
d) label mysterious objects "meta-classes".

Easy, eh?

You may ask: "Why on earth do they want to do that?".
They answer: "Poor soul... Go and see how cruel the Real-World is!".
You - fuzzy: "OK, will do!"

And here we go for another round of what I said in section 1 -- Classes.

However, be warned! The features we're going to talk about aren't fully implemented yet, because the Real-World don't let wizards to evaluate precisely how cruel it is, so the features are still highly-experimental.

a) Meta-class definition

A meta-class is meant to define the common properties of a set of classes. A meta-class is a "package" of properties. The assembly of properties in a meta-class package is sometimes called a meta-class structure (which isn't always appropriate).

In Python, a meta-class definition would have looked like this:

```
      >>> metaclass M:
              attr1 = "Hello"                    # an attribute of M
              def method1(self, *args): pass     # method1 of M
              def method2(self, *args): pass     # method2 of M
      >>>
```

So far, we defined the structure of the meta-class M. The meta-class M is of type <metaclass>. We cannot check this by asking Python, but if we could, it would have answered:

```
      >>> M                                  # What is M?
      <metaclass __main__.M at 2023e4e0>
```

b) Meta-class instantiation

Creating an object with the properties defined in the meta-class M is called instantiation of the meta-class M. After an instantiation of M, we obtain a new object, called an class, but now it is called also a meta-instance, which has the properties packaged in the meta-class M.

In Python, instantiating a meta-class would have looked like this:

```
>>> A = M()                          # 'A' is the 1st instance of M
>>> A                                # What is 'A'?
<class __main__.A at 2022b9d0>

>>> B = M()                          # 'B' is another instance of M
>>> B                                # What is 'B'?
<class __main__.B at 2022b9c0>
```

The metaclass-instances, A and B, are of type <class> and they both have the same properties. Note, that A and B are different objects. (their adresses differ). This is a bit hard to see, but if it was possible to ask Python, it would have answered:

```
>>> A == B                           # Is A the same class as B?
0                                    # No.
```

Class objects have one more special property, indicating the meta-class they are an instance of. This property is named __metaclass__.

```
>>> A.__metaclass__                  # What is the meta-class of A?
<metaclass __main__.M at 2023e4e0>   # A is an instance of M
>>> A.__metaclass__                  # What is the meta-class of B?
<metaclass __main__.M at 2023e4e0>   # B is an instance of M
>>> A.__metaclass__ == B.__metaclass__  # Is it the same meta-class M?
1                                    # Yes.
```

c) Meta-class inheritance (meta-class composition and specialization)

Meta-classes can be defined in terms of other existing meta-classes (and only meta-classes!). Thus, we can compose property packages and create new ones. We reuse the property set defined in a meta-class by defining a new meta-class, which "inherits" from the former.
In other words, a meta-class N which inherits from the meta-class M, inherits the properties defined in M, or, N inherits the structure of M.

In the same time, at the definition of the new meta-class N, we can enrich the inherited set of properties by adding new ones and/or modify some of the inherited properties.

```
>>> metaclass N(M):                      # N inherits M's properties
        attr2 = "World"                  # additional attr2
        def method2(self, arg1): pass    # method2 is redefined
        def method3(self, *args): pass   # additional method3

>>> N                                # What is N?
<metaclass __main__.N at 2023e500>
>>> N == M                           # Is N the same meta-class as M?
0                                    # No.
```

Meta-classes define one special property, indicating whether a meta-class inherits the properties of another meta-class. This property is called __metabases__ and it contains a list (a tuple) of the meta-classes the new meta-class inherits from. The meta-classes from which a meta-class is inheriting the properties are called super-meta-classes (in Python, we call them also -- super meta-bases).

```
>>> M.__metabases__                  # Does M have any supermetaclasses?
()                                   # No.
>>> N.__metabases__                  # Does N have any supermetaclasses?
(<metaclass __main__.M at 2023e360>,)  # Yes. It has a supermetaclass.
>>> N.__metabases__[0] == M          # Is it really the meta-class M?
1                                    # Yes, it is.
```

Triple congratulations on getting this far!
Now you know everything about meta-classes and the Real-World!

<unless-wizards-want-meta-classes-be-instances-of-mysterious-objects!>

--
Vladimir MARANGOZOV | Vladimir.Marangozov@inrialpes.fr
http://sirac.inrialpes.fr/~marangoz | tel:(+33-4)76615277 fax:76615252