The History of Python

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

Wednesday, June 23, 2010

Method Resolution Order

In languages that use multiple inheritance, the order in which base classes are searched when looking for a method is often called the Method Resolution Order, or MRO. (In Python this also applies to other attributes.) For languages that support single inheritance only, the MRO is uninteresting; but when multiple inheritance comes into play, the choice of an MRO algorithm can be remarkably subtle. Python has known at least three different MRO algorithms: classic, Python 2.2 new-style, and Python 2.3 new-style (a.k.a. C3). Only the latter survives in Python 3.

Classic classes used a simple MRO scheme: when looking up a method, base classes were searched using a simple depth-first left-to-right scheme. The first matching object found during this search would be returned. For example, consider these classes:

```
class A:
  def save(self): pass

class B(A): pass

class C:
  def save(self): pass

class D(B, C): pass
```

If we created an instance x of class D, the classic method resolution order would order the classes as D, B, A, C. Thus, a search for the method x.save() would produce A.save() (and not C.save()). This scheme works fine for simple cases, but has problems that become apparent when one considers more complicated uses of multiple inheritance. One problem concerns method lookup under "diamond inheritance." For example:

```
class A:
  def save(self): pass

class B(A): pass

class C(A):
  def save(self): pass

class D(B, C): pass
```

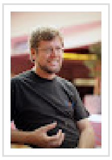
Here, class D inherits from B and C, both of which inherit from class A. Using the classic MRO, methods would be found by searching the classes in the order D, B, A, C, A. Thus, a reference to x.save() will call A.save() as before. However, this is unlikely what you want in this case! Since both B and C inherit from A, one can argue that the redefined method C.save() is actually the method that you want to call, since it can be viewed as being "more specialized" than the method in A (in fact, it probably calls A.save() anyways). For instance, if the save() method is being used to save the state of an object, not calling C.save() would break the

Followers

Followers (1469) Next

Blog Archive

About Me



Guido van Rossum

Python's BDFL

View my complete profile

program since the state of C would be ignored.

Although this kind of multiple inheritance was rare in existing code, new-style classes would make it commonplace. This is because all new-style classes were defined by inheriting from a base class object. Thus, any use of multiple inheritance in new-style classes would always create the diamond relationship described above. For example:

```
class B(object): pass

class C(object):
  def __setattr__(self, name, value): pass

class D(B, C): pass
```

Moreover, since object defined a number of methods that are sometimes extended by subtypes (e.g., __setattr__()), the resolution order becomes critical. For example, in the above code, the method C.__setattr__ should apply to instances of class D.

To fix the method resolution order for new-style classes in Python 2.2, I adopted a scheme where the MRO would be pre-computed when a class was defined and stored as an attribute of each class object. The computation of the MRO was officially documented as using a depth-first left-to-right traversal of the classes as before. If any class was duplicated in this search, all but the last occurrence would be deleted from the MRO list. So, for our earlier example, the search order would be D, B, C, A (as opposed to D, B, A, C, A with classic classes).

In reality, the computation of the MRO was more complex than this. I discovered a few cases where this new MRO algorithm didn't seem to work. Thus, there was a special case to deal with a situation when two bases classes occurred in a different order in the inheritance list of two different derived classes, and both of those classes are inherited by yet another class. For example:

```
class A(object): pass
class B(object): pass
class X(A, B): pass
class Y(B, A): pass
class Z(X, Y): pass
```

Using the tentative new MRO algorithm, the MRO for these classes would be Z, X, Y, B, A, object. (Here 'object' is the universal base class.) However, I didn't like the fact that B and A were in reversed order. Thus, the real MRO would interchange their order to produce Z, X, Y, A, B, object. Intuitively, this algorithm tried to preserve the order of classes for bases that appeared first in the search process. For instance, on class Z, the base class X would be checked first because it was first in the inheritance list. Since X inherited from A and B, the MRO algorithm would try to preserve that ordering. This is what I implemented for Python 2.2, but I documented the earlier algorithm (naïvely thinking it didn't matter much).

However, shortly after the introduction of new-style classes in Python 2.2, Samuele Pedroni discovered an inconsistency between the documented MRO algorithm and the results that were actually observed in real-code. Moreover, inconsistencies were occurring even in code that did not fall under the special case observed above. After much discussion, it was decided that the MRO adopted for Python 2.2 was broken and that Python should adopt the C3 Linearization algorithm described in the paper "A Monotonic Superclass Linearization for Dylan" (K. Barrett, et al, presented at OOPSLA'96).

Essentially, the main problem in the Python 2.2 MRO algorithm concerned the issue of monotonicity. In a complex inheritance hierarchy, each inheritance relationship defines a simple set of rules concerning the order in which classes should be checked. Specifically, if a class A inherits from class B, then the MRO should obviously check A before B. Likewise, if a class B uses multiple inheritance to inherit from C and D, then B should be checked before C and C should be checked before D.

Within a complex inheritance hierarchy, you want to be able to satisfy all of these possible rules in a way that is monotonic. That is, if you have already determined that class A should be checked before class B, then you should never encounter a situation that requires class B to be checked before class A (otherwise, the result is undefined and the inheritance hierarchy should be rejected). This is where the original MRO got it wrong and where the C3 algorithm comes into play. Basically, the idea behind C3 is that if you write down all of the ordering rules imposed by inheritance relationships in a complex class hierarchy, the algorithm will determine a monotonic ordering of the classes that satisfies all of them. If such an ordering can not be determined, the algorithm will fail.

Thus, in Python 2.3, we abandoned my home-grown 2.2 MRO algorithm in favor of the academically vetted C3 algorithm. One outcome of this is that Python will now reject any inheritance hierarchy that has an inconsistent ordering of base classes. For instance, in the previous example, there is an ordering conflict between class X and Y. For class X, there is a rule that says class A should be checked before class B. However, for class Y, the rule says that class B should be checked before A. In isolation, this discrepancy is fine, but if X and Y are ever combined together in the same inheritance hierarchy for another class (such as in the definition of class Z), that class will be rejected by the C3 algorithm. This, of course, matches the Zen of Python's "errors should never pass silently" rule.

Posted by Guido van Rossum at 10:41 AM

10 comments:

Timur Izhbulatov June 26, 2010 at 4:31 AM

I was reading the article late at night before going to bed and had to stop before the three last paragraphs to finish reading in the morning. Otherwise I think I wouldn't digest it properly.

Brief yet exhaustive. Thanks!

Reply

Senthil Kumaran June 26, 2010 at 6:48 PM

Did not know that C3 had academic roots. Interesting and the last part where error should be raised for conflicting inheritance is convincing argument.

Reply

python3 July 5, 2010 at 6:16 PM

Maybe there is a typo, "However, this unlikely want you want in this case!" should be "However, this unlikely what you want in this case!", the first want -> what.

Reply

python3 July 5, 2010 at 9:34 PM

and, Intuitively -> Intuitively

Reply

Guido van Rossum July 5, 2010 at 9:42 PM

Typos fixed - thanks!

Reply

Swapnil Talekar February 7, 2011 at 10:13 AM

According to this post, -
"in Python 2.3, we abandoned my home-grown 2.2 MRO algorithm in favor of the academically vetted C3 algorithm. One outcome of this is that Python will now reject any inheritance hierarchy that has an inconsistent ordering of base classes"

I am unable to get any error if I try such an inconsistent inheritance hierarchy on Python 2.5.4 and Python 2.6.5.
It is only on Python 3.1.2 that I get a TypeError and Python rejects the hierarchy. Why is that if C3 was incorporated in Python 2.3?

Reply

Guido van Rossum February 7, 2011 at 10:18 AM

Swapnil: can you contact me offline (guido@python.org) with the example you are trying?

Reply

Guido van Rossum February 21, 2011 at 9:11 AM

Off-line, Swapnil determined that his problem was caused by not using new-style classes (i.e. not inheriting from object) at the root of his class tree.

Reply

Anonymous May 15, 2011 at 9:06 PM

Anyone in here using python for google appengine ?
I have problem with CPU usage .
My python - Django project eat more than 30 CPU hour per day when less than 2000 request :(,

Reply

Guido van Rossum May 16, 2011 at 6:41 AM

@jewel, I recommend that you write with some more details to this support group: http://groups.google.com/group/google-appengine-python?pli=1

Reply

Enter your comment...

Comment as: Google Accoun

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)