The Python 2.3 Method Resolution Order

Release 3.13.0rc2

Guido van Rossum and the Python development team

September 25, 2024

Python Software Foundation
Email: docs@python.org

Contents

1 The beginning 2

2 The C3 Method Resolution Order 3

3 Examples 4

4 Bad Method Resolution Orders 6

5 The end 9

6 Resources 10

Note

This is a historical document, provided as an appendix to the official documentation. The Method Resolution Order discussed here was *introduced* in Python 2.3, but it is still used in later versions – including Python 3.

By Michele Simionato.

Abstract

This document is intended for Python programmers who want to understand the C3 Method Resolution Order used in Python 2.3. Although it is not intended for newbies, it is quite pedagogical with many worked out examples. I am not aware of other publicly available documents with the same scope, therefore it should be useful.

Disclaimer:

I donate this document to the Python Software Foundation, under the Python 2.3 license. As usual in these circumstances, I warn the reader that what follows should *be correct, but I don't give any warranty. Use it at your own risk and peril!*

Acknowledgments:

All the people of the Python mailing list who sent me their support. Paul Foley who pointed out various imprecisions and made me to add the part on local precedence ordering. David Goodger for help with the

formatting in reStructuredText. David Mertz for help with the editing. Finally, Guido van Rossum who enthusiastically added this document to the official Python 2.3 home-page.

1 The beginning

Felix qui potuit rerum cognoscere causas – Virgilius

Everything started with a post by Samuele Pedroni to the Python development mailing list[1]. In his post, Samuele showed that the Python 2.2 method resolution order is not monotonic and he proposed to replace it with the C3 method resolution order. Guido agreed with his arguments and therefore now Python 2.3 uses C3. The C3 method itself has nothing to do with Python, since it was invented by people working on Dylan and it is described in a paper intended for lispers[2]. The present paper gives a (hopefully) readable discussion of the C3 algorithm for Pythonistas who want to understand the reasons for the change.

First of all, let me point out that what I am going to say only applies to the *new style classes* introduced in Python 2.2: *classic classes* maintain their old method resolution order, depth first and then left to right. Therefore, there is no breaking of old code for classic classes; and even if in principle there could be breaking of code for Python 2.2 new style classes, in practice the cases in which the C3 resolution order differs from the Python 2.2 method resolution order are so rare that no real breaking of code is expected. Therefore:

Don't be scared!

Moreover, unless you make strong use of multiple inheritance and you have non-trivial hierarchies, you don't need to understand the C3 algorithm, and you can easily skip this paper. On the other hand, if you really want to know how multiple inheritance works, then this paper is for you. The good news is that things are not as complicated as you might expect.

Let me begin with some basic definitions.

1) Given a class C in a complicated multiple inheritance hierarchy, it is a non-trivial task to specify the order in which methods are overridden, i.e. to specify the order of the ancestors of C.

2) The list of the ancestors of a class C, including the class itself, ordered from the nearest ancestor to the furthest, is called the class precedence list or the *linearization* of C.

3) The *Method Resolution Order* (MRO) is the set of rules that construct the linearization. In the Python literature, the idiom "the MRO of C" is also used as a synonymous for the linearization of the class C.

4) For instance, in the case of single inheritance hierarchy, if C is a subclass of C1, and C1 is a subclass of C2, then the linearization of C is simply the list [C, C1 , C2]. However, with multiple inheritance hierarchies, the construction of the linearization is more cumbersome, since it is more difficult to construct a linearization that respects *local precedence ordering* and *monotonicity*.

5) I will discuss the local precedence ordering later, but I can give the definition of monotonicity here. A MRO is monotonic when the following is true: *if C1 precedes C2 in the linearization of C, then C1 precedes C2 in the linearization of any subclass of C*. Otherwise, the innocuous operation of deriving a new class could change the resolution order of methods, potentially introducing very subtle bugs. Examples where this happens will be shown later.

6) Not all classes admit a linearization. There are cases, in complicated hierarchies, where it is not possible to derive a class such that its linearization respects all the desired properties.

Here I give an example of this situation. Consider the hierarchy

```
>>> O = object
>>> class X(O): pass
>>> class Y(O): pass
>>> class A(X,Y): pass
>>> class B(Y,X): pass
```

[1] The thread on python-dev started by Samuele Pedroni: https://mail.python.org/pipermail/python-dev/2002-October/029035.html
[2] The paper *A Monotonic Superclass Linearization for Dylan*: https://doi.org/10.1145/236337.236343

which can be represented with the following inheritance graph, where I have denoted with O the `object` class, which is the beginning of any hierarchy for new style classes:

```
 -----------
|           |
|    O      |
|  /   \    |
 - X    Y  /
   |  / | /
   | /  |/
   A    B
   \   /
     ?
```

In this case, it is not possible to derive a new class C from A and B, since X precedes Y in A, but Y precedes X in B, therefore the method resolution order would be ambiguous in C.

Python 2.3 raises an exception in this situation (TypeError: MRO conflict among bases Y, X) forbidding the naive programmer from creating ambiguous hierarchies. Python 2.2 instead does not raise an exception, but chooses an *ad hoc* ordering (CABXYO in this case).

2 The C3 Method Resolution Order

Let me introduce a few simple notations which will be useful for the following discussion. I will use the shortcut notation:

```
C1 C2 ... CN
```

to indicate the list of classes [C1, C2, … , CN].

The *head* of the list is its first element:

```
head = C1
```

whereas the *tail* is the rest of the list:

```
tail = C2 ... CN.
```

I shall also use the notation:

```
C + (C1 C2 ... CN) = C C1 C2 ... CN
```

to denote the sum of the lists [C] + [C1, C2, … ,CN].

Now I can explain how the MRO works in Python 2.3.

Consider a class C in a multiple inheritance hierarchy, with C inheriting from the base classes B1, B2, … , BN. We want to compute the linearization L[C] of the class C. The rule is the following:

> *the linearization of C is the sum of C plus the merge of the linearizations of the parents and the list of the parents.*

In symbolic notation:

```
L[C(B1 ... BN)] = C + merge(L[B1] ... L[BN], B1 ... BN)
```

In particular, if C is the `object` class, which has no parents, the linearization is trivial:

```
L[object] = object.
```

However, in general one has to compute the merge according to the following prescription:

take the head of the first list, i.e L[B1][0]; if this head is not in the tail of any of the other lists, then add it to the linearization of C and remove it from the lists in the merge, otherwise look at the head of the next list and take it, if it is a good head. Then repeat the operation until all the class are removed or it is impossible to find good heads. In this case, it is impossible to construct the merge, Python 2.3 will refuse to create the class C and will raise an exception.

This prescription ensures that the merge operation *preserves* the ordering, if the ordering can be preserved. On the other hand, if the order cannot be preserved (as in the example of serious order disagreement discussed above) then the merge cannot be computed.

The computation of the merge is trivial if C has only one parent (single inheritance); in this case:

```
L[C(B)] = C + merge(L[B],B) = C + L[B]
```

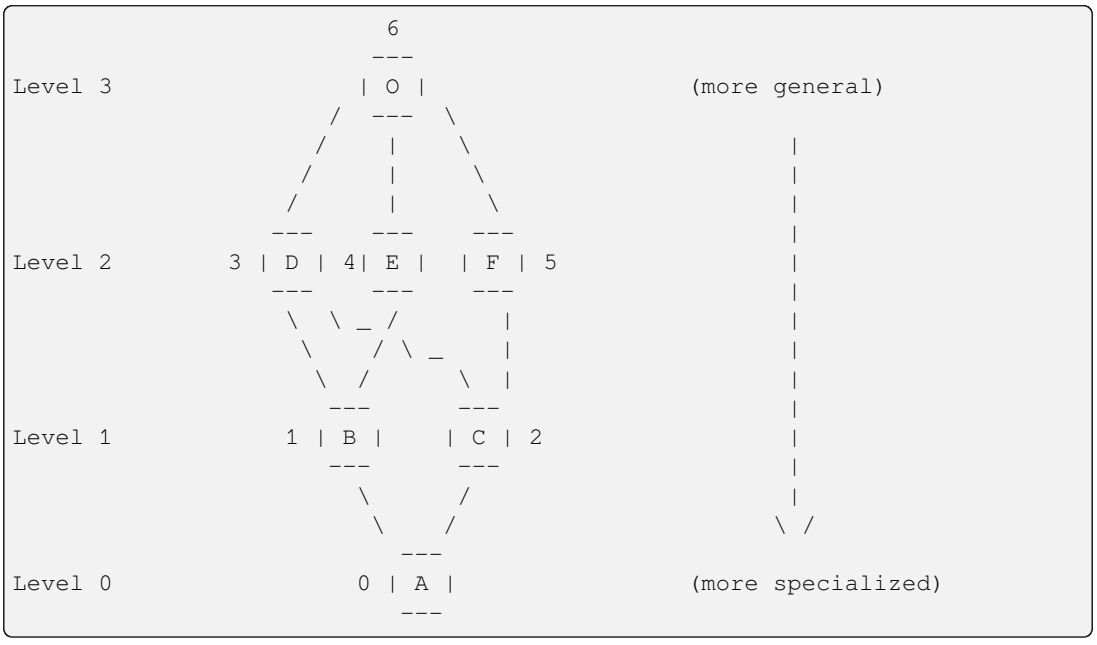
However, in the case of multiple inheritance things are more cumbersome and I don't expect you can understand the rule without a couple of examples ;-)

3 Examples

First example. Consider the following hierarchy:

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(D,E): pass
>>> class A(B,C): pass
```

In this case the inheritance graph can be drawn as:



The linearizations of O,D,E and F are trivial:

```
L[O] = O
L[D] = D O
L[E] = E O
L[F] = F O
```

The linearization of B can be computed as:

```
L[B] = B + merge(DO, EO, DE)
```

We see that D is a good head, therefore we take it and we are reduced to compute `merge(O,EO,E)`. Now O is not a good head, since it is in the tail of the sequence EO. In this case the rule says that we have to skip to the next sequence. Then we see that E is a good head; we take it and we are reduced to compute `merge(O,O)` which gives O. Therefore:

```
L[B] =  B D E O
```

Using the same procedure one finds:

```
L[C] = C + merge(DO,FO,DF)
     = C + D + merge(O,FO,F)
     = C + D + F + merge(O,O)
     = C D F O
```

Now we can compute:

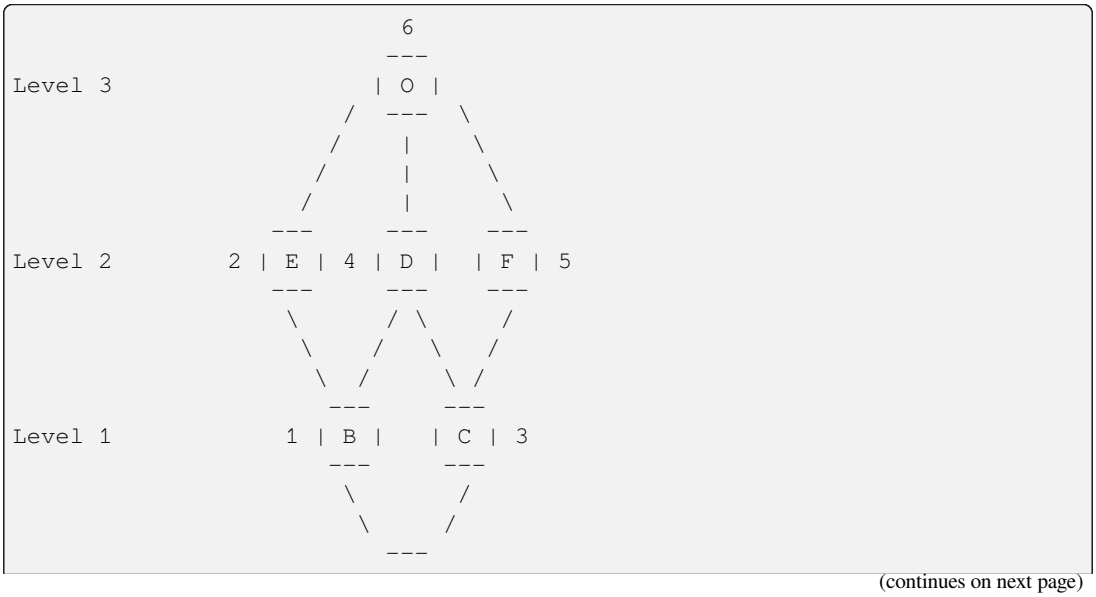
```
L[A] = A + merge(BDEO,CDFO,BC)
     = A + B + merge(DEO,CDFO,C)
     = A + B + C + merge(DEO,DFO)
     = A + B + C + D + merge(EO,FO)
     = A + B + C + D + E + merge(O,FO)
     = A + B + C + D + E + F + merge(O,O)
     = A B C D E F O
```

In this example, the linearization is ordered in a pretty nice way according to the inheritance level, in the sense that lower levels (i.e. more specialized classes) have higher precedence (see the inheritance graph). However, this is not the general case.

I leave as an exercise for the reader to compute the linearization for my second example:

```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(E,D): pass
>>> class A(B,C): pass
```

The only difference with the previous example is the change B(D,E) –> B(E,D); however even such a little modification completely changes the ordering of the hierarchy:



(continues on next page)

(continued from previous page)

```
Level 0                 0 | A |
                            ---
```

Notice that the class E, which is in the second level of the hierarchy, precedes the class C, which is in the first level of the hierarchy, i.e. E is more specialized than C, even if it is in a higher level.

A lazy programmer can obtain the MRO directly from Python 2.2, since in this case it coincides with the Python 2.3 linearization. It is enough to invoke the .mro() method of class A:

```
>>> A.mro()
[<class 'A'>, <class 'B'>, <class 'E'>,
<class 'C'>, <class 'D'>, <class 'F'>,
<class 'object'>]
```

Finally, let me consider the example discussed in the first section, involving a serious order disagreement. In this case, it is straightforward to compute the linearizations of O, X, Y, A and B:

```
L[O] = O
L[X] = X O
L[Y] = Y O
L[A] = A X Y O
L[B] = B Y X O
```

However, it is impossible to compute the linearization for a class C that inherits from A and B:

```
L[C] = C + merge(AXYO, BYXO, AB)
     = C + A + merge(XYO, BYXO, B)
     = C + A + B + merge(XYO, YXO)
```

At this point we cannot merge the lists XYO and YXO, since X is in the tail of YXO whereas Y is in the tail of XYO: therefore there are no good heads and the C3 algorithm stops. Python 2.3 raises an error and refuses to create the class C.

4 Bad Method Resolution Orders

A MRO is *bad* when it breaks such fundamental properties as local precedence ordering and monotonicity. In this section, I will show that both the MRO for classic classes and the MRO for new style classes in Python 2.2 are bad.

It is easier to start with the local precedence ordering. Consider the following example:

```
>>> F=type('Food',(),{'remember2buy':'spam'})
>>> E=type('Eggs',(F,),{'remember2buy':'eggs'})
>>> G=type('GoodFood',(F,E),{}) # under Python 2.3 this is an error!
```

with inheritance diagram

```
             O
             |
(buy spam)   F
             | \
             | E    (buy eggs)
             | /
             G

      (buy eggs or spam ?)
```

We see that class G inherits from F and E, with F *before* E: therefore we would expect the attribute *G.remember2buy* to be inherited by *F.rembermer2buy* and not by *E.remember2buy*: nevertheless Python 2.2 gives

```
>>> G.remember2buy
'eggs'
```

This is a breaking of local precedence ordering since the order in the local precedence list, i.e. the list of the parents of G, is not preserved in the Python 2.2 linearization of G:

```
L[G,P22]= G E F object    # F *follows* E
```

One could argue that the reason why F follows E in the Python 2.2 linearization is that F is less specialized than E, since F is the superclass of E; nevertheless the breaking of local precedence ordering is quite non-intuitive and error prone. This is particularly true since it is a different from old style classes:

```
>>> class F: remember2buy='spam'
>>> class E(F): remember2buy='eggs'
>>> class G(F,E): pass
>>> G.remember2buy
'spam'
```

In this case the MRO is GFEF and the local precedence ordering is preserved.

As a general rule, hierarchies such as the previous one should be avoided, since it is unclear if F should override E or vice-versa. Python 2.3 solves the ambiguity by raising an exception in the creation of class G, effectively stopping the programmer from generating ambiguous hierarchies. The reason for that is that the C3 algorithm fails when the merge:

```
merge(FO,EFO,FE)
```

cannot be computed, because F is in the tail of EFO and E is in the tail of FE.

The real solution is to design a non-ambiguous hierarchy, i.e. to derive G from E and F (the more specific first) and not from F and E; in this case the MRO is GEF without any doubt.

```
            O
            |
            F (spam)
          / |
(eggs)    E |
          \ |
            G
              (eggs, no doubt)
```

Python 2.3 forces the programmer to write good hierarchies (or, at least, less error-prone ones).

On a related note, let me point out that the Python 2.3 algorithm is smart enough to recognize obvious mistakes, as the duplication of classes in the list of parents:

```
>>> class A(object): pass
>>> class C(A,A): pass # error
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: duplicate base class A
```

Python 2.2 (both for classic classes and new style classes) in this situation, would not raise any exception.

Finally, I would like to point out two lessons we have learned from this example:

1. despite the name, the MRO determines the resolution order of attributes, not only of methods;
2. the default food for Pythonistas is spam ! (but you already knew that ;-)

Having discussed the issue of local precedence ordering, let me now consider the issue of monotonicity. My goal is to show that neither the MRO for classic classes nor that for Python 2.2 new style classes is monotonic.

To prove that the MRO for classic classes is non-monotonic is rather trivial, it is enough to look at the diamond diagram:

```
   C
  / \
 /   \
A     B
 \   /
  \ /
   D
```

One easily discerns the inconsistency:

```
L[B,P21] = B C        # B precedes C : B's methods win
L[D,P21] = D A C B C  # B follows C  : C's methods win!
```

On the other hand, there are no problems with the Python 2.2 and 2.3 MROs, they give both:

```
L[D] = D A B C
```

Guido points out in his essay[3] that the classic MRO is not so bad in practice, since one can typically avoids diamonds for classic classes. But all new style classes inherit from `object`, therefore diamonds are unavoidable and inconsistencies shows up in every multiple inheritance graph.

The MRO of Python 2.2 makes breaking monotonicity difficult, but not impossible. The following example, originally provided by Samuele Pedroni, shows that the MRO of Python 2.2 is non-monotonic:

```
>>> class A(object): pass
>>> class B(object): pass
>>> class C(object): pass
>>> class D(object): pass
>>> class E(object): pass
>>> class K1(A,B,C): pass
>>> class K2(D,B,E): pass
>>> class K3(D,A):   pass
>>> class Z(K1,K2,K3): pass
```

Here are the linearizations according to the C3 MRO (the reader should verify these linearizations as an exercise and draw the inheritance diagram ;-)

```
L[A] = A O
L[B] = B O
L[C] = C O
L[D] = D O
L[E] = E O
L[K1]= K1 A B C O
L[K2]= K2 D B E O
L[K3]= K3 D A O
L[Z] = Z K1 K2 K3 D A B C E O
```

Python 2.2 gives exactly the same linearizations for A, B, C, D, E, K1, K2 and K3, but a different linearization for Z:

```
L[Z,P22] = Z K1 K3 A K2 D B C E O
```

It is clear that this linearization is *wrong*, since A comes before D whereas in the linearization of K3 A comes *after* D. In other words, in K3 methods derived by D override methods derived by A, but in Z, which still is a subclass of K3, methods derived by A override methods derived by D! This is a violation of monotonicity. Moreover, the Python 2.2 linearization of Z is also inconsistent with local precedence ordering, since the local precedence list of the class Z is [K1, K2, K3] (K2 precedes K3), whereas in the linearization of Z K2 *follows* K3. These problems explain why the 2.2 rule has been dismissed in favor of the C3 rule.

[3] Guido van Rossum's essay, *Unifying types and classes in Python 2.2*: https://web.archive.org/web/20140210194412/http://www.python.org/download/releases/2.2.2/descrintro

5 The end

This section is for the impatient reader, who skipped all the previous sections and jumped immediately to the end. This section is for the lazy programmer too, who didn't want to exercise her/his brain. Finally, it is for the programmer with some hubris, otherwise s/he would not be reading a paper on the C3 method resolution order in multiple inheritance hierarchies ;-) These three virtues taken all together (and *not* separately) deserve a prize: the prize is a short Python 2.2 script that allows you to compute the 2.3 MRO without risk to your brain. Simply change the last line to play with the various examples I have discussed in this paper.:

```
#<mro.py>

"""C3 algorithm by Samuele Pedroni (with readability enhanced by me)."""

class __metaclass__(type):
    "All classes are metamagically modified to be nicely printed"
    __repr__ = lambda cls: cls.__name__

class ex_2:
    "Serious order disagreement" #From Guido
    class O: pass
    class X(O): pass
    class Y(O): pass
    class A(X,Y): pass
    class B(Y,X): pass
    try:
        class Z(A,B): pass #creates Z(A,B) in Python 2.2
    except TypeError:
        pass # Z(A,B) cannot be created in Python 2.3

class ex_5:
    "My first example"
    class O: pass
    class F(O): pass
    class E(O): pass
    class D(O): pass
    class C(D,F): pass
    class B(D,E): pass
    class A(B,C): pass

class ex_6:
    "My second example"
    class O: pass
    class F(O): pass
    class E(O): pass
    class D(O): pass
    class C(D,F): pass
    class B(E,D): pass
    class A(B,C): pass

class ex_9:
    "Difference between Python 2.2 MRO and C3" #From Samuele
    class O: pass
    class A(O): pass
    class B(O): pass
    class C(O): pass
    class D(O): pass
    class E(O): pass
    class K1(A,B,C): pass
    class K2(D,B,E): pass
    class K3(D,A): pass
    class Z(K1,K2,K3): pass
```

(continues on next page)

(continued from previous page)

```
def merge(seqs):
    print '\n\nCPL[%s]=%s' % (seqs[0][0],seqs),
    res = []; i=0
    while 1:
      nonemptyseqs=[seq for seq in seqs if seq]
      if not nonemptyseqs: return res
      i+=1; print '\n',i,'round: candidates...',
      for seq in nonemptyseqs: # find merge candidates among seq heads
          cand = seq[0]; print ' ',cand,
          nothead=[s for s in nonemptyseqs if cand in s[1:]]
          if nothead: cand=None #reject candidate
          else: break
      if not cand: raise "Inconsistent hierarchy"
      res.append(cand)
      for seq in nonemptyseqs: # remove cand
          if seq[0] == cand: del seq[0]

def mro(C):
    "Compute the class precedence list (mro) according to C3"
    return merge([[C]]+map(mro,C.__bases__)+[list(C.__bases__)])

def print_mro(C):
    print '\nMRO[%s]=%s' % (C,mro(C))
    print '\nP22 MRO[%s]=%s' % (C,C.mro())

print_mro(ex_9.Z)

#</mro.py>
```

That's all folks,

enjoy !

6 Resources