
A Ordem de Resolução de Métodos do Python 2.3

Release 3.12.4

Guido van Rossum and the Python development team

julho 31, 2024

Python Software Foundation
Email: docs@python.org

Sumário

1	O início	2
2	A ordem de resolução de métodos C3	3
3	Exemplos	4
4	Ordens de resolução de métodos ruins	6
5	O fim	9
6	Recursos	10

Nota

Este é um documento histórico, fornecido como apêndice à documentação oficial. A ordem de resolução de métodos discutida aqui foi *introduzida* no Python 2.3, mas ainda é usada em versões posteriores – incluindo o Python 3.

por Michele Simionato.

Resumo

Este documento é destinado a programadores Python que desejam entender a ordem de resolução de métodos C3 usada no Python 2.3. Embora não seja destinado a iniciantes, é bastante pedagógico com muitos exemplos elaborados. Não tenho conhecimento de outros documentos publicamente disponíveis com o mesmo escopo, portanto devem ser úteis.

Aviso:

Eu doo este documento para a Python Software Foundation, sob a licença Python 2.3. Como de costume nestas circunstâncias, aviso o leitor que o que se segue deve estar correto, mas não dou nenhuma garantia. Use-o por sua própria conta e risco!

Reconhecimentos:

Todas as pessoas da lista de discussão do Python que me enviaram seu apoio. Paul Foley, que apontou várias imprecisões e me fez acrescentar a parte sobre ordem de precedência local. David Goodger pela ajuda com a formatação em reStructuredText. David Mertz pela ajuda com a edição. Finalmente, Guido van Rossum que adicionou com entusiasmo este documento à página inicial oficial do Python 2.3.

1 O início

Felix qui potuit rerum cognoscere causas – Virgilius

Tudo começou com uma postagem de Samuele Pedroni na lista de discussão de desenvolvimento Python¹. Em sua postagem, Samuele mostrou que a ordem de resolução de métodos do Python 2.2 não é monotônica e propôs substituí-la pela ordem de resolução de métodos C3. Guido concordou com seus argumentos e, portanto, agora o Python 2.3 usa C3. O método C3 em si não tem nada a ver com Python, pois foi inventado por pessoas que trabalharam em Dylan e está descrito em um artigo destinado a lispers². O presente artigo fornece uma discussão (espero) legível do algoritmo C3 para Pythonistas que desejam entender os motivos da mudança.

Primeiro de tudo, deixe-me salientar que o que vou dizer se aplica apenas às *classes no novo estilo* introduzidas no Python 2.2: *classes clássicas* mantêm sua antiga ordem de resolução de métodos, profundidade primeiro e depois da esquerda para a direita. Portanto, não há quebra de código antigo para classes clássicas; e mesmo que em princípio pudesse haver quebra de código para novas classes de estilo do Python 2.2, na prática os casos em que a ordem de resolução C3 difere da ordem de resolução de métodos do Python 2.2 são tão raros que nenhuma quebra real de código é esperada. Portanto:

Não tenha medo!

Além disso, a menos que você faça uso intenso de herança múltipla e tenha hierarquias não triviais, você não precisa entender o algoritmo C3 e pode facilmente pular este artigo. Por outro lado, se você realmente deseja saber como funciona a herança múltipla, este artigo é para você. A boa notícia é que as coisas não são tão complicadas quanto você imagina.

Deixe-me começar com algumas definições básicas.

- 1) Dada uma classe C em uma complicada hierarquia de herança múltipla, não é uma tarefa trivial especificar a ordem na qual os métodos são substituídos, ou seja, especificar a ordem dos ancestrais de C.
- 2) A lista dos ancestrais de uma classe C, incluindo a própria classe, ordenada do ancestral mais próximo ao mais distante, é chamada de lista de precedência de classe ou *linearização* de C.
- 3) A *Ordem de Resolução de Métodos* (em inglês Method Resolution Order, MRO) é o conjunto de regras que constroem a linearização. Na literatura Python, a expressão “a MRO de C” também é usada como sinônimo de linearização da classe C.
- 4) Por exemplo, no caso de hierarquia de herança única, se C é uma subclasse de C1 e C1 é uma subclasse de C2, então a linearização de C é simplesmente a lista [C, C1, C2]. No entanto, com múltiplas hierarquias de herança, a construção da linearização é mais complicada, pois é mais difícil construir uma linearização que respeite a *ordem de precedência local* e a *monotonicidade*.
- 5) Discutirei a ordem de precedência local mais tarde, mas posso dar aqui a definição de monotonicidade. Uma MRO é monotônico quando o seguinte é verdadeiro: *se C1 precede C2 na linearização de C, então C1 precede C2 na linearização de qualquer subclasse de C*. Caso contrário, a operação inócua de derivar uma nova classe poderia alterar a ordem de resolução dos métodos, potencialmente introduzindo bugs muito sutis. Exemplos onde isso acontece serão mostrados posteriormente.
- 6) Nem todas as classes admitem uma linearização. Existem casos, em hierarquias complicadas, em que não é possível derivar uma classe tal que a sua linearização respeite todas as propriedades desejadas.

Aqui dou um exemplo desta situação. Considere a hierarquia

¹ O tópico no python-dev iniciado por Samuele Pedroni: <https://mail.python.org/pipermail/python-dev/2002-October/029035.html>

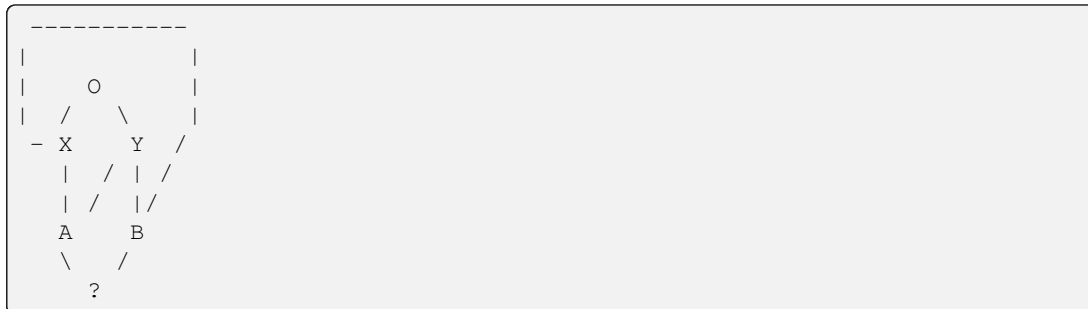
² O artigo *Uma Linearização Monotônica de Superclasses para Dylan*, em inglês: <https://doi.org/10.1145/236337.236343>

```

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

```

que pode ser representada com o seguinte grafo de herança, onde denotamos com O a classe `object`, que é o início de qualquer hierarquia para classes de novo estilo:



Neste caso, não é possível derivar uma nova classe C de A e B, pois X precede Y em A, mas Y precede X em B, portanto a ordem de resolução de métodos seria ambígua em C.

Python 2.3 levanta uma exceção nesta situação (`TypeError: MRO conflict among bases Y, X`) proibindo o programador ingênuo de criar hierarquias ambíguas. Em vez disso, o Python 2.2 não levanta uma exceção, mas escolhe uma ordem *ad hoc* (CABXYO neste caso).

2 A ordem de resolução de métodos C3

Deixe-me apresentar algumas notações simples que serão úteis para a discussão a seguir. Usarei a notação de atalho:

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

para indicar a lista de classes $[C1, C2, \dots, CN]$.

head da lista é o seu primeiro elemento:

```
head = C1
```

enquanto *tail* é o resto da lista:

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

Também usarei a notação:

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

para denotar a soma das listas $[C] + [C1, C2, \dots, CN]$.

Agora posso explicar como funciona a MRO no Python 2.3.

Considere uma classe C em uma hierarquia de herança múltipla, com C herdando das classes base B1, B2, ..., BN. Queremos calcular a linearização L[C] da classe C. A regra é a seguinte:

a linearização de C é a soma de C mais a mesclagem das linearizações dos pais e da lista dos pais.

Em notação simbólica:

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

Em particular, se C é a classe `object`, que não tem pais, a linearização é trivial:

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

Contudo, em geral, deve-se calcular a mesclagem de acordo com a seguinte prescrição:

considere o topo da primeira lista, ou seja, $L[B1][0]$; se esse head não estiver no final de nenhuma das outras listas, então adicione-o à linearização de C e remova-o das listas na mesclagem, caso contrário olhe para o head da próxima lista e pegue-o, se for um bom head. Em seguida, repita a operação até que todas as classes sejam removidas ou seja impossível encontrar boas cabeças. Neste caso, é impossível construir a mesclagem, o Python 2.3 se recusará a criar a classe C e vai levantar uma exceção.

Esta prescrição garante que a operação de mesclagem *preserva* a ordem, se a ordem puder ser preservada. Por outro lado, se a ordem não puder ser preservada (como no exemplo de desacordo sério sobre ordem discutido acima), então a mesclagem não poderá ser calculada.

O cálculo da mesclagem é trivial se C tiver apenas um pai (herança única); nesse caso:

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

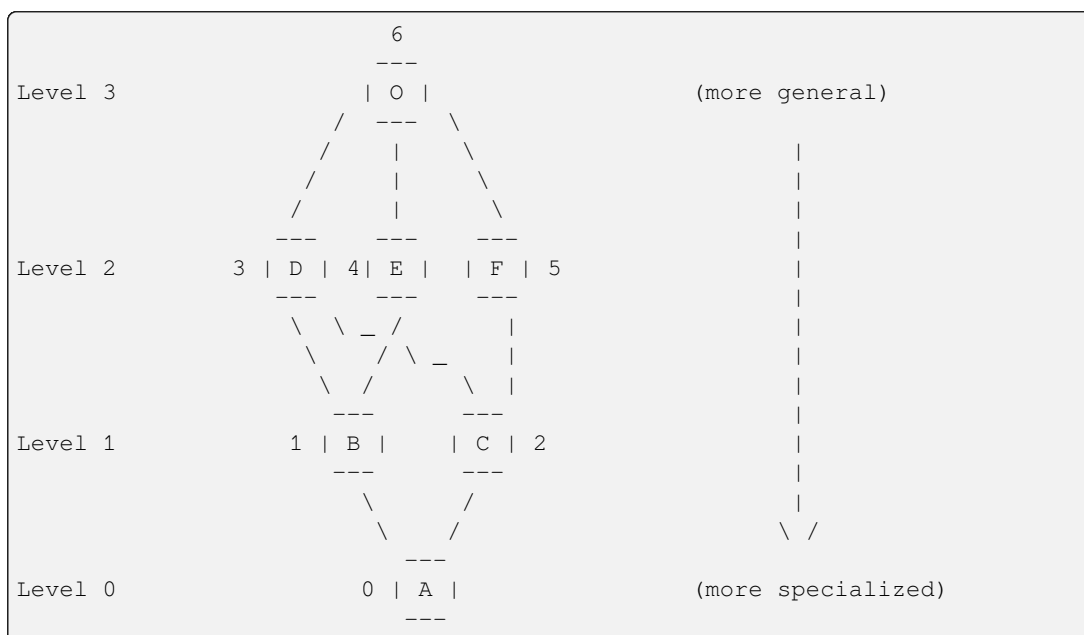
No entanto, no caso de herança múltipla, as coisas são mais complicadas e não espero que você consiga entender a regra sem alguns exemplos ;-)

3 Exemplos

Primeiro exemplo. Considere a seguinte hierarquia:

```
>>> 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
```

Neste caso, o grafo de herança pode ser desenhado como:



As linearizações de O, D, E e F são triviais:

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

A linearização de B pode ser calculada como:

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

Vemos que D é um bom *head*, portanto pegamos ele e ficamos reduzidos a calcular `merge(O, EO, E)`. Agora O não é um bom *head*, pois está no *tail* da sequência EO. Neste caso a regra diz que temos que pular para a próxima sequência. Então vemos que E é um bom *head*; nós pegamos isso e somos reduzidos a calcular `merge(O, O)` que dá O. Portanto:

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

Usando o mesmo procedimento encontra-se:

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

Agora podemos calcular:

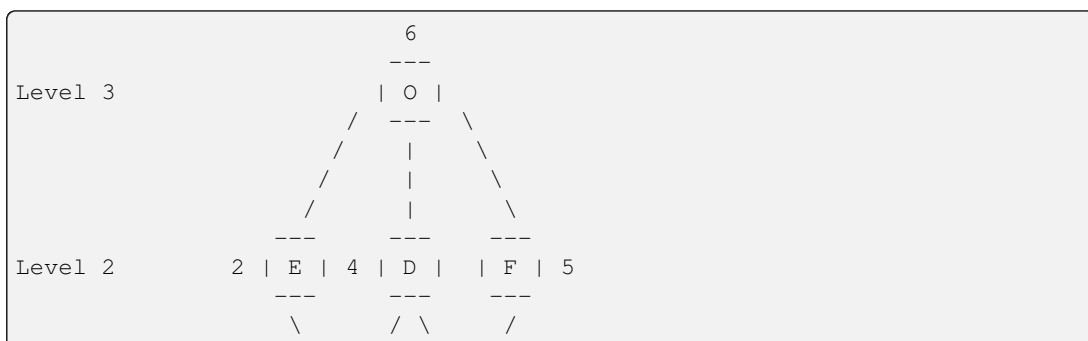
```
L[A] = A + merge(BDEO, CDFO, BC)
      = A + B + merge(DEO, CDFO, C)
      = A + B + C + merge(DEO, DFO)
      = A + B + C + D + merge(EO, FO)
      = A + B + C + D + E + merge(O, FO)
      = A + B + C + D + E + F + merge(O, O)
      = A B C D E F O
```

Neste exemplo, a linearização é ordenada de maneira bastante agradável de acordo com o nível de herança, no sentido de que níveis mais baixos (ou seja, classes mais especializadas) têm precedência mais alta (veja o grafo de herança). No entanto, este não é o caso geral.

Deixo como exercício para o leitor calcular a linearização do meu segundo exemplo:

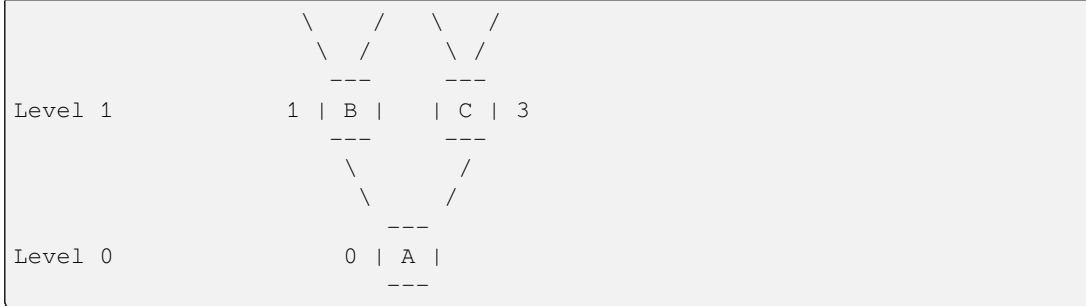
```
>>> O = object
>>> class F(O): pass
>>> class E(O): pass
>>> class D(O): pass
>>> class C(D,F): pass
>>> class B(E,D): pass
>>> class A(B,C): pass
```

A única diferença com o exemplo anterior é a mudança `B(D,E) -> B(E,D)`; no entanto, mesmo uma pequena modificação muda completamente a ordem da hierarquia:



(continua na próxima página)

(continuação da página anterior)



Observe que a classe E, que está no segundo nível da hierarquia, precede a classe C, que está no primeiro nível da hierarquia, ou seja, E é mais especializado que C, mesmo que esteja em um nível superior.

Um programador preguiçoso pode obter a MRO diretamente do Python 2.2, pois neste caso ela coincide com a linearização do Python 2.3. Basta invocar o método `.mro()` da classe A:

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

Finalmente, deixe-me considerar o exemplo discutido na primeira seção, envolvendo um sério desacordo de ordem. Neste caso, é simples calcular as linearizações de O, X, Y, A e B:

```
L[O] = 0
L[X] = X O
L[Y] = Y O
L[A] = A X Y O
L[B] = B Y X O
```

Porém, é impossível calcular a linearização para uma classe C que herda de A e B:

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

Neste ponto não podemos mesclar as listas XYO e YXO, uma vez que X está no *tail* de YXO enquanto Y está no *tail* de XYO: portanto não há bons *head* e o algoritmo C3 para. Python 2.3 levanta um erro e se recusa a criar a classe C.

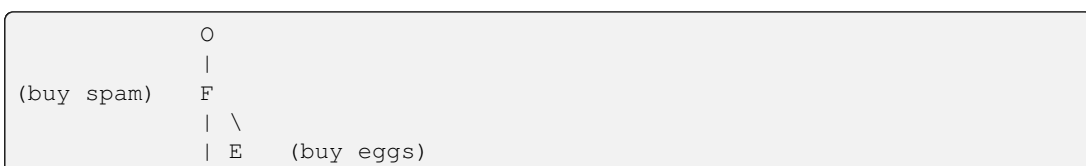
4 Ordens de resolução de métodos ruins

Uma MRO é *ruim* quando quebra propriedades fundamentais como ordem de precedência local e monotonicidade. Nesta seção, mostrarei que tanto a MRO para classes clássicas quanto a MRO para classes de novo estilo em Python 2.2 são ruins.

É mais fácil começar com a ordem de precedência local. Considere o seguinte exemplo:

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

com diagrama de herança



(continua na próxima página)

```

      | /
      G
      (buy eggs or spam ?)

```

Vemos que a classe G herda de F e E, com F *antes* de E: portanto, esperaríamos que o atributo `G.remember2buy` fosse herdado por `F.remember2buy` e não por `E.remember2buy`: no entanto, Python 2.2 dá

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

Isto é uma quebra da ordem de precedência local, uma vez que a ordem na lista de precedência local, ou seja, a lista dos pais de G, não é preservada na linearização de G em Python 2.2:

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

Pode-se argumentar que a razão pela qual F segue E na linearização do Python 2.2 é que F é menos especializado que E, uma vez que F é a superclasse de E; no entanto, a quebra da ordem de precedência local é bastante não intuitiva e sujeita a erros. Isto é particularmente verdadeiro porque é diferente das classes de estilo antigo:

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

Neste caso a MRO é GFEF e a ordem de precedência local é preservada.

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)
```

não puder ser calculada, porque F está no *tail* de EFO e E está no *tail* de FE.

A verdadeira solução é conceber uma hierarquia não ambígua, ou seja, derivar G de E e F (o mais específico primeiro) e não de F e E; neste caso a MRO é GEF, sem dúvida.

```

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

```

Python 2.3 força o programador a escrever boas hierarquias (ou, pelo menos, menos propensas a erros).

Falando nisso, deixe-me salientar que o algoritmo Python 2.3 é inteligente o suficiente para reconhecer erros óbvios, como a duplicação de classes na lista de pais:

```
>>> 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
```

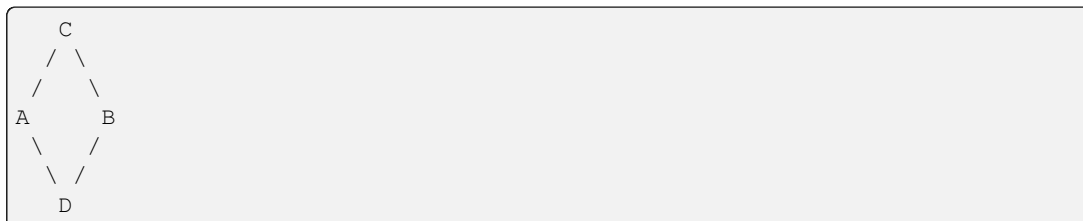
Python 2.2 (tanto para classes clássicas quanto para classes de novo estilo) nesta situação, não levantaria nenhuma exceção.

Por fim, gostaria de destacar duas lições que aprendemos com este exemplo:

1. apesar do nome, a MRO determina a ordem de resolução dos atributos, não apenas dos métodos;
2. o alimento padrão para Pythonistas é spam! (mas você já sabia disso ;-)

Tendo discutido a questão da ordem de precedência local, deixe-me agora considerar a questão da monotonicidade. Meu objetivo é mostrar que nem a MRO para classes clássicas nem para o novo estilo de classes do Python 2.2 são monotônicas.

Para provar que a MRO para classes clássicas não é monotônica é bastante trivial, basta olhar o diagrama em losango:



Percebe-se facilmente a inconsistência:

```
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!
```

Por outro lado, não há problemas com as MROs do Python 2.2 e do 2.3, elas fornecem ambos:

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

Guido ressalta em seu ensaio³ que a MRO clássica não é tão ruim na prática, já que normalmente se pode evitar losangos para classes clássicas. Mas todas as classes de novo estilo herdam de `object`, portanto os diamantes são inevitáveis e inconsistências aparecem em cada grafo de herança múltipla.

A MRO do Python 2.2 torna difícil quebrar a monotonicidade, mas não impossível. O exemplo a seguir, fornecido originalmente por Samuele Pedroni, mostra que a MRO do Python 2.2 não é monotônica:

```
>>> 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
```

Aqui estão as linearizações de acordo com a MRO C3 (o leitor deverá verificar essas linearizações como exercício e desenhar o diagrama de herança ;-)

```
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 fornece exatamente as mesmas linearizações para A, B, C, D, E, K1, K2 e K3, mas uma linearização diferente para Z:

³ Ensaio de Guido van Rossum, *Unificando tipos e classes em Python 2.2*, em inglês: <https://web.archive.org/web/20140210194412/http://www.python.org/download/releases/2.2.2/descrintro>


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

É claro que esta linearização está *errada*, uma vez que A vem antes de D, enquanto na linearização de K3 A vem *depois* de D. Em outras palavras, em métodos K3 derivados de D substituem os métodos derivados de A, mas em Z, que ainda é uma subclasse de K3, os métodos derivados de A substituem os métodos derivados de D! Isto é uma violação da monotonicidade. Além disso, a linearização de Z do Python 2.2 também é inconsistente com a ordem de precedência local, uma vez que a lista de precedência local da classe Z é [K1, K2, K3] (K2 precede K3), enquanto na linearização de Z K2 *segue* K3. Estes problemas explicam porque é que a regra do 2.2 foi rejeitada em favor da regra C3.

5 O fim

Esta seção é para o leitor impaciente, que passou direto por todas as seções anteriores e pulou imediatamente para o final. Esta seção também é para o programador preguiçoso, que não quer exercitar seu cérebro. Finalmente, é para o programador com alguma arrogância, caso contrário ele/ela não estaria lendo um artigo sobre a ordem de resolução de métodos C3 em múltiplas hierarquias de herança ;-) Essas três virtudes tomadas em conjunto (e *não* separadamente) merecem um prêmio: o prêmio é um pequeno script em Python 2.2 que permite calcular a MRO do 2.3 sem risco para o seu cérebro. Basta alterar a última linha para brincar com os vários exemplos que discuti neste artigo.:

```
#<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
```

(continua na próxima página)

```

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

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>

```

Isso é tudo, pessoal!

Divirtam-se !

6 Recursos