Порядок разрешения методов в Python

В этой заметке рассматривается алгоритм MRO С3 и некоторые специфические проблемы множественного наследования. Хотя и алгоритм и проблемы не ограничиваются рамками одного языка, я акцентировал своё внимание на Питоне. В конце приведён список полезных ссылок по данной теме.

Порядок разрешения методов (method resolution order) позволяет Питону выяснить, из какого класса-предка нужно вызывать метод, если он не обнаружен непосредственно в классе-потомке. Если у каждого потомка есть только один предок, то задача тривиальна. Происходит восходящий поиск по всей иерархии. Если же используется множественное наследование, то можно столкнуться со специфическими проблемами, которые описаны ниже.

В старых версиях Питона порядок разрешения методов был достаточно примитивным: поиск вёлся во всех родительских классах слева направо на максимальную глубину. Т.е. если у родительского класса в свою очередь не было нужного метода, но были родители, то поиск производился в них по тем же правилам. Для примера возьмём структуру:

A C

| |

B D

\ /

E

При обращении к методу экземпляра класса E такой алгоритм произвёл бы поиск последовательно в классах E, B, A, D и C. Таким образом поиск вёлся сначала в первом классе-родителе и во всех его предках, затем во втором классе-родителе со всеми предками и т. д. Этот способ не вызывал особых нареканий, пока у классов не было общего предка. Однако, начиная с версии 2.2, появился базовый класс object, от которого рекомендовалось наследовать все пользовательские классы. Зачем его ввели — тема для отдельной статьи. Пожалуй, наиболее существенным фактором можно назвать разделение объектной модели и модели мета-данных. Начиная с версии 3.0, старые классы больше не поддерживаются, а все пользовательские классы по умолчанию происходят от класса object.

Это породило проблему «ромбовидной структуры» («diamond diagram»).

object

/ \

A B

\ /

C

Если у нас есть классы A и B, от которых наследуется класс C, то при поиске метода по старому алгоритму (C, A, object, B) получается, что если метод не определён в классах C и A он будет извлечён из object, даже если он определён в B. Это создаёт определённые неудобства, т.к. в object определены многие магические методы, типа \_\_init\_\_, \_\_str\_\_ и т.п. Но даже если object заменить на некий пользовательский класс D, то проблема останется — менее специфичный метод класса-предка может отработать вместо более специфичного метода класса-потомка.

Итак, у нас на руках есть сам класс, список всех его предков и связей между ними. Из этих данных нам нужно построить упорядоченный список классов, в которых будет производиться поиск метода слева направо. Такой список называется линеаризацией класса. Для простоты возьмем структуру:

object

|

A

|

B

Тогда линеаризацией для класса B будет список [B, A, object]. Т.е. при вызове B().something() метод сначала будет искаться в классе B. Если он там не найден, то поиск продолжится в классе A. Если его нет и там, то поиск завершится в классе object. Перебрав все классы из линеаризации и не обнаружив нужного метода, Питон выкинет ошибку Attribute Error.

Для решения проблемы ромбовидной структуры линеаризация должна быть монотонной. Это значит, что если в линеаризации некого класса C класс A следует за классом B (она имеет вид [C, …, B, …, A]), то и для любого его потомка D класс B будет следовать за A в его линеаризации (она будет иметь вид [D, …, C, …, B, …, A]). Как вы можете видеть, старый порядок разрешения методов не монотонен, т.к. в случае ромбовидной структуры для класса A линеаризация есть [A, object], для класса B — [B, object], но для класса C — [C, A, object, B], что нарушает свойство монотонности по отношению к классу B.

Для удовлетворения свойству монотонности в этом случае подойдут две линеаризации: [C, A, B, object] и [C, B, A, object]. Очевидно, что они обе не нарушают монотонности ни в отношении класса A (т.к. object следует за A в обоих случаях), ни в отношении класса B (т.к. object следует за B в обоих случаях). Так какую из них выбрать? В этом случае наиболее интуитивно-понятный способ — посмотреть на определение класса C. Если класс объявлен как C(A, B), то разумно будет взять первую линеаризацию, поскольку в ней B следует за A. Если класс объявлен как C(B, A), то будет лучше взять вторую линеаризацию, в которой A следует за B.

Такой выбор определяется порядком локального старшинства (local precedence ordering). Свойство порядка локального старшинства требует соблюдения для классов-родителей в линеаризации класса-потомка того же порядка, что и при его объявлении. Например, если класс объявлен как D(A, B, C), то в линеаризации D класс A должен стоять раньше B, а класс B — раньше C.

Подведём промежуточные итоги:

линеаризацией класса называется список из самого класса и всех его предков (родителей и прородителей) в котором по порядку слева направо будет производиться поиск метода.

порядок разрешения методов (MRO) — это способ, с помощью которого составляется линеаризация класса.

монотонность — это свойство, которое требует соблюдения в линеаризации класса-потомка того же порядка следования классов-прородителей, что и в линеаризации класса-родителя.

порядок локального старшинства — это свойство, которое требует соблюдения в линеаризации класса-потомка того же порядка следования классов-родителей, что и в его объявлении.

Мы хотим чтобы в нашей линеаризации соблюдались и монотонность, и порядок локального старшинства.

**Алгоритм порядка разрешения методов C3.**

Для достижения указанных выше целей в Питоне используется алгоритм C3. Он достаточно прост, однако для лучшего понимания введём следующие условные обозначения:

[C1, C2, … CN] – список из элементов C1, C2, … CN. Соответственно, [С] — список из одного элемента C.

L[C] – линеаризация класса C. Важно помнить, что любая линеаризация есть список по определению.

merge(L[C1], L[C2], …, L[CN]) – объединение элементов линеаризаций L[C1], L[C2], …, L[CN] в список с помощью некоторого алгоритма. По сути, этот алгоритм должен упорядочить все классы из L[C1], L[C2], …, L[CN] и исключить дублирование классов в итоговом списке.

**Алгоритм C3 представляет из себя набор следующих правил:**

Линеаризация класса C есть одноэлементный список из самого класса C плюс объединение линеаризаций его родителей и списка всех его родителей. В условных обозначениях это можно записать как L[C] = [C] + merge(L[C1], L[C2], …, L[CN], [C1, C2, …, CN]), если класс C был объявлен как class C(C1, C2, …, CN). Надо отметить, что каждая линеаризация L[CX] начинается с класса CX, который был дополнительно приписан в конец списка объединения как непосредственный родитель класса C. Зачем это сделано станет ясно далее.

Объединение линеаризаций происходит следующим образом:

Берётся нулевой элемент из первой линеаризации (L[C1][0]).

Этот элемент ищется во всех других линеаризациях (от L[C2] до L[CN]).

Если этот элемент найден где-то вне начала списка (L[CK][X] == L[C1][0], X != 0; по сути это значит, что L[C1][0] является чьйм-то предком), то алгоритм переходит к первому шагу, беря в качестве нулевого элемент из следующей линеаризации (L[C2][0]).

Если элемент нигде не найден в позиции отличной от нулевой, то он добавляется в конец итогового списка линеаризации и удаляется из всех рассматриваемых списков (от L[C1] до L[CN]; один и тот же класс не может встречаться в итоговом списке дважды). Если после удаления элемента остались пустые списки — они исключаются из объединения. После этого алгоритм повторяется с самого начала (от нового элемента L[C1][0]), если он есть. Если его нет — объединение закончено.

Если алгоритм дошёл до последнего элемента L[CN], но ни один из нулевых элементов не удовлетворяет правилам, то линеаризация не возможна.

Чтобы алгоритм стал понятнее, разберём несколько примеров. Первым делом посмотрим, что же теперь происходит с нашей ромбовидной структурой. Для неё мы получаем:

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

L[A] = [A] + merge(L[object], [object])

L[B] = [B] + merge(L[object], [object])

L[object] = [object] (вырожденный случай)

Наибольший интерес представляет процесс объединения, так что разберём его более подробно. В случаях L[A] и L[B] выражение merge(L[object], [object]) = merge([object], [object]) раскрывается тривиально. Поскольку и первый и второй списки состоят из одного элемента object, по пункту 4 правила объединения результатом будет [object].

Теперь разберём L[C] = [C] + merge(L[A], L[B], [A, B]) = [C] + merge([A, object], [B, object], [A, B]). Распишем наше объединение по правилам C3:

Возьмем нулевой элемент первого списка — A.

Поищем его во всех прочих списках, т.е. в [B, object] и [A, B].

Поскольку класс A не обнаружен в списке [B, object], а в списке [A, B] является нулевым элементом, то мы добавляем его в итоговый список линеаризации L = [] + [A] = [A]. После этого класс A нужно удалить изо всех списков в объединении. Получим L[C] = [C] + [A] + merge([object], [B, object], [B]).

Снова возьмём нулевой элемент первого списка — object.

Поищем его во всех других списках и обнаружим, что object является первым (не нулевым) элементом списка [B, object]. Как уже было сказано выше, по сути, это означает, что класс object предок класса B. Стало быть, сначала имеет смысл поискать более специфичный метод в классе B. Поэтому алгоритм справедливо возвращается к пункту 1.

Снова возьмём нулевой элемент, но на этот раз уже из второго списка, т.е. B из списка [B, object].

Поищем его в других списках и обнаружим его только в третьем списке [B] в нулевой позиции, что нас вполне устраивает. Следовательно, добавим его в итоговый список, а затем удалим изо всех списков в объединении. Получим L = [A] + [B] = [A, B] и соответственно L[C] = [C] + [A, B] + merge([object], [object]).

Полученное объединение merge([object], [object]) уже рассматривалось выше. В итоге получим L[C] = [C] + [A, B] + [object] = [C] + [A, B, object] = [C, A, B, object]. Список [A, B, object] — это и есть результат нашего объединения merge(L[A], L[B], [A, B]).

Надеюсь что алгоритм уже стал понятнее. Однако ещё не ясно, зачем в конец объединения по правилам необходимо добавлять список всех родителей. Особо прозорливые уже могли заметить, что если бы мы поменяли в объединении местами линеаризации L[A] и L[B], т.е. написали бы merge(L[B], L[A], [A, B]), то список родителей, указанных в том же порядке, что и при инициализации класса-потомка class C(A, B) не позволил бы классу B быть обысканным раньше, чем A. И это правда. Список родителей необходим, чтобы не позволить нарушить правило порядка локального старшинства. Но давайте разберём пример class C(D, E):

object

|

D

| \

| E

| /

C

Запишем линеаризацию L[C]:

L[C] = [C] + merge(L[D], L[E], [D, E])

L[E] = [E] + merge(L[D], [D])

L[D] = [D] + merge(L[object], [object]) = [D, object]

Проведём подстановки и получим:

L[E] = [E] + merge([D, object], [D]) = [E, D, object]

L[C] = [C] + merge([D, object], [E, D, object], [D, E])

Теперь посмотрите, что мы получили. Объединение merge([D, object], [E, D, object], [D, E]) не может быть завершено, поскольку в списках [D, object] и [D, E] нулевым элементом является D, которое является первым элементом списка [E, D, object]. И наоборот, E, являющееся нулевым элементом [E, D, object] одновременно является и первым элементом [D, E]. Таким образом, через 3 итерации алгоритм придёт к пункту 5, после чего Питон выкинет ошибку TypeError: Cannot create a consistent method resolution order (MRO) for bases D, E. Если бы наше объединение не заканчивалось списком родителей, то мы бы получили нарушение порядка локального старшинства: L[C] = [C] + merge([D, object], [E, D, object]) = [C] + [E] + merge([D, object], [D, object]) = [C] + [E, D] + [object] = [C, E, D, object]. При такой линеаризации класса C поиск вёлся бы сначала в классе E, а потом в классе D, хотя в объявлении было записано C(D, E).

Решить эту проблему не сложно. Достаточно переписать объявление на class C(E, D). В таком случае мы получим:

L[C] = [C] + merge([E, D, object], [D, object], [E, D]) = [C] + [E] + merge([D, object], [D, object], [D]) = [C] + [E, D, object] = [C, E, D, object].

По сути ничего не изменилось, за исключением того, что в объявлении класса порядок перечисления родителей получился таким же, как и в линеаризации класса, т.е. соблюдается порядок локального старшинства. Надо отметить, что хотя Питон и намекает, в каком порядке логичнее было бы указать родителей, он не станет мешать вам искать приключений, если вы объявите свой собственный MRO через метакласс. Но об этом ближе к концу.

Питон вычисляет линеаризацию один раз при создании класса. Если вы хотите её получить для самопроверки или каких-то других целей — используйте свойство класса \_\_mro\_\_ (например, C.\_\_mro\_\_). Чтобы закрепить материал разберём примеры позаковыристее. Класс object умышленно выброшен, чтобы не загромождать линеаризации. Как известно из сказанного выше, при одиночном наследовании классы просто выстраиваются в цепочку от потомка к предкам, так что object всегда будет находиться в конце любой цепочки. И ещё. Я не кулинар и не музыкант, так что примеры — это всего лишь примеры. Не стоит акцентировать своё внимание на смысловых неточностях в них.

Music

/ \

Rock Gothic ------

/ \ / \

Metal Gothic Rock \

| | \

\------------------ Gothic Metal

| /

The 69 Eyes

class Music(object): pass

class Rock(Music): pass

class Gothic(Music): pass

class Metal(Rock): pass

class GothicRock(Rock, Gothic): pass

class GothicMetal(Metal, Gothic): pass

class The69Eyes(GothicRock, GothicMetal): pass

L[The69Eyes] = [The69Eyes] + merge(L[GothicRock], L[GothicMetal], [GothicRock, GothicMetal])

L[GothicRock] = [GothicRock] + merge(L[Rock], L[Gothic], [Rock, Gothic])

L[GothicMetal] = [GothicMetal] + merge(L[Metal], L[Gothic], [Metal, Gothic])

L[Rock] = [Rock, Music]

L[Gothic] = [Gothic, Music]

L[Metal] = [Metal] + [Rock, Music] = [Metal, Rock, Music]

После подстановок получаем:

L[GothicRock] = [GothicRock] + merge([Rock, Music], [Gothic, Music], [Rock, Gothic]) = [GothicRock, Rock, Gothic, Music]

L[GothicMetal] = [GothicMetal] + merge([Metal, Rock, Music], [Gothic, Music], [Metal, Gothic]) = [GothicMetal] + [Metal, Rock, Gothic, Music] = [GothicMetal, Metal, Rock, Gothic, Music]

L[The69Eyes] = [The69Eyes] + merge([GothicRock, Rock, Gothic, Music], [GothicMetal, Metal, Rock, Gothic, Music], [GothicRock, GothicMetal])

= [The69Eyes] + [GothicRock, GothicMetal] + merge([Rock, Gothic, Music], [Metal, Rock, Gothic, Music])

= [The69Eyes] + [GothicRock, GothicMetal, Metal] + merge([Rock, Gothic, Music], [Rock, Gothic, Music])

= [The69Eyes, GothicRock, GothicMetal, Metal, Rock, Gothic, Music]

Food -------

/ \ \

Meat Milk Flour

| \ \ /

Rabbit Pork Pasty

\ | /

Pie

class Food(object): pass

class Meat(Food): pass

class Milk(Food): pass

class Flour(Food): pass

class Rabbit(Meat): pass

class Pork(Meat): pass

class Pasty(Milk, Flour): pass

class Pie(Rabbit, Pork, Pasty): pass

L[Pie] = [Pie] + merge(L[Rabbit], L[Pork], L[Pasty], [Rabbit, Pork, Pasty])

L[Rabbit] = [Rabbit] + merge(L[Meat], [Meat])

L[Pork] = [Pork] + merge(L[Meat], [Meat])

L[Pasty] = [Pasty] + merge(L[Milk], L[Flour], [Milk, Flour])

L[Meat] = [Meat] + merge(L[Food], [Food]) = [Meat, Food]

L[Milk] = [Milk] + merge(L[Food], [Food]) = [Milk, Food]

L[Flour] = [Flour] + merge(L[Food], [Food]) = [Flour, Food]

После подстановок получаем:

L[Rabbit] = [Rabbit, Meat, Food]

L[Pork] = [Pork, Meat, Food]

L[Pasty] = [Pasty] + merge([Milk, Food], [Flour, Food], [Milk, Flour]) = [Pasty] + [Milk, Flour, Food] = [Pasty, Milk, Flour, Food]

L[Pie] = [Pie] + merge([Rabbit, Meat, Food], [Pork, Meat, Food], [Pasty, Milk, Flour, Food], [Rabbit, Pork, Pasty])

= [Pie] + [Rabbit] + merge([Meat, Food], [Pork, Meat, Food], [Pasty, Milk, Flour, Food], [Pork, Pasty])

= [Pie] + [Rabbit, Pork] + merge([Meat, Food], [Meat, Food], [Pasty, Milk, Flour, Food], [Pasty])

= [Pie] + [Rabbit, Pork, Meat] + merge([Food], [Food], [Pasty, Milk, Flour, Food], [Pasty])

= [Pie] + [Rabbit, Pork, Meat, Pasty] + merge([Food], [Food], [Milk, Flour, Food])

= [Pie] + [Rabbit, Pork, Meat, Pasty, Milk, Flour, Food]

= [Pie, Rabbit, Pork, Meat, Pasty, Milk, Flour, Food]

**Как обратиться к предкам.**

Множественное наследование порождает ещё одну специфическую проблему. На самом деле непосредственный поиск какого-то метода в родительских классах — это лишь часть пользы, которую можно извлечь из него. Как и в случае одиночного наследования, часто можно облегчить себе жизнь, реализовав в потомке метод, который помимо некоторых действий будет вызывать тот же самый метод родителя. Например, достаточно часто можно встретить такое:

class B(A):

def \_\_init\_\_(self):

# something

A.\_\_init\_\_(self)

Однако, для случая множественного наследования этот подход не годится. И вот по каким причинам:

class C(B, A):

def \_\_init\_\_(self):

# something

B.\_\_init\_\_(self)

A.\_\_init\_\_(self)

Во-первых, мы явно обращаемся к родительским классам (вообще-то и в примере с одиночным наследованием то же самое). Если мы захотим заменить кого-то из предков на другой класс или вообще убрать, то нам придётся изменять все функции, которые к нему обращались. Это чревато багами, если мы что-нибудь пропустим. Но это ещё пол беды. Во-вторых, мы ничего не знаем о классах A и B. Возможно, у них есть общие предки, к которым они обращаются аналогичным образом:

class A(P1, P2):

def \_\_init\_\_(self):

# something

P1.\_\_init\_\_(self)

P2.\_\_init\_\_(self)

class B(P1, P2):

def \_\_init\_\_(self):

# something

P1.\_\_init\_\_(self)

P2.\_\_init\_\_(self)

Если это так, то получится, что инициализация общих предков отработает два раза. Это не правильно. Чтобы этого избежать в Питоне есть класс super. В версии 3.0 он окончательно очеловечен и к нему можно обращаться следующим образом:

class C(B, A):

def \_\_init\_\_(self):

# something

super().\_\_init\_\_() # для версий младше 3.0 нужно использовать super(C, self)

Обратите внимание, что в версиях 2.x первым аргументом нужно указывать сам класс, а не какого-либо его родителя. По сути, объект класса super запоминает аргуметы переданные ему в момент инициализации и при вызове любого метода (super().\_\_init\_\_(self) в примере выше) проходит по списку линеаризации класса второго аргумента (self.\_\_class\_\_.\_\_mro\_\_), пытаясь вызвать этот метод по очереди для всех классов, следующих за классом в первом аргументе (класс C), передавая в качестве параметра первый аргумент (self). Т.е. для нашего случая:

self.\_\_class\_\_.\_\_mro\_\_ = [C, B, A, P1, P2, …]

super(C, self).\_\_init\_\_() => B.\_\_init\_\_(self)

super(B, self).\_\_init\_\_() => A.\_\_init\_\_(self)

super(A, self).\_\_init\_\_() => P1.\_\_init\_\_(self)

Как видите, из метода B.\_\_init\_\_ при использовании super будет вызван метод A.\_\_init\_\_, хотя класс A с ним никак не связан и не является его предком. В этом случае при необходимости по цепочке однократно отработают методы всех предков.

Благодаря такому подходу можно, например, ввести для рассмотренного примера с пирогом метод allergen для всех компонентов, который будет последовательно проходя по цепочке предков составлять список всех аллергенов компонентов, чтобы предупредить покупателей. Будет удобно вместо того, чтобы переписывать каждый раз список «не рекомендуется» для каждого продукта просто унаследовать его от составляющих компонентов. Тогда список будет формироваться автоматически. Аналогично можно будет предлагать пользователю на выбор напитки к каждому продукту, или подкручивать веса предпочтений пользователя на самописной интернет-радиостанции, идя от конкретной группы к жанрам с некоторым коэффициентом затухания.

Пример с пирогом мог бы выглядеть следующим образом (для версии 2.x):

class Food(object):

def drink(self):

return ['Water', 'Cola']

def allergen(self):

return []

class Meat(Food):

def drink(self):

return ['Red wine'] + super(Meat, self).drink()

class Milk(Food):

def allergen(self):

return ['Milk-protein'] + super(Milk, self).allergen()

class Flour(Food): pass

class Rabbit(Meat):

def drink(self):

return ['Novello wine'] + super(Rabbit, self).drink()

class Pork(Meat):

def drink(self):

return ['Sovinion wine'] + super(Pork, self).drink()

def allergen(self):

return ['Pork-protein'] + super(Pork, self).allergen()

class Pasty(Milk, Flour): pass

class Pie(Rabbit, Pork, Pasty):

def drink(self):

return ['Mineral water'] + super(Pie, self).drink()

if \_\_name\_\_ == "\_\_main\_\_":

pie = Pie()

print 'List of allergens: '

for allergen in pie.allergen(): print ' - ' + allergen

print 'List of recommended drinks: '

for drink in pie.drink(): print ' - ' + drink

В результате мы увидим следующее:

List of allergens:

— Pork-protein

— Milk-protein

List of recommended drinks:

— Mineral water

— Novello wine

— Sovinion wine

— Red wine

— Water

— Cola

Исходя из этого списка, можно понять, каким образом обходился список линеаризации. Как видите, ни один из родительских методов не был вызван дважды, иначе в списке аллергенов или рекомендаций обнаружились бы повторения. Кроме того обратите внимание, что в самом старшем классе Food определены оба метода — allergen и drink. Вызов super() не совершает никаких проверок, так что если мы пытаемся обратиться к несуществующему методу, то получим ошибку типа AttributeError: 'super' object has no attribute 'allergen'.

Когда линеаризация не может быть составлена.

Выше уже был разобран случай, когда составить линеаризацию по алгоритму C3 было невозможно. Однако, тогда проблема решилась переменой мест классов-предков в объявлении класса-потомка. Существуют и другие случай, в которых составление линеаризации не возможно:

class C(A, B): pass

class D(B, A): pass

class E(C, D): pass

L[E] = [E] + merge(L[C], L[D], [C, D]) = [E] + merge([C, A, B], [D, B, A], [C, D])

= [E] + [C, D] + merge([A, B], [B, A])

Как видите, конфликт получился неразрешимый, поскольку в объявлении C класс A стоит перед B, а в объявлении D — наоборот. По-хрошему, с этого места надо идти и пересматривать структуру. Но если вам очень надо быстро подхачить или вы просто знаете что делаете, Питон не станет вас ограничивать. Свою собственную линеаризацию можно задать через метаклассы. Для этого достаточно в метаклассе указать метод mro(cls) — по сути, переопределить метод базового метакласса type, — который вернёт нужную вам линеаризацию.

class MetaMRO(type):

def mro(cls):

return (cls, A, B, C, D object)

Дальше объявление класса различается для версии 3.0 и 2.x:

class E(C, D): \_\_metaclass\_\_ = MetaMRO # 2.x

class E(C, D, metaclass = MetaMRO): pass # 3.0

После этого E.\_\_mro\_\_ = [E, A, B, C, D, object]. Заметте, что если вы берёте на себя ответственность за MRO Питон не проводит никаких дополнительных проверок и вы можете спокойно провести поиск в предках раньше чем в потомках. И хотя это не желательная практика, но это возможно.