**Программирование метаклассов на Python, часть 2**

Автор: Мишель Симионато (Michele Simionato), физик, Питтсбургский университет

Перевод:Intersoft Lab

Метаклассы: что мы не рассмотрели в первой статье

В нашей первой статье о программировании метаклассов на Python мы представили концепцию метаклассов, показали некоторые их возможности и продемонстрировали, как их использовать при решении таких проблем, как динамическая настройка классов и библиотек во время исполнения.

Эта статья оказалась довольно популярной, но в своем сжатом заключении мы кое-что упустили. А определенные тонкости использования метаклассов заслуживают дополнительного разъяснения. Опираясь на читательский отклик и на обсуждение, развернувшееся на comp.lang.python, в этой статье мы рассмотрим некоторые более каверзные вопросы. В частности, мы считаем, что следующие моменты важны для любого программиста, желающего овладеть метаклассами:

Пользователи должны понимать различия между программированием метаклассов и традиционным объектно-ориентированным программированием и их взаимосвязь (при единичном и множественном наследовании).

В Python 2.2 появились встроенные функции staticmethod() и classmethod(), предназначенные для создания методов, которые не требуют экземпляра во время вызова. До некоторой степени методы класса совпадают по назначению с (мета)методами, определенными в метаклассах. Но отдельно взятые схожие черты и различия также породили замешательство в умах многих программистов.

Пользователям следует понимать причину конфликтов метаклассов и способы их разрешения. Это необходимо, если вы хотите использовать более одного метакласса, определенного пользователем. Мы объясним концепцию композиции (composition) метаклассов.

Воплощение (instantiation) или наследование (inheritance)

Многие программисты недопонимают различие между метаклассом и базовым классом. На поверхностном уровне "определения класса" они оба кажутся одинаковыми. Но если посмотреть глубже, концепции расходяться.

Прежде чем рассмотреть примеры, стоит определиться с терминологией. Экземпляр - это объект Python, который был "произведен" классом; класс действует как разновидность шаблона для экземпляра. Каждый экземпляр - это экземпляр только одного класса (но класс может иметь множество экземпляров). То, что мы часто называем экземпляром класса (instance object) - или, возможно, "простым экземпляром" - является "окончательным" в том смысле, что он не может выступать в качестве шаблона для других объектов (но он все равно может быть фабрикой или делегатом (delegate), которые используются для частично совпадающих целей).

Некоторые экземпляры - сами по себе классы; а все классы являются экземплярами соответствующего метакласса. Даже классы появляются исключительно посредством механизма воплощения. Обычно классы - это экземпляры встроенного, стандартного метакласса type; только когда мы задаем метаклассы, отличные от type, нам нужно думать о программировании метаклассов. Мы также называем класс, используемый для воплощения объекта, типом (type) этого объекта.

Ортогональной к идее воплощения является понятие наследования. В данном случае у класса может быть один или множество родителей, а не только уникальный тип. Родители могут иметь родителей, создавая транзитивное отношение производных классов, которое легко можно извлечь с помощью встроенной функции issubclass(). Например, если мы определим несколько классов и экземпляр:

Листинг 1. Типичная иерархия наследования

>>> class A(object): a1 = "A"

...

>>> class B(object): a2 = "B"

...

>>> class C(A,B): a3 = "C(A,B)"

...

>>> class D(C): a4 = "D(C)"

...

>>> d = D()

>>> d.a5 = "instance d of D"

Мы может протестировать это отношение:

Листинг 2. Тестирование родословной

>>> issubclass(D,C)

True

>>> issubclass(D,A)

True

>>> issubclass(A,B)

False

>>> issubclass(d,D)

[...]

TypeError: issubclass() arg 1 must be a class

А теперь интересный вопрос, необходимый для понимания различия между базовыми классами и метаклассами: как разрешается атрибут наподобие d.attr. Для простоты, рассмотрим только стандартное правило просмотра, а не "сваливание" в .\_\_getattr\_\_(). Первый шаг в таком разрешении - поискать в d.\_\_dict\_\_ имя attr. Если оно найдено - это все; но если нет, должно произойти что-то фантастическое, как, например:

>>> d.\_\_dict\_\_, d.a5, d.a1

({'a5': 'instance d'}, 'instance d', 'A')

Хитрость, позволяющая найти атрибут, который не прикреплен к экземпляру - поискать его в классе экземпляра, а после этого во всех базовых классах. Порядок, в котором просматриваются производные классы, называется порядком разрешения метода (method resolution order) для этого класса. Вы можете увидеть его с помощью (мета)метода .mro() (но только из объектов класса):

>>> [k.\_\_name\_\_ for k in d.\_\_class\_\_.mro()]

['D', 'C', 'A', 'B', 'object']

Другими словами, обращение к d.attr сначала ищет в d.\_\_dict\_\_, а затем в D.\_\_dict\_\_, C.\_\_dict\_\_, A.\_\_dict\_\_, B.\_\_dict\_\_ и в конце в object.\_\_dict\_\_. Если имя не найдено ни в одном из этих мест, возбуждается исключение AttributeError.

Заметьте, что в этой процедуре поиска метаклассы не были упомянуты ни разу.

Метаклассы или предки

Ниже приведен простой пример обычного наследования. Мы определяем базовый класс Noble с производными классами, как, например, Prince, Duke, Baron и т.д.

Листинг 3. Наследование атрибутов

>>> for s in "Power Wealth Beauty".split(): exec '%s="%s"'%(s,s)

...

>>> class Noble(object): # ...in fairy tale world

... attributes = Power, Wealth, Beauty

...

>>> class Prince(Noble):

... pass

...

>>> Prince.attributes

('Power', 'Wealth', 'Beauty')

Класс Prince наследует атрибут от Noble. Экземпляр класса Prince по-прежнему придерживается последовательности поиска, рассмотренной выше:

Листинг 4. Атрибуты в экземплярах

>>> charles=Prince()

>>> charles.attributes # ...remember, not the real world

('Power', 'Wealth', 'Beauty')

Если оказалось, что у класса Duke есть метакласс, определенный пользователем, он может получить некоторые атрибуты следующим образом:

>>> class Nobility(type): attributes = Power, Wealth, Beauty

...

>>> class Duke(object): \_\_metaclass\_\_ = Nobility

...

Кроме того, что Duke - класс, он является экземпляром метакласса Nobility - поиск атрибутов происходит как и с любым объектом:

>>> Duke.attributes

('Power', 'Wealth', 'Beauty')

Но Nobility не является базовым классом Duke, поэтому нет причин, почему экземпляр класса Duke нашел бы Nobility.attributes:

Листинг 5. Атрибуты и метаклассы

>>> Duke.mro()

[<class '\_\_main\_\_.Duke'>, <type 'object'>]

>>> earl = Duke()

>>> earl.attributes

[...]

AttributeError: 'Duke' object has no attribute 'attributes'

Доступность атрибутов метакласса не является транзитивной, другими словами, атрибуты метакласса доступны его экземплярам, но не экземплярам экземпляров. Именно это и есть главное различие между метаклассами и базовыми классами. Следующая диаграмма подчеркивает ортогональность наследования и воплощения:

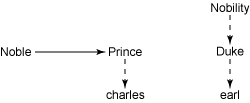


Рис. 1. Воплощение и наследование

Поскольку у earl по-прежнему есть класс, вы можете, однако, не напрямую отыскать этот атрибут:

>>> earl.\_\_class\_\_.attributes

На рисунке 1 противопоставляются простые случаи, когда используется либо наследование, либо задействованы метаклассы, но не обе концепции одновременно. Иногда, однако, у класса C есть и класс M, определенный пользователем, и базовый класс B:

Листинг 6. Комбинирование базового класса и метакласса

>>> class M(type):

... a = 'M.a'

... x = 'M.x'

...

>>> class B(object): a = 'B.a'

...

>>> class C(B): \_\_metaclass\_\_=M

...

>>> c=C()

Графически:

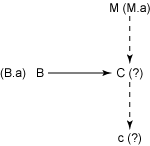


Рис. 2. Комбинированные базовый класс и метакласс

Согласно предшествующему объяснению, мы могли бы представить, что C.a разрешился бы либо в M.a, либо в B.a. Оказывается, поиск по классу следует его порядку разрешения метода до того, как он осуществляется в его метаклассе:

Листинг 7. Разрешение метаклассов и базовых классов

>>> C.a, C.x

('B.a', 'M.x')

>>> c.a

'B.a'

>>> c.x

[...]

AttributeError: 'C' object has no attribute 'x'

Вы все же можете задать величину атрибута, используя метакласс - вам просто нужно указать ее для воплощаемого объекта класса, а не в качестве атрибута метакласса.

Листинг 8. Задание атрибута в метаклассе

>>> class M(type):

... def \_\_init\_\_(cls, \*args):

... cls.a = 'M.a'

...

>>> class C(B): \_\_metaclass\_\_=M

...

>>> C.a, C().a

('M.a', 'M.a')

**Еще о магии классов**

То, что ограничение воплощения слабее ограничения наследования, существенно для разработки специальных методов, как .\_\_new\_\_(), .\_\_init\_\_(), .\_\_str\_\_() и т.д. Рассмотрим метод .\_\_str\_\_(), анализ для других специальных методов -проводится аналогично.

Читатели, вероятно, знают, что печатаемое представление объекта класса можно модифицировать, подменив его метод .\_\_str\_\_(). В том же смысле печатаемое представление класса можно модифицировать, подменив метод .\_\_str\_\_() его метакласса. Например:

Листинг 9. Настройка вывода класса на печатающее устройство

>>> class Printable(type):

... def \_\_str\_\_(cls):

... return "This is class %s" % cls.\_\_name\_\_

...

>>> class C(object): \_\_metaclass\_\_ = Printable

...

>>> print C # equivalent to print Printable.\_\_str\_\_(C)

This is class C

>>> c = C()

>>> print c # equivalent to print C.\_\_str\_\_(c)

<C object at 0x40380a6c>

Эту ситуацию можно представить с помощью следующей диаграммы:

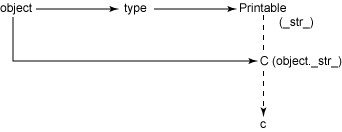


Рис. 3. Метаклассы и магические методы

Из предыдущего обсуждения ясно, что метод .\_\_str\_\_() в Printable не может заменить метод .\_\_str\_\_() в C, который наследуется из object и, следовательно, обладает приоритетом; печать C по-прежнему дает стандартный результат.

Если бы C наследовал свой метод .\_\_str\_\_() из Printable, а не из object, это породило бы проблему: у экземпляров C нет атрибута .\_\_name\_\_, и печать C сгенерировала бы ошибку. Разумеется, вы по-прежнему могли бы определить метод .\_\_str\_\_() в C, что изменило бы то, как печатается C.

Методы класса в сравнении с метаметодами

Другая путаница происходит между методами класса Python и методами, определенными в метаклассе, которые лучше называть метаметодами.

Рассмотрим пример:

Листинг 10. Метаметоды и методы класса

>>> class M(Printable):

... def mm(cls):

... return "I am a metamethod of %s" % cls.\_\_name\_\_

...

>>> class C(object):

... \_\_metaclass\_\_=M

... def cm(cls):

... return "I am a classmethod of %s" % cls.\_\_name\_\_

... cm=classmethod(cm)

...

>>> c=C()

Частично эта путаница вызвана тем, что C.mm в терминологии Smalltalk назывался бы "методом класса C". Однако методы класса Python - нечто совсем иное.

Метаметод "mm" может быть вызван либо из метакласса, либо из класса, но не из экземпляра. Метод класса может быть вызван и из класса, и из его экземпляров (но не существует в метаклассе).

Листинг 11. Вызов метаметода

>>> print M.mm(C)

I am a metamethod of C

>>> print C.mm()

I am a metamethod of C

>>> print c.mm()

[...]

AttributeError: 'C' object has no attribute 'mm'

>>> print C.cm()

I am a classmethod of C

>>> print c.cm()

I am a classmethod of C

Кроме того, метаметод извлекается dir(M), а не dir(C), в то время как метод класса извлекается dir(C) и dir(c).

Вы можете вызывать только методы метакласса, которые определены в порядке разрешения метода класса, выполнив диспетчеризацию по метаклассу (встроенные функции, как print, делают это неявно):

Листинг 12. Магический метод метакласса

>>> print C.\_\_str\_\_()

[...]

TypeError: descriptor '\_\_str\_\_' of 'object' object needs an argument

>>> print M.\_\_str\_\_(C)

This is class C

Важно заметить, что этот конфликт диспетчеризации не ограничен магическими методами. Если мы изменим C, добавив атрибут C.mm, возникнет та же проблема (не имеет значения, является ли имя регулярным методом, методом класса, статическим методом или простым атрибутом):

Листинг 13. Немагический метод метакласса

>>> C.mm=lambda self: "I am a regular method of %s" % self.\_\_class\_\_

>>> print C.mm()

[...]

TypeError: unbound method <lambda>() must be called with

C instance as first argument (got nothing instead)

**Конфликты метаклассов**

Стоит вам всерьез поработать с метаклассами, и вы хотя бы раз столкнетесь с конфликтом метаклассов/метатипов. Рассмотрим класс A с метаклассом M\_A и класс B с метаклассом M\_B; предположим, что мы производим C от A и B. Возникает вопрос: что является метаклассом C? M\_A или M\_B?

Правильный ответ - M\_C, где M\_C - это метакласс, который наследуется от M\_A и M\_B, как показано на следующей диаграмме (см. ниже в разделе Ресурсы ссылку на книгу "Использование метаклассов" (Putting metaclasses to work)):

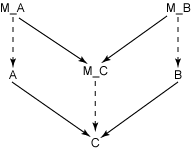


Рис. 4. Предотвращение конфликта метаклассов

Однако, Python автоматически не создает (пока) M\_C. Вместо этого он возбуждает исключение TypeError, предупреждая программиста о конфликте:

Листинг 14. Конфликты метаклассов

>>> class M\_A(type): pass

...

>>> class M\_B(type): pass

...

>>> class A(object): \_\_metaclass\_\_ = M\_A

...

>>> class B(object): \_\_metaclass\_\_ = M\_B

...

>>> class C(A,B): pass # Error message less specific under 2.2

[...]

TypeError: metaclass conflict: the metaclass of a derived class must

be a (non-strict) subclass of the metaclasses of all its bases

Конфликта метаклассов можно избежать, вручную создав необходимый метакласс для C:

Листинг 15. Разрешение конфликта метаклассов вручную

>>> M\_AM\_B = type("M\_AM\_B", (M\_A,M\_B), {})

>>> class C(A,B): \_\_metaclass\_\_ = M\_AM\_B

...

>>> type(C)

<class 'M\_AM\_B'>

Разрешение конфликтов метаклассов становится более сложным, если вы желаете "вставить" дополнительные метаклассы в класс вслед за используемыми его предками. Кроме того, в зависимости от метаклассов родительских классов могут появиться избыточные метаклассы - и идентичные метаклассы в различные предках, и отношения базовый класс/производный класс среди метаклассов. Модуль noconflict предоставляет пользователям автоматическое и надежное решение этих проблем (см. Ресурсы).

**Заключение**

В этой статье рассматривается ряд предостережений и непростых ситуаций. Для работы с метаклассами требуется определенное количество проб и ошибок, прежде чем их поведение станет интуитивным. Однако, эти вопросы никоим образом не безнадежны - эта довольно краткая статья затрагивает большинство подводных камней. Поэкспериментируйте с этими ситуациями сами. И к концу дня вы обнаружите, что метаклассы предоставляют новую степень программного обобщения; выигрыш сторицей окупает небольшой риск.