**Программирование метаклассов на Python**

Дэвид Мертц, Мишель Симионато

Перевод:Intersoft Lab

Поднятие объектно-ориентированного программирования на новый уровень

Большинство читателей уже знакомо с концепциями объектно-ориентированного программирования: наследованием, инкапсуляцией, полиморфизмом. Но создание объектов заданного класса с определенными родителями обычно представляется исходно заданной операцией. Оказывается, что ряд новых программных конструкций становится либо более простым, либо вообще хоть сколько-нибудь возможным, если вы можете настраивать процесс создания объектов. Метаклассы разрешают определенные типы "аспектно-ориентированного программирования"; например, можно усилить классы такими характеристиками, как возможности трассировки, объектная персистентность, регистрация исключений и так далее.

Краткий обзор объектно-ориентированного программирования (ООП)

Давайте, потратив полминуты, вспомним, что такое ООП. В языке объектно-ориентированного программирования можно определять классы, задача которых - объединить связанные данные и поведение. Эти классы могут наследовать некоторые или все свойства своих родителей, они также определяют свои собственные атрибуты (данные) или методы (поведение). В результате, классы, как правило, выступают в качестве шаблонов для создания экземпляров (которые время от времени называют просто объектами). Различные экземпляры одного и того же класса обычно имеют разные данные, хотя они будут представлены в одинаковом виде, например, у обоих объектов класса 'Employee' bob и jane есть .salary и .room\_number, но значения room (комната) и salary (жалование) у каждого различны.

Некоторые объектно-ориентированные языки программирования, включая Python, предусматривают интроспективные (или рефлексивные) объекты. Другими словами, интроспективный объект может сам себя описывать: к какому классу принадлежит этот экземпляр? Кто предки этого класса? Какие методы и атрибуты доступны объекту? Интроспекция позволяет функции или методу, управляющему объектами, принимать решения, основываясь на том, какой вид объекта передается. Даже без интроспекции функции часто ветвятся, опираясь на данные экземпляра - например, маршрут к jane.room\_number отличается от пути к bob.room\_number, поскольку они в "различных комнатах" (значения room у них различны). С помощью интроспекции также можно безошибочно вычислить bonus (премиальные) jane, пропустив это вычисление для bob, например, потому что у jane есть атрибут .profit\_share или из-за того, что bob является экземпляром производного класса Hourly(Employee).

"Метапрограммный" ответ

Базовая система ООП, очерченная выше, является достаточно мощной. Однако, в этом описании один момент не получил должного внимания: в Python (и других языках) сами классы являются объектами, которые можно передавать и подвергать интроспекции. Но поскольку объекты, как отмечалось, создаются с использованием классов в качестве шаблонов, то что же является шаблоном для создания классов? Разумеется, метаклассы.

В Python всегда были метаклассы. Однако, технология, задействованная в метаклассах, стала гораздо более очевидной с выходом Python 2.2. А именно, в версии 2.2 Python перестал быть языком только с одним специальным (обычно невидимым) метаклассом, который создавал каждый объект класса. Теперь программисты могут наследоваться от встроенного метакласса type и даже динамически генерировать классы с различными метаклассами. Разумеется, только то, что вы можете манипулировать метаклассами на Python 2.2, еще не объясняет, зачем вам это.

Более того, вам не нужно использовать метаклассы, определенные пользователем, чтобы управлять созданием классов. Несколько менее головоломная концепция - фабрика классов (class factory): обыкновенная функция может возвращать класс, который был динамически создан в пределах тела функции. В традиционном синтаксисе Python вы можете написать:

Листинг 1. Традиционная фабрика классов на Python 1.5.2

Python 1.5.2 (#0, Jun 27 1999, 11:23:01) [...]

Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam

>>> def class\_with\_method(func):

... class klass: pass

... setattr(klass, func.\_\_name\_\_, func)

... return klass

...

>>> def say\_foo(self): print 'foo'

...

>>> Foo = class\_with\_method(say\_foo)

>>> foo = Foo()

>>> foo.say\_foo()

foo

Функция фабрики class\_with\_method() динамически создает и возвращает класс, содержащий метод/функцию, передаваемую в эту фабрику. Сам класс обрабатывается в пределах тела функции до того, как он возвращен. Модуль new обеспечивает более лаконичное выражение, но без возможности определения пользователем дополнительного кода в пределах тела фабрики классов. Например:

Листинг 2. Фабрика классов в модуле new

>>> from new import classobj

>>> Foo2 = classobj('Foo2',(Foo,),{'bar':lambda self:'bar'})

>>> Foo2().bar()

'bar'

>>> Foo2().say\_foo()

foo

Во всех этих случаях поведение класса (Foo, Foo2) не записано непосредственно в виде кода, а создается посредством вызова во время исполнения функций с вычисляемыми аргументами. Следует подчеркнуть, что динамически создаются именно сами классы, а не просто экземпляры.

**Метаклассы: решение, требующее проблемы?**

"Метаклассы - большая магия, чем нужно 99% пользователей. Если вы задаетесь вопросом, нужны ли они вам, значит, они вам не нужны (те, кому они действительно нужны, точно это знают, и им не требуется объяснение, зачем)".

Тим Питерс (Tim Peters), крупнейший авторитет в области Python

Методы (классов), как и обычные функции, могут возвращать объекты. В этом смысле очевидно, что фабрики классов столь же могут быть классами, как и функциями. В частности, Python 2.2+ предоставляет специальный класс, называемый type, который именно и есть такая фабрика классов. Разумеется, читатели узнают в type() менее претенциозную встроенную функцию более ранних версий Python - к счастью, поведение старой функции type() поддерживается классом type (другими словами, type(obj) возвращает тип/класс объекта obj). Новый класс работает в качестве фабрики классов точно так же, как прежде делала функция new.classobj:

Листинг 3. type в качестве метакласса фабрики классов

>>> X = type('X',(),{'foo':lambda self:'foo'})

>>> X, X().foo()

(<class '\_\_main\_\_.X'>, 'foo')

Но поскольку теперь type - это (мета)класс, вы можете создать от него производный класс:

Листинг 4. Потомок type как фабрика классов

>>> class ChattyType(type):

... def \_\_new\_\_(cls, name, bases, dct):

... print "Allocating memory for class", name

... return type.\_\_new\_\_(cls, name, bases, dct)

... def \_\_init\_\_(cls, name, bases, dct):

... print "Init'ing (configuring) class", name

... super(ChattyType, cls).\_\_init\_\_(name, bases, dct)

...

>>> X = ChattyType('X',(),{'foo':lambda self:'foo'})

Allocating memory for class X

Init'ing (configuring) class X

>>> X, X().foo()

(<class '\_\_main\_\_.X'>, 'foo')

Магические методы .\_\_new\_\_() и .\_\_init\_\_() являются специальными, но концептуально они такие же, как и у любого другого класса. Метод .\_\_init\_\_() позволяет конфигурировать созданный объект; метод .\_\_new\_\_() разрешает настраивать его создание. Последний, разумеется, не используется широко, но существует для каждого класса нового стиля Python 2.2 (обычно наследуется, а не подменяется).

У потомков type есть одно свойство, которое необходимо учитывать; на нем ловятся все, кто впервые использует метаклассы. Первый аргумент в методах обычно называется cls, а не self, поскольку эти методы обрабатывают созданный класс, а не метакласс. На самом деле, в этом нет ничего особенного; все методы связываются со своимиэкземплярами, а экземпляр метакласса является классом. Неспециальное имя делает это более очевидным[1]:

Листинг 5. Прикрепление методов класса к созданным классам

>>> class Printable(type):

... def whoami(cls): print "I am a", cls.\_\_name\_\_

...

>>> Foo = Printable('Foo',(),{})

>>> Foo.whoami()

I am a Foo

>>> Printable.whoami()

Traceback (most recent call last):

TypeError: unbound method whoami() [...]

Вся эта удивительно непримечательная технология сопровождается некими синтаксическими украшениями, упрощающими работу с метаклассами и одновременно запутывающими новых пользователей. В этом дополнительном синтаксисе есть несколько элементов. Порядок интерпретации этих новых вариаций мудреный. Классы могут наследовать метаклассы от своих предков - заметьте, что это не одно и то же, что наличие метаклассов в качестве предков (еще одно обычное заблуждение). Для классов старого стиля определение глобальной переменной \_\_metaclass\_\_ приводит к использованию метаклассса, определенного пользователем. Однако по большей части самый безопасный подход - задать атрибут класса \_\_metaclass\_\_ для класса, который хочет быть созданным по метаклассу, определенному пользователем. Вы должны задать эту переменную в самом определении класса, поскольку метакласс не используется, если этот атрибут задан позднее (после того как объект класса уже был создан). Например:

Листинг 6. Задание метакласса с атрибутом класса

>>> class Bar:

... \_\_metaclass\_\_ = Printable

... def foomethod(self): print 'foo'

...

>>> Bar.whoami()

I am a Bar

>>> Bar().foomethod()

foo

**Проблемы, решаемые магией**

До сих пор мы рассматривали основы метаклассов. Однако реальное применение метаклассов имеет свои тонкости. Сложность с использованием метаклассов заключается в том, что в типичной модели ООП классы в действительности делают не много. Структура наследования классов удобна для инкапсуляции и объединения данных и методов, но обычно реально используются именно экземпляры.

Существует две общие категории задач программирования, для которых, на наш взгляд, метаклассы являются чрезвычайно полезными.

Первый, и вероятно более общий случай, это когда в период проектирования вы не знаете точно, что нужно делать классу. Очевидно, что у вас появится представление об этом, но некая определенная деталь может зависеть от информации, которая станет доступной позднее. Само "позднее" может быть двух видов: (а) когда библиотечный модуль будет использоваться приложением; (б) во время исполнения, когда будет существовать некая ситуация. Этот категория близка к тому, что часто называют "Аспектно-ориентированным программированием" (АОП). Продемонстрируем, что мы имеем в виду на следующем элегантном примере:

Листинг 7. Конфигурирование метакласса во время исполнения

% cat dump.py

#!/usr/bin/python

import sys

if len(sys.argv) > 2:

module, metaklass = sys.argv[1:3]

m = \_\_import\_\_(module, globals(), locals(), [metaklass])

\_\_metaclass\_\_ = getattr(m, metaklass)

class Data:

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

self.num = 38

self.lst = ['a','b','c']

self.str = 'spam'

dumps = lambda self: `self`

\_\_str\_\_ = lambda self: self.dumps()

data = Data()

print data

% dump.py

<\_\_main\_\_.Data instance at 1686a0>

Как вы могли ожидать, это приложение выводит весьма общее описание объекта data (условный объект экземпляра). Однако, если аргументы времени исполнения передаются в приложение, можно получить несколько отличный результат:

Листинг 8. Добавление метакласса внешней сериализации

% dump.py gnosis.magic MetaXMLPickler

<?xml version="1.0"?>

<!DOCTYPE PyObject SYSTEM "PyObjects.dtd">

<PyObject module="\_\_main\_\_" class="Data" id="720748">

<attr name="lst" type="list" id="980012" >

<item type="string" value="a" />

<item type="string" value="b" />

<item type="string" value="c" />

</attr>

<attr name="num" type="numeric" value="38" />

<attr name="str" type="string" value="spam" />

</PyObject>

В этом частном примере применяется стиль сериализации gnosis.xml.pickle, но текущая версия пакета gnosis.magic также содержит метаклассы сериализаторов MetaYamlDump, MetaPyPickler и MetaPrettyPrint. Кроме того, пользователь "приложения" dump.py может потребовать использование любого желаемого "MetaPickler" из любого пакета Python, который его определяет. Соответствующий метакласс, предназначенный для этой цели, будет выглядеть приблизительно так:

Листинг 9. Добавление атрибута с метаклассом

class MetaPickler(type):

"Metaclass for gnosis.xml.pickle serialization"

def \_\_init\_\_(cls, name, bases, dict):

from gnosis.xml.pickle import dumps

super(MetaPickler, cls).\_\_init\_\_(name, bases, dict)

setattr(cls, 'dumps', dumps)

Замечательное достижение этого подхода заключается в том, что прикладному программисту не нужно ничего знать о том, какая сериализация будет использоваться -будет ли даже добавляться в командной строке сериализация или другой способ внешнего представления.

Возможно, наиболее общее использование метаклассов схоже с применением MetaPicklers: добавление, удаление, переименование или подстановка методов вместо методов, определенных в созданном классе. В нашем примере "встроенный" метод Data.dump() заменяется другим методом, внешним по отношению к приложению, во время создания класса Data (и, следовательно, в каждом последующем экземпляре).

Другие способы решения проблем с помощью магии

Существует область программирования, где классы зачастую более важны, чем экземпляры. Например, декларативные мини-языки (declarative mini-languages) - это библиотеки Python, программная логика которых выражена непосредственно в объявлении класса. Дэвид рассматривает их в своей статье "Создание декларативных мини-языков" (Create declarative mini-languages). В подобных случаях использование метаклассов для воздействия на процесс создания класса может быть весьма эффективным.

Одной из декларативных библиотек, основанных на классах, является gnosis.xml.validity. В этой структуре вы объявляете ряд "классов допустимости" ("validity classes"), которые описывают набор ограничений для допустимых документов XML. Эти объявления очень близки к тем, что содержатся в описаниях типа документа (DTDs). Например, документ "диссертация" может быть сконфигурирован с помощью следующего кода:

Листинг 10. Правила gnosis.xml.validity в simple\_diss.py

from gnosis.xml.validity import \*

class figure(EMPTY): pass

class \_mixedpara(Or): \_disjoins = (PCDATA, figure)

class paragraph(Some): \_type = \_mixedpara

class title(PCDATA): pass

class \_paras(Some): \_type = paragraph

class chapter(Seq): \_order = (title, \_paras)

class dissertation(Some): \_type = chapter

Если попытаться создать экземпляр dissertation без надлежащих подэлементов, возбуждается исключение, описывающее ситуацию; подобное имеет место для каждого подэлемента. Правильные подэлементы будут автоматически сгенерированы из более простых аргументов, если существует только один непротиворечивый способ "достроить" тип до корректного состояния.

Хотя классы допустимости часто (неформально) базируются на предварительно существующем DTD, экземпляры этих классов печатаются как внеконтекстные (unadorned) фрагменты документа XML, например:

Листинг 11. Создание документа с базовым классом допустимости

>>> from simple\_diss import \*

>>> ch = LiftSeq(chapter, ('It Starts','When it began'))

>>> print ch

<chapter><title>It Starts</title>

<paragraph>When it began</paragraph>

</chapter>

Используя метакласс для создания классов допустимости, мы можем генерировать DTD из самих объявлений класса (и при этом добавить дополнительный метод в эти классы):

Листинг 12. Использование метаклассов во время импорта модуля

>>> from gnosis.magic import DTDGenerator, \

... import\_with\_metaclass, \

... from\_import

>>> d = import\_with\_metaclass('simple\_diss',DTDGenerator)

>>> from\_import(d,'\*\*')

>>> ch = LiftSeq(chapter, ('It Starts','When it began'))

>>> print ch.with\_internal\_subset()

<?xml version='1.0'?>

<!DOCTYPE chapter [

<!ELEMENT figure EMPTY>

<!ELEMENT dissertation (chapter)+>

<!ELEMENT chapter (title,paragraph+)>

<!ELEMENT title (#PCDATA)>

<!ELEMENT paragraph ((#PCDATA|figure))+>

]>

<chapter><title>It Starts</title>

<paragraph>When it began</paragraph>

</chapter>

Пакету gnosis.xml.validity ничего неизвестно о DTD и внутренних подмножествах. Эти концепции и возможности всецело представлены метаклассом DTDGenerator без внесения каких-либо изменений в gnosis.xml.validity или simple\_diss.py. DTDGenerator не подставляет свой собственный метод .\_\_str\_\_() в классы, которые он создает - вы по-прежнему можете вывести внеконтекстный фрагмент XML - но метакласс мог бы легко модифицировать подобные магические методы.

**Метапреимущества**

Пакет gnosis.magic содержит несколько утилит для работы с метаклассами, а также некоторые примеры метаклассов, которые можно применять в аспектно-ориентированном программировании. Наиболее важная из этих утилит - import\_with\_metaclass(). Эта функция, задействованная в предыдущем примере, позволяет импортировать произвольный модуль, создавая все классы этого модуля с использованием метакласса, определенного пользователем, а не type. Какую бы новую возможность вы ни захотели задать в этом модуле, она может быть определена в метаклассе, который вы создаете (или получаете). gnosis.magic содержит некоторые подключаемые метаклассы сериализации; другой пакет мог бы включать возможности трассировки, объектную персистентность, регистрацию исключений или же что-нибудь еще.

Функция import\_with\_metaclass() иллюстрирует некоторые возможности программирования метаклассов:

Листинг 13. Функция import\_with\_metaclass() из [gnosis.magic]

def import\_with\_metaclass(modname, metaklass):

"Module importer substituting custom metaclass"

class Meta(object): \_\_metaclass\_\_ = metaklass

dct = {'\_\_module\_\_' : modname}

mod = \_\_import\_\_(modname)

for key, val in mod.\_\_dict\_\_.items():

if inspect.isclass(val):

setattr(mod, key, type(key, (val, Meta), dct))

return mod

В этой функции стоит обратить внимание на стиль - обыкновенный класс Meta создан с использованием заданного метакласса. Но как только Meta добавлен в качестве предка, его потомки также создаются с помощью этого метакласса. В принципе, класс, подобный Meta, мог бы предоставлять и генератор метакласса, и ряд наследуемых методов - эти два аспекта наследования являются ортогональными.