**Метаклассы в python 2.X с примерами и полным разоблачением**

**Теория, часть 1. Метаклассы**

Все начинается с объявления класса:

Hightlited/Raw

class A(object):

field = 12

def method(self, param):

return param + self.field

Имеющие опыт программирования на компилируемых языках могут увидеть здесь декларативную конструкцию, но это только обман зрения. В python всего две декларативные конструкции - объявление кодировки файла и импорт синтаксических конструкций "из будущего". Все остальное - исполняемое. Написанное объявление это синтаксический сахар для следующего:

Hightlited/Raw

txt\_code = """

field = 12

def method(self, param):

return param + self.field

"""

class\_body = {}

compiled\_code = compile(txt\_code, \_\_file\_\_, "exec")

eval(compiled\_code, globals(), class\_body)

A = type("A", (object,), class\_body)

Оба этих способа создать класс A совершенно эквивалентны. Окончательно убедиться в том, что объявление класса исполнимо можно посмотрев вот на это:

Hightlited/Raw

def this\_is\_not\_cplusplus(some\_base\_class, num\_methods):

class Result(some\_base\_class):

x = some\_base\_class()

for pos in range(num\_methods):

# добавим функцию в locals - она попадает в тело

# класса Result и станет его методом

locals()['method\_' + str(pos)] = \

lambda self, m : m + num\_methods

return Result

class\_with\_10\_methods = this\_is\_not\_cplusplus(object, 10)

class\_with\_20\_methods = this\_is\_not\_cplusplus(

class\_with\_10\_methods, 20)

print class\_with\_10\_methods().method\_3(2) # напечатает 12

print class\_with\_20\_methods().method\_13(2) # напечатает 22

Функция this\_is\_not\_cplusplus создает новый класс каждый раз, когда мы ее вызываем, используя переданный тип в качестве базового и создавая в новом классе такое количество методов, какое мы запросили.

Но вернемся ко второму блоку кода. В нем нас особенно интересует последняя строка:

Hightlited/Raw

A = type("A", (object,), class\_body)

Она похожа на инстанцирование типа type, и это на самом деле так. Т.е. класс A это экзампляр типа type. В python есть некая супер иерархия типов - снизу находятся экземпляры обычных классов, потом обычные классы (унаследованные от object или он ничего) и на самом верху type (который, кстати, экземпляр самого себя) и все, что от него унаследовано - метаклассы. Инстанцируя метаклассы мы получаем обычные классы - метаклассы являются типами классов.

type -----------> MetaB --> MetaC

| | |

object --> A object ---> B ------> C

| | |

a b c

Здесь вертикальная черта - инстанцирование, а горизинтальная стрелка - наследование.

Как и объекты, классы хранят свои метаклассы в атрибуте \_\_class\_\_. Так же, как классы управляют жизненным циклом объектов и их поведением, метаклассы управляют жизненным циклом и поведением классов.

Начнем с простого:

Hightlited/Raw

class MyMeta(type):

pass

MyMeta простейший метакласс. Если в теле класса присвоить его полю \_\_metaclass\_\_ то он будет использован для конструирования класса и станет его типом:

Hightlited/Raw

class A(object):

pass

class B(A):

\_\_metaclass\_\_ = MyMeta

class C(B):

pass

print "type(A) =", type(A)

# напечатает "type(A) = <type 'type'>"

print "type(B) =", type(B)

# напечатает "type(B) = <class '\_\_main\_\_.MyMeta'>"

print "type(C) =", type(C)

# напечатает "type(C) = <class '\_\_main\_\_.MyMeta'>"

В итоге B и C имеют тип MyMeta. Т.е. метаклассы наследуются. Я напомню, что инстанцирование класса в питоне это тоже синтаксический сахар:

Hightlited/Raw

c = SomeClass(1, x=12)

=>

Hightlited/Raw

c = SomeClass.\_\_new\_\_(SomeClass, 1, x=12)

SomeClass.\_\_init\_\_(c, 1, x=12)

\_\_new\_\_ создает новый объект класса, а \_\_init\_\_ инициализирует его. Здесь полная аналогия с С++ методами new и конструктором (чаще всего \_\_new\_\_ наследуется от object). Рассмотрим эти методы на примере:

Hightlited/Raw

class MyMeta2(type):

def \_\_new\_\_(cls, name, bases, class\_dict):

print "\_\_new\_\_({0}, {1}, {2}, {3})".format(

cls, name, bases, class\_dict)

return super(MyMeta2, cls).\_\_new\_\_(cls,

name,

bases,

class\_dict)

def \_\_init\_\_(self, name, bases, class\_dict):

print "\_\_init\_\_({0}, {1}, {2}, {3})".format(

cls, name, bases, class\_dict)

return super(MyMeta2, self).\_\_init\_\_(self,

name,

bases,

class\_dict)

class D(object):

\_\_metclass\_\_ = MyMeta2

x = 12

Эта конструкция напечатает ожидаемые строки:

\_\_new\_\_(<class '\_\_main\_\_.MyMeta2'>, D, (<type 'object'>,),

{'x': 12, '\_\_module\_\_': '\_\_main\_\_',

'\_\_metaclass\_\_': <class '\_\_main\_\_.MyMeta2'>})

\_\_init\_\_(<class '\_\_main\_\_.D'>, D, (<type 'object'>,),

{'x': 12, '\_\_module\_\_': '\_\_main\_\_',

'\_\_metaclass\_\_': <class '\_\_main\_\_.MyMeta2'>})

**Практика**

Что мы получили по итогу - с помощью метаклассов можно:

Изменять типы создаваемых классов

Автоматически вызывать некоторый код при каждом прямом или непрямом наследовании данного класса

Менять параметры нового класса - метакласс может подменить в методе \_\_new\_\_ переменные name, bases, class\_dict до их передачи в type.\_\_new\_\_ или повлиять на полученный класс.

Альтернативные методы некоторых возможностей метаклассов реализуют декораторы, но:

декораторы классов исполняются уже после создания класса, и некоторые модификации класса в них более громоздки (аналогично методу \_\_init\_\_ метаклассов). Также метакласс может влиять на список базовых классов

декораторы функций требуют применения к каждому методу в отдельности, в то время как метаклассы позволяют применять функциональность ко всем методам объекта сразу.

Чаще всего в метаклассе перегружается метод \_\_new\_\_ , позволяющий менять параметры класса до создания. Читая все ниже написаное нужно помнить, что достичь того-же эффекта часто можно и без метаклассов, но придется писать больше однообразного кода. Метаклассы, как и многие возможности python, позволяют писать удобные для использования библиотеки, но менее полезны для написания конечного продукта.

Итак какие практические результаты мы можем извлечь из этого метасчастья? Разобьем примеры использования по списку возможностей метаклассов:

**Изменять типы создаваемых классов**

Большое количество синтаксических конструкций python преобразуются в вызовы специальных методов класса используемого объекта, с передачей объекта параметром в метод, например:

Hightlited/Raw

a + b # => a.\_\_class\_\_.\_\_add\_\_(a, b)

a.c # => a.\_\_class\_\_.\_\_getattr\_\_(a, 'c')

# кроме этого a.\_\_class\_\_.\_\_dict\_\_ будет использован для

# поиска атрибута 'c', если он не будет найден

# в a.\_\_dict\_\_. Вообще говоря поиск атрибута в объекте длинная

# история - о ней будет своя статья

str(a) # => a.\_\_str\_\_(a)

iter(a) # => a.iter(a)

# и т.д.

Таким образом мы можем перегрузить операторы для класса, перегрузив соответствующие методы в его метаклассе. Например наследование при помощи сложения:

Hightlited/Raw

class MixinMeta(type):

def \_\_add\_\_(self, other\_type):

new\_name = "Mixin\_{0}\_{1}".format(self.\_\_name\_\_,

other\_type.\_\_name\_\_)

# явное построение нового класса, наследующего текущий и

# 'other\_type'

return self.\_\_class\_\_(new\_name,

(self, other\_type),

{})

class Mixin(object):

\_\_metaclass\_\_ = MixinMeta

class A(Mixin):

x = 'A.x'

y = 'A.y'

class B(object):

x = 'B.x'

z = 'B.z'

print (A + B).y # A.y

print (A + B).x # A.x

print (A + B).z # B.z

# можно и так

class C(object):

pass

tp = Mixin + C + B

MixinMeta позволяет создавать дочерний класс складывая базовые - вместо class C(A,B):pass писать A+B. Точно так же можно, например, облагородить 'str(A)' или хранить в классе список ссылок на все его экземпляры и итерировать по ним (этот пример идет в конце, поскольку использует все возможности метаклассов). По поводу доступа к атрибутам нужно помнить, что атрибуты и методы метаклассов доступны через его классы однако не через экземпляры этих классов.

Hightlited/Raw

class M(type):

X = 1

class B(object):

\_\_metaclass\_\_ = M

b = B()

print M.X # 1

print B.X # 1

print b.X # => AtrributeError

**Поиск атрибута производится по объекту и его типу, но не по типу его типа.**

Автоматически вызывать некоторый код при каждом прямом или непрямом наследовании данного класса

Это позволяет вести реестр всех классов, унаследовавших данный интерфейс. Удобно для написания плагинов и других расширяемых архитектур.

Hightlited/Raw

# в файле plugin\_api.py

plugin\_registry = {}

def get\_registry():

class RegMeta(type):

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

self.interface\_name = None

def \_\_new\_\_(cls, name, bases, cls\_dict):

new\_cls = super(RegMeta, cls).\_\_new\_\_(name,

bases,

cls\_dict)

# первый раз этот вызов произойдет из тела интерфейса

if self.interface\_name is None:

self.interface\_name = name

else:

plugin\_registry.setdefault(self.interface\_name, []).append(new\_cls)

return new\_cls

return RegMeta

def get\_all\_implementations(interface):

return plugin\_registry[interface.\_\_name\_\_]

class IDataGridUI(object):

\_\_metaclass\_\_ = get\_registry()

provides\_ui = None

def display\_my\_data(self, data):

pass

# в файле data\_display\_html\_qt.py

from plugin\_api import IDataGridUI

class QtDataDisplayer(IDataGridUI):

provides\_ui = 'qt'

def display\_my\_data(self, data):

#some qt code

pass

class HTMLDataDisplayer(IDataGridUI):

provides\_ui = 'html'

def display\_my\_data(self, data):

#some template code

pass

# в файле build\_ui.py

from plugin\_api import get\_all\_implementations

print get\_all\_implementations(IDataGridUI)

# [..., QtDataDisplayer, HTMLDataDisplayer, ....]

Нужно учесть что для того, чтобы реализация попала в реестр файл реализации должен быть импортирован где нибудь до вызова get\_all\_implementations. Имея реестр всех реализаций можно, например, автоматически расширять API программы (Rpc/REST/командная строка).

**Изменять параметры создаваемого дочернего класса**

Свойство с наиболее обширным спектром применений. Варианты его применения: Реализовать возможности аспектно-ориентированного подхода - автоматически модифицировать поля и методы класса, основываясь на их имени:

Hightlited/Raw

class PropertyAutoMakerMeta(type):

"""автоматически делает property из всех

всех методов вида get\_Something"""

get\_prefix = 'get\_'

def \_\_new\_\_(cls, name, bases, cls\_dict):

add\_props = {}

for fname, val in cls\_dict.items():

if fname.startswith(cls.get\_prefix) and callable(val):

prop\_name = fname[len(cls.get\_prefix):]

add\_props[prop\_name] = property(fget=val,

doc=val.\_\_doc\_\_)

cls\_dict.update(add\_props)

return super(PropertyAutoMakerMeta,

cls).\_\_new\_\_(cls, name, bases, cls\_dict)

class PropertyAutoMaker(object):

\_\_metaclass\_\_ = PropertyAutoMakerMeta

class A(PropertyAutoMaker):

def get\_X(self):

return 1

print A().X # 1

Изменять наборы параметров методов, применять к ним декораторы, менять документацию, байтокод, добавлять/убирать базовые классы, etc. Например - проверка соответствия объекта заявленному интерфейсу:

Hightlited/Raw

from interface import Interface, ImplementsBase

class MyInterface(Interface):

def func(x, y, z):

ok(x).is\_a(int)

ok(y).in\_((1,2,3))

class Impl(ImplementsBase):

\_\_implements\_\_ = [MyInterface]

def func(self, x, y, z=13):

pass

Код модуля interface слишком большой для этой стать и находится здесь: github.com/koder-ua/Interface\_example .

При конструировании Impl будет проверенно что он предоставляет все методы, требуемые от MyInterface и совместимость сигнатур методов (т.е. что любой набор параметров, который синтаксически подходит для MyInterface.func синтаксически подходит и для Impl.func). Также при отладочных настройках перед каждым вызовом Impl.func будет вызываться MyInterface.func, которая проверит ограничения на входные параметры. Т.е. система с одной стороны проверяет, что класс предоставляет необходимые методы, а с другой проверяет что при вызове в методы передаются правильные параметры. Эту функциональность можно расширить разными способами - извлекать ограничения на типы из строк документации, вести реестр всех реализаций интерфейса, добавить автоматическую проверку пост и пред условий на объект, проверять поля класса и т.д.

Простой ORM:

Hightlited/Raw

class Field(object):

class Base(object):

pass

class Int(Base):

pass

class String(Base):

pass

class LittleORMMeta(type):

def \_\_new\_\_(cls, name, bases, cls\_dict):

fields = []

for fname, tp in cls\_dict.items():

try:

if issubclass(tp, Field.Base):

fields.append(fname)

except TypeError:

pass

cls\_dict['\_fields'] = fields

return super(LittleORMMeta, cls).\_\_new\_\_(cls, name,

bases, cls\_dict)

def \_\_lshift\_\_(self, fields):

self.insert(\*\*fields)

class Table(object):

\_\_metaclass\_\_ = LittleORMMeta

@classmethod

def execute(cls, request):

print request

@classmethod

def insert(cls, \*\*fields):

insert\_request = "insert into {0} ({1}) values ({2})"

values = ','.join(

repr(fields[fname])

for fname in cls.\_fields)

field\_names = ','.join(cls.\_fields)

cls.execute(

insert\_request.format( cls.\_\_name\_\_, field\_names, values)

)

class MyTable(Table):

rec\_id = Field.Int

name = Field.String

#Table.connect(conn\_str)

MyTable << dict(rec\_id=1, name='a')

MyTable << dict(rec\_id=2, name='b')

MyTable << dict(rec\_id=3, name='c')

Система автоматического логирования всех вызовов. В функции log\_call можно реализовать расширенную фильтрацию логов:

Hightlited/Raw

def logme(class\_name, func):

name = "{0}.{1}".format(class\_name, func)

@functools.wraps(func)

def closure(self, \*dt, \*\*mp):

log\_call(name, dt, mp)

return func(self, \*dt, \*\*mp)

return closure

class CallLoggerMeta(type):

def \_\_new\_\_(cls, name, bases, cdict):

for fname, val in cdict:

if isinstance(val, types.FunctionType):

cdict[fname] = logme(name, val)

return super(CallLoggerMeta, cls).\

\_\_new\_\_(cls, name, bases, cdict)

В общем можно слепить из входящего в полях name, bases и cls\_dict пластилина то, что нам нужно. Обещанный пример класса, ведущего список всех своих экземпляров и позволяющего по ним итерировать:

Hightlited/Raw

class IterOverChildsMeta(type):

def \_\_new\_\_(cls, name, bases, cls\_dict):

fname = '\_{0}\_\_all\_childs'.format(name)

cls\_dict[fname] = []

ntype = super(IterOverChildsMeta,

cls).\_\_new\_\_(cls, name, bases, cls\_dict)

old\_new = ntype.\_\_new\_\_

def closure(cls1, \*args, \*\*kwargs):

# workaroud for warning

if old\_new is object.\_\_new\_\_:

obj = old\_new(cls1)

else:

obj = old\_new(cls1, \*args, \*\*kwargs)

# skip objects of a child classes

if cls1.\_\_name\_\_ == name:

# real code should use weakref.ref here

ntype.\_\_dict\_\_[fname].append(obj)

return obj

ntype.\_\_new\_\_ = classmethod(closure)

return ntype

def \_\_iter\_\_(self):

return iter(getattr(self,

'\_{0}\_\_all\_childs'.format(self.\_\_name\_\_)))

class IterOverChilds(object):

\_\_metaclass\_\_ = IterOverChildsMeta

class A(IterOverChilds):

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

self.name = name

def \_\_str\_\_(self):

return "<A name={0!r}>".format(self.name)

a1 = A('a1')

a2 = A('a2')

for obj in A:

print obj

Теория, часть 2 - наследование метаклассов и метаклассы-функции

Python позволяет присвоить полю \_\_metaclass\_\_ функцию - она будет вызвана вместо методов \_\_new\_\_ и \_\_init\_\_ метакласса.

Hightlited/Raw

def meta\_func(name, bases, dct):

print "meta\_func called"

return type(name, bases, dct)

class F(object):

\_\_metaclass\_\_ = meta\_func

# здесь напечатается 'meta\_func called'

Если у одного из родительских классов тип отличается от type, то для создания дочернего класса будет использован он и он же станет типом дочернего класса - это позволяет метаклассам наследоваться. Функция-метакласс не может стать типом создаваемого класса и, соответственно, не наследуется. Итого - пусть у нас есть такая иерархия классов:

Hightlited/Raw

class MetaClass(type):

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

print name + ":metaclass == MetaClass"

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

def meta\_func(name, bases, dct):

print name + ":metaclass == meta\_func"

return type(name, bases, dct)

class A(object):

\_\_metaclass\_\_ = MetaClass

class B(A):

pass

class C(object):

pass

class D(A, C):

pass

class E(C, A):

pass

class F(object):

\_\_metaclass\_\_ = meta\_func

class G(F):

pass

for cls in [A, B, C, D, E, F, G]:

print "{0}.\_\_class\_\_ == {1}".format(cls.\_\_name\_\_,

cls.\_\_class\_\_.\_\_name\_\_)

Вопрос - что будет напечатанное при ее конструировании и какие метаклассы будут у полученных классов? Ответ: для конструирования A, B, D и E будет использован MetaClass, для конструирования F будет использована meta\_func. Для конструирования C и G будет использован type. Для A, B, D и E \_\_class\_\_ будет MetaClass, для F, G и C будет type.

А как все обстоит с множественным наследованием, если у двух или более базовых классов есть метакласс? Python не может самостоятельно разобраться в этой ситуации и заставит нас сделать все руками:

Hightlited/Raw

class MetaClass(type):

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

print name + ":metaclass == MetaClass"

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

class A(object):

\_\_metaclass\_\_ = MetaClass

class SecondMetaClass(type):

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

print name + ":metaclass == SecondMetaClass"

return super(SecondMetaClass,

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

class H(object):

\_\_metaclass\_\_ = SecondMetaClass

class J(H, A):

pass

Это приведет к:

Hightlited/Raw

Traceback (most recent call last):

File "test.py", line 19, in <module>

class J(H, A):

File "test.py", line 14, in \_\_new\_\_

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

TypeError: Error when calling the metaclass bases

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

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

Чтобы унаследовать классы A и H в указанном примере, нужно создать класс, наследующий MetaClass и SecondMetaClass и определить его, как метакласс для J. Впрочем создание такого класса можно автоматизировать:

Hightlited/Raw

def meta\_base(name, bases, cdict, add\_meta = tuple()):

meta\_set = set(cls.\_\_class\_\_

for cls in ( bases + add\_meta )

if cls.\_\_class\_\_ is not type)

if len(meta\_set) != 0:

new\_meta = type("tempo\_meta", tuple(meta\_set), {})

else:

new\_meta = type

return new\_meta(name, bases, cdict)

class J(H, A):

\_\_metaclass\_\_ = meta\_base

Если нужно добавить еще метаклассов, кроме базовых:

Hightlited/Raw

def meta\_base\_plus(\*\*metas):

def closure(name, bases, cdict) :

return meta\_base(name, bases, cdict,

add\_meta=metas)

return closure

class Jplus(H, A):

\_\_metaclass\_\_ = meta\_base\_plus(SomeAdditionalMeta1,

SomeAdditionalMeta2)

Метаклассы представляют достаточно мощный инструмент для создания повторно используемого кода, но результат их деятельности может стать большой неожиданностью для тех, кто будет их использовать. Это подчеркивает важность документирования всей нетривиальной функциональности, реализованной с их помощью.

**Disclamer**

Приведенный код сознательно сокращен для упрощения понимания. Для реального использования его нужно дорабатывать, впрочем не сильно.

Все описанное относится к "новым классам", т.е. прямо или косвенно унаследованным от object. Все кто во втором десятилетии 21го века не наследуют свои классы от object создают себе лишние проблемы. В python3 все объекты по умолчанию "новые".

Про метаклассы в python написано достаточно много (см. ссылки). Эта статья основывается на моем опыте преподавания python, описывает некоторые темы, которые сложно найти (например почему метаклассы-функции не наследуются) и включает большое количество примеров практического применения метаклассов.