Метаклассы в Python

<http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python>

Источник: https://habr.com/post/145835/

*Как сказал один из пользователей StackOverflow, «using SO is like doing lookups with a hashtable instead of a linked list». Мы снова обращаемся к этому замечательному ресурсу, на котором попадаются чрезвычайно подробные и понятные ответы на самые различные вопросы.  
  
В этот раз мы обсудим, что такое метаклассы, как, где и зачем их использовать, а также почему обычно этого делать не стоит.*

Классы как объекты

Перед тем, как изучать метаклассы, надо хорошо разобраться с классами, а классы в Питоне — вещь весьма специфическая (основаны на идеях из языка Smalltalk).  
  
В большинстве языков класс это просто кусок кода, описывающий, как создать объект. В целом это верно и для Питона:

>>> class ObjectCreator(object):

... pass

...

>>> my\_object = ObjectCreator()

>>> print my\_object

<\_\_main\_\_.ObjectCreator object at 0x8974f2c>

Но в Питоне класс это нечто большее — классы также являются объектами.  
  
Как только используется ключевое слово class, Питон исполняет команду и создаёт *объект*. Инструкция

>>> class ObjectCreator(object):

... pass

...

создаст в памяти объект с именем ObjectCreator.  
  
Этот объект (класс) сам может создавать объекты (экземпляры), поэтому он и является классом.  
  
Тем не менее, это объект, а потому:

* его можно присвоить переменной,
* его можно скопировать,
* можно добавить к нему атрибут,
* его можно передать функции в качестве аргумента,

Динамическое создание классов

Так как классы являются объектами, их можно создавать на ходу, как и любой объект.  
  
Например, можно создать класс в функции, используя ключевое слово class:

>>> def choose\_class(name):

... if name == 'foo':

... class Foo(object):

... pass

... return Foo *# возвращает класс, а не экземпляр*

... else:

... class Bar(object):

... pass

... return Bar

...

>>> MyClass = choose\_class('foo')

>>> print MyClass *# функция возвращает класс, а не экземпляр*

<class '\_\_main\_\_.Foo'>

>>> print MyClass() # можно создать экземпляр этого класса

<\_\_main\_\_.Foo object at 0x89c6d4c>

Однако это не очень-то динамично, поскольку по-прежнему нужно самому писать весь класс целиком.  
  
Поскольку классы являются объектами, они должны генерироваться чем-нибудь.  
  
Когда используется ключевое слово class, Питон создаёт этот объект автоматически. Но как и большинство вещей в Питоне, есть способ сделать это вручную.  
  
Помните функцию type? Старая-добрая функция, которая позволяет определить тип объекта:

>>> print type(1)

<type 'int'>

>>> print type("1")

<type 'str'>

>>> print type(ObjectCreator)

<type 'type'>

>>> print type(ObjectCreator())

<class '\_\_main\_\_.ObjectCreator'>

На самом деле, у функции type есть совершенно иное применение: она также может создавать классы на ходу. type принимает на вход описание класса и созвращает класс.  
  
(Я знаю, это по-дурацки, что одна и та же функция может использоваться для двух совершенно разных вещей в зависимости от передаваемых аргументов. Так сделано для обратной совместимости)  
  
type работает следующим образом:

type(<имя класса>,

<кортеж родительских классов>, *# для наследования, может быть пустым*

<словарь, содержащий атрибуты и их значения>)

Например,

>>> class MyShinyClass(object):

... pass

может быть создан вручную следующим образом:

>>> MyShinyClass = type('MyShinyClass', (), {}) *# возвращает объект-класс*

>>> print MyShinyClass

<class '\_\_main\_\_.MyShinyClass'>

>>> print MyShinyClass() # создаёт экземпляр класса

<\_\_main\_\_.MyShinyClass object at 0x8997cec>

Возможно, вы заметили, что мы используем «MyShinyClass» и как имя класса, и как имя для переменной, содержащей ссылку на класс. Они могут быть различны, но зачем усложнять?  
  
type принимает словарь, определяющий атрибуты класса:

>>> class Foo(object):

... bar = True

можно переписать как

>>> Foo = type('Foo', (), {'bar':True})

и использовать как обычный класс

>>> print Foo

<class '\_\_main\_\_.Foo'>

>>> print Foo.bar

True

>>> f = Foo()

>>> print f

<\_\_main\_\_.Foo object at 0x8a9b84c>

>>> print f.bar

True

Конечно, можно от него наследовать:

>>> class FooChild(Foo):

... pass

превратится в

>>> FooChild = type('FooChild', (Foo,), {})

>>> print FooChild

<class '\_\_main\_\_.FooChild'>

>>> print FooChild.bar # bar is inherited from Foo

True

В какой-то момент вам захочется добавить методов вашему классу. Для этого просто определите функцию с нужной сигнатурой и присвойте её в качестве атрибута:

>>> def echo\_bar(self):

... print self.bar

...

>>> FooChild = type('FooChild', (Foo,), {'echo\_bar': echo\_bar})

>>> hasattr(Foo, 'echo\_bar')

>>> hasattr(FooChild, 'echo\_bar')

True

>>> my\_foo = FooChild()

>>> my\_foo.echo\_bar()

True

Уже понятно, к чему я клоню: в Питоне классы являются объектами и можно создавать классы на ходу.  
  
Это именно то, что Питон делает, когда используется ключевое слово class, и делает он это с помощью метаклассов.

Что такое метакласс (наконец)

Метакласс это «штука», которая создаёт классы.  
  
Мы создаём класс для того, чтобы создавать объекты, так? А классы являются объектами. Метакласс это то, что создаёт эти самые объекты. Они являются классами классов, можно представить это себе следующим образом:

MyClass = MetaClass()

MyObject = MyClass()

Мы уже видели, что type позволяет делать что-то в таком духе:

MyClass = type('MyClass', (), {})

Это потому что функция type на самом деле является метаклассом. type это метакласс, который Питон внутренне использует для создания всех классов.  
  
Естественный вопрос: с чего это он его имя пишется в нижнем регистре, а не Type?  
  
Я полагаю, это просто для соответствия str, классу для создания объектов-строк, и int, классу для создания объектов-целых чисел. type это просто класс для создания объектов-классов.  
  
Это легко проверить с помощью атрибута \_\_class\_\_:  
  
В питоне всё (вообще всё!) является объектами. В том числе числа, строки, функции и классы — они все являются объектами и все были созданы из класса:

>>> age = 35

>>> age.\_\_class\_\_

<type 'int'>

>>> name = 'bob'

>>> name.\_\_class\_\_

<type 'str'>

>>> def foo(): pass

>>> foo.\_\_class\_\_

<type 'function'>

>>> class Bar(object): pass

>>> b = Bar()

>>> b.\_\_class\_\_

<class '\_\_main\_\_.Bar'>

А какой же \_\_class\_\_ у каждого \_\_class\_\_?

>>> a.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> age.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> foo.\_\_class\_\_.\_\_class\_\_

<type 'type'>

>>> b.\_\_class\_\_.\_\_class\_\_

<type 'type'>

Итак, метакласс это просто штука, создающая объекты-классы.  
  
Если хотите, можно называть его «фабрикой классов»  
  
type это встроенный метакласс, который использует Питон, но вы, конечно, можете создать свой.

Атрибут \_\_metaclass\_\_

При написании класса можно добавить атрибут \_\_metaclass\_\_:

class Foo(object):

\_\_metaclass\_\_ = something...

[...]

В таком случае Питон будет использовать указанный метакласс при создании класса Foo.  
  
Осторожно, тут есть тонкость!  
  
Хоть вы и пишете class Foo(object), объект-класс пока ещё не создаётся в памяти.  
  
Питон будет искать \_\_metaclass\_\_ в определении класса. Если он его найдёт, то использует для создания класса Foo. Если же нет, то будет использовать type.  
  
То есть когда вы пишете

class Foo(Bar):

pass

Питон делает следующее:  
  
Есть ли у класса Foo атрибут \_\_metaclass\_\_?  
  
Если да, создаёт в памяти объект-класс с именем Foo, используя то, что указано в \_\_metaclass\_\_.  
  
Если Питон не находит \_\_metaclass\_\_, он ищет \_\_metaclass\_\_ в родительском классе Bar и попробует сделать то же самое.  
  
Если же \_\_metaclass\_\_ не находится ни в одном из родителей, Питон будет искать \_\_metaclass\_\_ на уровне *модуля*.  
  
И если он не может найти вообще ни одного \_\_metaclass\_\_, он использует type для создания объекта-класса.  
  
Теперь важный вопрос: что можно положить в \_\_metaclass\_\_?  
  
Ответ: что-нибудь, что может создавать классы.  
  
А что создаёт классы? type или любой его подкласс, а также всё, что использует их.

Пользовательские метаклассы

Основная цель метаклассов — автоматически изменять класс в момент создания.  
  
Обычно это делает для API, когда хочется создавать классы в соответсвии с текущим контекстом.  
  
Представим глупый пример: вы решили, что у всех классов в вашем модуле имена атрибутов должны быть записать в верхнем регистре. Есть несколько способов это сделать, но один из них — задать \_\_metaclass\_\_ на уровне модуля.  
  
В таком случае все классы этого модуля будут создаваться с использованием указанного меакласса, а нам остаётся только заставить метакласса переводить имена всех атрибутов в верхний регистр.  
  
К счастью, \_\_metaclass\_\_ может быть любым вызываемым объектом, не обязательно формальным классом (я знаю, что-то со словом «класс» в названии не обязано быть классом, что за ерунда? Однако это полезно).  
  
Так что мы начнём с простого примера, используя функцию.

*# метаклассу автоматически придёт на вход те же аргументы,*

*# которые обычно используются в `type`*

def upper\_attr(future\_class\_name, future\_class\_parents, future\_class\_attr):

"""

Возвращает объект-класс, имена атрибутов которого

переведены в верхний регистр

"""

*# берём любой атрибут, не начинающийся с '\_\_'*

attrs = ((name, value) for name, value in future\_class\_attr.items() if not name.startswith('\_\_'))

*# переводим их в верхний регистр*

uppercase\_attr = dict((name.upper(), value) for name, value in attrs)

*# создаём класс с помощью `type`*

return type(future\_class\_name, future\_class\_parents, uppercase\_attr)

\_\_metaclass\_\_ = upper\_attr *# это сработает для всех классов в модуле*

class Foo(object):

*# или можно определить \_\_metaclass\_\_ здесь, чтобы сработало только для этого класса*

bar = 'bip'

print hasattr(Foo, 'bar')

*# Out: False*

print hasattr(Foo, 'BAR')

*# Out: True*

f = Foo()

print f.BAR

*# Out: 'bip'*

А теперь то же самое, только используя настояший класс:

*# помним, что `type` это на само деле класс, как `str` и `int`,*

*# так что от него можно наследовать*

class UpperAttrMetaclass(type):

*# Метод \_\_new\_\_ вызывается перед \_\_init\_\_*

*# Этот метод создаёт обхект и возвращает его,*

*# в то время как \_\_init\_\_ просто инициализирует объект, переданный в качестве аргумента.*

*# Обычно вы не используете \_\_new\_\_, если только не хотите проконтролировать,*

*# как объект создаётся*

*# В данном случае созданный объект это класс, и мы хотим его настроить,*

*# поэтому мы перегружаем \_\_new\_\_.*

*# Можно также сделать что-нибудь в \_\_init\_\_, если хочется.*

*# В некоторых более продвинутых случаях также перегружается \_\_call\_\_,*

*# но этого мы сейчас не увидим.*

def \_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, future\_class\_attr):

attrs = ((name, value) for name, value in future\_class\_attr.items() if not name.startswith('\_\_'))

uppercase\_attr = dict((name.upper(), value) for name, value in attrs)

return type(future\_class\_name, future\_class\_parents, uppercase\_attr)

Но это не совсем ООП. Мы напрямую вызываем type и не перегружаем вызов \_\_new\_\_ родителя. Давайте сделаем это:

class UpperAttrMetaclass(type):

def \_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, future\_class\_attr):

attrs = ((name, value) for name, value in future\_class\_attr.items() if not name.startswith('\_\_'))

uppercase\_attr = dict((name.upper(), value) for name, value in attrs)

*# используем метод type.\_\_new\_\_*

*# базовое ООП, никакой магии*

return type.\_\_new\_\_(upperattr\_metaclass, future\_class\_name,

future\_class\_parents, uppercase\_attr)

Вы, возможно, заметили дополнительный аргумент upperattr\_metaclass. Ничего особого в нём нет: метод всегда получает первым аргументом текущий экземпляр. Точно так же, как вы используете self в обычным методах.  
  
Конечно, имена, которые я тут использовал, такие длинные для ясности, но как и self, есть соглашение об именовании всех этих аргументов. Так что реальный метакласс выгляит как-нибудь так:

class UpperAttrMetaclass(type):

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

attrs = ((name, value) for name, value in dct.items() if not name.startswith('\_\_'))

uppercase\_attr = dict((name.upper(), value) for name, value in attrs)

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

Можно сделать даже лучше, использовав super, который вызовет наследование (поскольку, конечно, можно создать метакласс, унаследованный от метакласса, унаследованного от type):

class UpperAttrMetaclass(type):

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

attrs = ((name, value) for name, value in dct.items() if not name.startswith('\_\_'))

uppercase\_attr = dict((name.upper(), value) for name, value in attrs)

return super(UpperAttrMetaclass, cls).\_\_new\_\_(cls, name, bases, uppercase\_attr)

Вот и всё. О метаклассах больше ничего и не сказать.  
  
Причина сложности кода, использующего метаклассы, не в самих метаклассах. Она в том, что обычно метаклассы используются для всяких изощрённых вещей, основанных на интроспекции, манипуляцией наследованием, переменными вроде \_\_dict\_\_ и тому подобном.  
  
Действительно, метаклассы особенно полезны для всякой «чёрной магии», а, следовательно, сложных штук. Но сами по себе они просты:

* перехватить создание класса
* изменить класс
* вернуть модифицированный

Зачем использовать метаклассы вместо функций?

Поскольку \_\_metaclass\_\_ принимает любой вызываемый объект, с чего бы вдруг использовать класс, если это очевидно сложнее?  
  
Тому есть несколько причин:

* Назначение яснее. Когда вы видите UpperAttrMetaclass(type), вы сразу знаете, что дальше будет.
* Можно использовать ООП. Метаклассы могту наследоваться от метаклассов, перегружая родитальские методы.
* Лучше структурированный код. Вы не будете использовать метаклассы для таких простых вещей, как в примере выше. Обычно это что-то сложное. Возможность создать несколько методов и сгруппировать их в одном классе очень полезна, чтобы сделать код более удобным для чтения.
* Можно использовать \_\_new\_\_, \_\_init\_\_ и \_\_call\_\_. Конечно, обычно можно всё сделать в \_\_new\_\_, но некоторым комфортнее использовать \_\_init\_\_
* Они называются мета*классами*, чёрт возьми! Это должно что-то значить!

Зачем вообще использовать метаклассы?

Наконец, главный вопрос. С чего кому-то использовать какую-то непонятную (и способствующую ошибкам) фичу?  
  
Ну, обычно и не надо использовать:

Метаклассы это глубокая магия, о которой 99% пользователей даже не нужно задумываться. Если вы думаете, нужно ли вам их использовать — вам не нужно (люди, которым они реально нужны, точно знают, зачем они им, и не нуждаются в объяснениях, почему).  
~ Гуру Питона Тим Питерс

Основное применение метаклассов это создание API. Типичный пример — Django ORM.  
  
Она позволяет написать что-то в таком духе:

class Person(models.Model):

name = models.CharField(max\_length=30)

age = models.IntegerField()

Однако если вы выполните следующий код:

guy = Person(name='bob', age='35')

print guy.age

вы получите не IntegerField, а int, причём значение может быть получено прямо из базы данных.  
  
Это возможно, потому что models.Model определяет \_\_metaclass\_\_, который сотворит некую магию и превратит класс Person, который мы только что определили простым выражением в сложную привязку к базе данных.  
  
Django делает что-то сложное выглядящим простым, выставляя наружу простое API и используя метаклассы, воссоздающие код из API и незаметно делающие всю работу.

Напоследок

ВО-первых, вы узнали, что классы это объекты, которые могут создавать экземпляры.  
  
На самом деле, классы это тоже экземпляры. Экземпляры метаклассов.

>>> class Foo(object): pass

>>> id(Foo)

142630324

Всё что угодно является объектом в Питоне: экземпляром класса или экземпляром метакласса.  
  
Кроме type.  
  
type является собственным метаклассом. Это нельзя воспроизвести на чистом Питоне и делается небольшим читерством на уровне реализации.  
  
Во-вторых, метаклассы сложны. Вам не нужно использовать их для простого изменения классов. Это можно делать двумя разными способами:

* руками
* декораторы классов

В 99% случаев, когда вам нужно изменить класс, лучше использовать эти два.  
  
Но в 99% случаев вам вообще не нужно изменять классы :-)

Once more about this theme here - http://qaru.site/questions/102/what-is-a-metaclass-in-python