Руководство к дескрипторам перевод

Краткий обзор

В этой статье я расскажу о том, что такое дескрипторы, о протоколе дескрипторов, покажу как вызываются дескрипторы. Опишу создание собственных и исследую несколько встроенных дескрипторов, включая функции, свойства, статические методы и методы класса. С помощью простого приложения покажу, как работает каждый из них, приведу эквиваленты внутренней реализации работы дескрипторов кодом на чистом питоне.

Изучение того, как работают дескрипторы, откроет доступ к большему числу рабочих инструментов, поможет лучше понять как работает питон, и ощутить элегантность его дизайна.

Введение и определения

Если говорить в общем, то дескриптор — это атрибут объекта со связанным поведением (англ. binding behavior), т.е. такой, чьё поведение при доступе переопределяется методами протокола дескриптора. Эти методы: \_\_get\_\_, \_\_set\_\_ и \_\_delete\_\_. Если хотя бы один из этих методов определён для объекта, то он становится дескриптором.

Стандартное поведение при доступе к атрибутам — это получение, установка и удаление атрибута из словаря объекта. Например, a.x имеет такую цепочку поиска атрибута: a.\_\_dict\_\_['x'], затем в type(a).\_\_dict\_\_['x'], и далее по базовым классам type(a) не включая метаклассы. Если же искомое значение — это объект, в котором есть хотя бы один из методов, определяющих дескриптор, то питон может изменить стандартную цепочку поиска и вызвать один из методов дескриптора. Как и когда это произойдёт зависит от того, какие методы дескриптора определены для объекта. Дескрипторы вызываются только для объектов или классов нового стиля (класс является таким, если наследует от object или type).

Дескрипторы — это мощный протокол с широкой областью применения. Они являются тем механизмом, который стоит за свойствами, методами, статическими методами, методами класса и вызовом super(). Внутри самого питона с их помощью реализуются классы нового стиля, которые были представлены в версии 2.2. Дескрипторы упрощают понимание нижележащего кода на C, а также представляют гибкий набор новых инструментов для любых программ на питоне.

Протокол дескрипторов

descr.\_\_get\_\_(self, obj, type=None) --> значение

descr.\_\_set\_\_(self, obj, value) --> None

descr.\_\_delete\_\_(self, obj) --> None

Собственно это всё. Определите любой из этих методов и объект будет считаться дескриптором, и сможет переопределять стандартное поведение, если его будут искать как атрибут.

Если объект определяет сразу и \_\_get\_\_, и \_\_set\_\_, то он считается дескриптором данных (англ. data descriptor). Дескрипторы, которые определили только \_\_get\_\_ называются дескрипторами не данных (англ. non-data descriptors). Их называются так, потому что они используют для методов, но другие способы их применения также возможны.

Дескрипторы данных и не данных отличаются в том, как будет изменено поведение поиска, если в словаре объекта уже есть запись с таким же именем как у дескриптора. Если попадается дескриптор данных, то он вызывается раньше, чем запись из словаря объекта. Если в такой же ситуации окажется дескриптор не данных, то запись из словаря объекта имеет преимущество перед этим дескриптором.

Чтобы создать дескриптор данных только для чтения, определите и \_\_get\_\_, и \_\_set\_\_, и сделайте так, чтобы \_\_set\_\_ выбрасывал исключение AttributeError. Определения метода \_\_set\_\_ и выбрасывания исключения достаточно, чтобы этот дескриптор считался дескриптором данных.

Вызов дескрипторов

Дескриптор можно вызвать напрямую через его метод. Например, d.\_\_get\_\_(obj).

Однако, наиболее частый вариант вызова дескриптора — это автоматический вызов во время доступа к атрибуту. Например, obj.d ищет d в словаре obj. Если d определяет метод \_\_get\_\_, то будет вызван d.\_\_get\_\_(obj). Вызов будет сделан согласно правилам, описанным ниже.

Детали вызова различаются от того, чем является obj — объектом или классом. В любом случае, дескрипторы работают только для объектов и классов нового стиля. Класс является классом нового стиля, если он является потомком object.

Для объектов алгоритм реализуется с помощью object.\_\_getattribute\_\_, который преобразует запись b.x в type(b).\_\_dict\_\_['x'].\_\_get\_\_(b, type(b)). Реализация работает через цепочку предшественников, в которой дескрипторы данных имеют приоритет перед переменными объекта, переменные объекта имеют приоритет перед дескрипторами не данных, и самый низкий приоритет у метода \_\_getattr\_\_, если он определён. Полную реализацию на языке C можно найти в PyObject\_GenericGetAttr() в файле Objects/object.c.

Для классов алгоритм реализуется с помощью type.\_\_getattribute\_\_, который преобразует запись B.x в B.\_\_dict\_\_['x'].\_\_get\_\_(None, B). На чистом питоне это выглядит так:

def \_\_getattribute\_\_(self, key):

"Эмуляция type\_getattro() в Objects/typeobject.c"

v = object.\_\_getattribute\_\_(self, key)

if hasattr(v, '\_\_get\_\_'):

return v.\_\_get\_\_(None, self)

return v

Важные части, которые следует запомнить:

дескрипторы вызываются с помощью метода \_\_getattribute\_\_

переопределение \_\_getattribute\_\_ прекратит автоматический вызов дескрипторов

\_\_getattribute\_\_ доступен только внутри классов и объектов нового стиля

object.\_\_getattribute\_\_ и type.\_\_getattribute\_\_ делают разные вызовы к \_\_get\_\_

дескрипторы данных всегда имеют преимущество перед переменными объекта

дескрипторы не данных могут потерять преимущество из-за переменных объекта

Объект, который возвращается после вызова super() также имеет собственную реализацию метода \_\_getattribute\_\_, с помощью которой вызывает дескрипторы. Вызов super(B, obj).m() ищет в obj.\_\_class\_\_.\_\_mro\_\_ базовый класс A, за которым сразу следует B, и возвращает A.\_\_dict\_\_['m'].\_\_get\_\_(obj, A). Если это не дескриптор, то m возвращается неизменённой. Если m нет в словаре, то возвращаемся к поиску через object.\_\_getattribute\_\_.

Примечание: в питоне 2.2, super(B, obj).m() вызывал \_\_get\_\_ только если m был дескриптором данных. В питоне 2.3, дескрипторы не данных тоже вызываются, за исключением тех случаев, когда используются классы старого стиля. Детали реализации можно найти в super\_getattro() в файле Objects/typeobject.c, а эквивалент на чистом питоне можно найти в пособии от Guido.

Детали выше описывают, что алгоритм вызова дескрипторов реализуется с помощью метода \_\_getattribute\_\_() для object, type и super. Классы наследуют этот алгоритм, когда они наследуют от object или если у них есть метакласс, реализующий подобную функциональность. Таким образом, классы могут отключить вызов дескрипторов, если переопределят \_\_getattribute\_\_().

Пример дескриптора

Следующий код создаёт класс, чьи объекты являются дескрипторам данных и всё, что они делают — это печатают сообщение на каждый вызов get или set. Переопределение \_\_getattribute\_\_ — это альтернативный подход, с помощью которого мы могли бы сделать это для каждого атрибута. Но если мы хотим наблюдать только за отдельными атрибутами, то это проще сделать с помощью дескриптора.

class RevealAccess(object):

"""Дескриптор данных, который устанавливает и возвращает значения,

и печатает сообщение о том, что к атрибуту был доступ.

"""

def \_\_init\_\_(self, initval=None, name='var'):

self.val = initval

self.name = name

def \_\_get\_\_(self, obj, objtype):

print 'Получаю', self.name

return self.val

def \_\_set\_\_(self, obj, val):

print 'Обновляю' , self.name

self.val = val

>>> class MyClass(object):

x = RevealAccess(10, 'var "x"')

y = 5

>>> m = MyClass()

>>> m.x

Получаю var "x"

10

>>> m.x = 20

Обновляю var "x"

>>> m.x

Получаю var "x"

20

>>> m.y

5

Этот простой протокол предоставляет просто увлекательные возможности. Некоторые из них настолько часто используются, что были объединены в отдельные функции. Свойства, связанные и несвязанные методы, статические методы и методы класса — все они основаны на этом протоколе.

Свойства

Вызова property() достаточно, чтобы создать дескриптор данных, который вызывает нужные функции во время доступа к атрибуту. Вот его сигнатура:

property(fget=None, fset=None, fdel=None, doc=None) --> атрибут, реализующий свойства

В документации показано типичное использование property() для создания управляемого атрибута x:

class C(object):

def getx(self): return self.\_\_x

def setx(self, value): self.\_\_x = value

def delx(self): del self.\_\_x

x = property(getx, setx, delx, "Я свойство 'x'.")

Вот эквивалент property на чистом питоне, чтобы было понятно как реализовано property() с помощью протокола дескрипторов:

class Property(object):

"Эмуляция PyProperty\_Type() в Objects/descrobject.c"

def \_\_init\_\_(self, fget=None, fset=None, fdel=None, doc=None):

self.fget = fget

self.fset = fset

self.fdel = fdel

self.\_\_doc\_\_ = doc

def \_\_get\_\_(self, obj, objtype=None):

if obj is None:

return self

if self.fget is None:

raise AttributeError, "нечитаемый атрибут"

return self.fget(obj)

def \_\_set\_\_(self, obj, value):

if self.fset is None:

raise AttributeError, "не могу установить атрибут"

self.fset(obj, value)

def \_\_delete\_\_(self, obj):

if self.fdel is None:

raise AttributeError, "не могу удалить атрибут"

self.fdel(obj)

Встроенная реализация property() может помочь, когда существовал интерфейс доступа к атрибуту и произошли какие-то изменения, в результате которых понадобилось вмешательство метода.

Например, класс электронной таблицы может давать доступ к значению ячейки через Cell('b10').value. В результате последующих изменений в программе, понадобилось сделать так, чтобы это значение пересчитывалось при каждом доступе к ячейке, однако программист не хочет менять клиентский код, который обращается к атрибуту напрямую. Эту проблему можно решить, если обернуть атрибут value с помощью дескриптора данных, который будет создан с помощью property():

class Cell(object):

. . .

def getvalue(self, obj):

"Пересчитываем ячейку прежде чем вернуть значение"

self.recalc()

return obj.\_value

value = property(getvalue)

Функции и методы

В питоне все объектно-ориентированные возможности реализованы с помощью функционального подхода. Это сделано совсем незаметно с помощью дескрипторов не данных.

Словари классов хранят методы в виде функций. При определении класса, методы записываются с помощью def и lambda — стандартных инструментов для создания функций. Единственное отличие этих функций от обычных в том, что первый аргумент зарезервирован под экземпляр объекта. Этот аргумент обычно называется self, но может называться this или любым другим словом, которым можно называть переменные.

Для того, чтобы поддерживать вызов методов, функции включают в себя метод \_\_get\_\_, который автоматически делает их дескрипторами не данных при поиске атрибутов. Функции возвращают связанные или не связанные методы, в зависимости от того, через что был вызван этот дескриптор.

class Function(object):

. . .

def \_\_get\_\_(self, obj, objtype=None):

"Симуляция func\_descr\_get() в Objects/funcobject.c"

return types.MethodType(self, obj, objtype)

С помощью интерпретатора мы можем увидеть как на самом деле работает дескриптор функции:

>>> class D(object):

def f(self, x):

return x

>>> d = D()

>>> D.\_\_dict\_\_['f'] # Внутренне хранится как функция

<function f at 0x00C45070>

>>> D.f # Доступ через класс возвращает несвязанный метод

<unbound method D.f>

>>> d.f # Доступ через экземпляр объекта возвращает связанный метод

<bound method D.f of <\_\_main\_\_.D object at 0x00B18C90>>

Вывод интерпретатора подсказывает нам, что связанные и несвязанные методы — это два разных типа. Даже если они могли бы быть реализованы таким образом, на самом деле, реализация PyMethod\_Type в файле Objects/classobject.c содержит единственный объект с двумя различными отображениями, которые зависят только от того, есть ли в поле im\_self значение или там содержится NULL (C эквивалент значения None).

Таким образом, эффект вызова метода зависит от поля im\_self. Если оно установлено (т.е. метод связан), то оригинальная функция (хранится в поле im\_func) вызывается, как мы и ожидаем, с первым аргументом, установленным в значение экземпляра объекта. Если же она не связана, то все аргументы передаются без изменения оригинальной функции. Настоящая C реализация instancemethod\_call() чуть более сложная, потому что включает в себя некоторые проверки типов и тому подобное.

Статические методы и методы класса

Дескрипторы не данных предоставляют простой механизм для различных вариантов привязки функций к методам.

Повторим ещё раз. Функции имеют метод \_\_get\_\_, с помощью которых они становятся методами, во время поиска атрибутов и автоматического вызова дескрипторов. Дескрипторы не данных преобразуют вызов obj.f(\*args) в вызов f(obj, \*args), а вызов klass.f(\*args) становится f(\*args).

В этой таблице показано связывание и два наиболее популярных варианта:

Преобразование Вызвана через объект Вызвана через класс

Дескриптор функция f(obj, \*args) f(\*args)

staticmethod f(\*args) f(\*args)

classmethod f(type(obj), \*args) f(klass, \*args)

Статические методы возвращают функцию без изменений. Вызовы c.f или C.f эквиваленты вызовам object.\_\_getattribute\_\_(c, "f") или object.\_\_getattribute\_\_(C, "f"). Как результат, функция одинаково доступна как из объекта, так и из класса.

Хорошими кандидатами для статических методов являются методы, которым не нужна ссылка на переменную self.

Например, пакет для статистики может включать класс для экспериментальных данных. Класс предоставляет обычные методы для расчёта среднего, ожидания, медианы и другой статистики, которая зависит от данных. Однако, там могут быть и другие функции, которые концептуально связаны, но не зависят от данных. Например, erf(x) это простая функция для преобразования, которая нужна в статистике, но не зависит от конкретного набора данных в этом классе. Она может быть вызвана и из объекта, и из класса: s.erf(1.5) --> 0.9332 или Sample.erf(1.5) --> 0.9332.

Так как staticmethod() возвращает функцию без изменений, то этот пример не удивляет:

>>> class E(object):

def f(x):

print x

f = staticmethod(f)

>>> print E.f(3)

3

>>> print E().f(3)

3

Если использовать протокол дескриптора не данных, то на чистом питоне staticmethod() выглядел бы так:

class StaticMethod(object):

"Эмуляция PyStaticMethod\_Type() в Objects/funcobject.c"

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

self.f = f

def \_\_get\_\_(self, obj, objtype=None):

return self.f

В отличие от статических методов, методы класса подставляют в начало вызова функции ссылку на класс. Формат вызова всегда один и тот же, и не зависит от того, вызываем мы метод через объект или через класс.

>>> class E(object):

def f(klass, x):

return klass.\_\_name\_\_, x

f = classmethod(f)

>>> print E.f(3)

('E', 3)

>>> print E().f(3)

('E', 3)

Это поведение удобно, когда нашей функции всегда нужна ссылка на класс и ей не нужны данные. Один из способов использования classmethod() — это создание альтернативных конструкторов класса. В питоне 2.3, метод класса dict.fromkeys() создаёт новый словарь из списка ключей. Эквивалент на чистом питоне будет таким:

class Dict:

. . .

def fromkeys(klass, iterable, value=None):

"Эмуляция dict\_fromkeys() в Objects/dictobject.c"

d = klass()

for key in iterable:

d[key] = value

return d

fromkeys = classmethod(fromkeys)

Теперь новый словарь уникальных ключей можно создать таким образом:

>>> Dict.fromkeys('abracadabra')

{'a': None, 'r': None, 'b': None, 'c': None, 'd': None}

Если использовать протокол дескриптора не данных, то на чистом питоне classmethod() выглядел бы так:

class ClassMethod(object):

"Эмуляция PyClassMethod\_Type() в Objects/funcobject.c"

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

self.f = f

def \_\_get\_\_(self, obj, klass=None):

if klass is None:

klass = type(obj)

def newfunc(\*args):

return self.f(klass, \*args)

return newfunc