Заметки об объектной системе языка Python ч.3

Python\*

Третья часть заметок об объектной системе python'a (первая и вторая части). В статье рассказывается о том, почему c.\_\_call\_\_() не то же самое, что и c(), как реализовать singleton с помощью метаклассов, что такое name mangling и как оно работает.

c.\_\_call\_\_ vs c(), c.\_\_setattr\_\_ vs setattr

Легко убедиться, что x(arg1, arg2) не равносильно x.\_\_call\_\_(arg1, arg2) для новых классов, хотя для старых это справедливо.

>>> class C(object):

... pass

...

>>> c = C()

>>> c.\_\_call\_\_ = lambda: 42

>>> c()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'C' object is not callable

>>> C.\_\_call\_\_ = lambda self: 42

>>> c()

42

На самом деле правильно:

c() <=> type©.\_\_call\_\_(с)

Абсолютно такая же ситуация с \_\_setattr\_\_/setattr и многими другими магическими (и специальными) методами и соответствующими встроенными функциями, которые определены для всех объектов, в том числе и для объектов типа — классов.

Зачем это было сделано можно рассмотреть на примере setattr [1].

В начале убедимся, что setattr(a, 'x', 1) <==> type(a).\_\_setattr\_\_(a, 'x', 1).

a.x = 1 <=> setattr(a, 'x', 1)

>>> class A(object): pass

...

>>> a = A()

>>> a.x = 1

>>> a

<\_\_main\_\_.A object at 0x7fafa9b26f90>

>>> setattr(a, 'y', 2)

>>> a.\_\_dict\_\_

{'y': 2, 'x': 1}

Устанавливаем с помощью метода \_\_setattr\_\_ новый атрибут, который пойдет в \_\_dict\_\_

>>> a.\_\_setattr\_\_('z', 3)

вроде бы все правильно:

>>> a.\_\_dict\_\_

{'y': 2, 'x': 1, 'z': 3}

Однако:

Установим в a.\_\_setattr\_\_ заведомо неправильный метод:

>>> a.\_\_setattr\_\_ = lambda self: 42

Вызов, которого приводит к ошибке:

>>> a.\_\_setattr\_\_('z', 4)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: <lambda>() takes exactly 1 argument (2 given)

Однако, несмотря на это, setattr работает:

>>> setattr(a, 'foo', 'bar')

>>> a.\_\_dict\_\_

{'y': 2, 'x': 1, '\_\_setattr\_\_': <function <lambda> at 0x7fafa9b3a140>, 'z': 3, 'foo': 'bar'}

А вот если переопределить метод класса:

>>> A.\_\_setattr\_\_ = lambda self: 42

то setattr для экземпляра класса выдаст ошибку:

>>> setattr(a, 'baz', 'quux')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: <lambda>() takes exactly 1 argument (3 given)

Зачем это было сделано?

Пусть setattr(a, 'x',1) тоже самое, что a.\_\_setattr\_\_('x', 1), тогда

>>> class A(object):

... def \_\_setattr\_\_(self, attr, value):

... print 'for instances', attr, value

... object.\_\_setattr\_\_(self, attr, value)

...

>>> a = A()

Установим новый атрибут для a. a.x = 1 <==> a.\_\_setattr\_\_('x', 1)

Все нормально:

>>> a.\_\_setattr\_\_('x', 1)

for instances x 1

>>> a.\_\_dict\_\_

{'x': 1}

А теперь попробуем установить новый атрибут для самого класса, он же ведь тоже является объектом: A.foo = 'bar' <==> A.\_\_setattr\_\_('foo', 'bar')

>>> A.\_\_setattr\_\_('foo', 'bar')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unbound method \_\_setattr\_\_() must be called with A instance as first argument (got str instance instead)

Все логично, согласно алгоритму поиска атрибутов в классах (типах), сначала атрибут ищется в \_\_dict\_\_ класса (типа):

>>> A.\_\_dict\_\_['\_\_setattr\_\_']

<function \_\_setattr\_\_ at 0x7f699d22fa28>

Но дело в том, что он предназначен для экземпляров класса, а не для самого класса. Поэтому вызов A.\_\_setattr\_\_('foo', 'bar') будет неправильным. И именно поэтому setattr() должен делать явный поиск в классе (типе) объекта. Собственно, по этой же причине это сделано и для других магических методов \_\_add\_\_, \_\_len\_\_, \_\_getattr\_\_ и т.д.

Класс, как вызываемый (callable) тип

Класс (тип) — это вызываемый (callable) тип, и его вызов — это конструктор объекта.

>>> class C(object):

... pass

...

>>> С()

<\_\_main\_\_.C object at 0x1121e10>

Эквивалентно:

>>> type(C).\_\_call\_\_(C)

<\_\_main\_\_.C object at 0x1121ed0>

Т.к. C — обычный класс, то его метаклассом является type, поэтому будет использован вызов type(C).\_\_call\_\_(С) <==> type.\_\_call\_\_(С). Внутри type.\_\_call\_\_(C) уже происходит вызов C.\_\_new\_\_(cls, ...) и C.\_\_init\_\_(self, ...).

Важно то, что и \_\_new\_\_ и \_\_init\_\_ ищутся с помощью обычного алгоритма поиска атрибутов в классе. И при отсутствии их в C.\_\_dict\_\_, будут вызваны методы из родительского класса object: object.\_\_new\_\_ и object.\_\_init\_\_, в то время как метод \_\_call\_\_ — это метод класса (типа) объекта — type: type.\_\_call\_\_(C).

Singleton v.2

Зная это, создадим метаклассную реализацию синглтона.

Что нам нужно от синглтона? Чтобы вызов A() возвращал один и тот же объект.

A() <=> type(A).\_\_call\_\_(A)

Значит, нам нужно изменить поведение метода \_\_call\_\_, который определяется в метаклассе. Сделаем это, не забывая, что в общем случае в \_\_call\_\_ могут передаваться любые параметры.

>>> class SingletonMeta(type):

... def \_\_call\_\_(cls, \*args, \*\*kw):

... return super(SingletonMeta, cls).\_\_call\_\_(\*args, \*\*kw)

...

>>>

Заглушка готова.

Пусть единственный объект будет храниться в классовом атрибуте instance. Для этого инициализируем в cls.instance в \_\_init\_\_.

>>> class SingletonMeta(type):

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

... cls.instance = None

... def \_\_call\_\_(cls, \*args, \*\*kw):

... return super(SingletonMeta, cls).\_\_call\_\_(\*args, \*\*kw)

...

>>>

И вставим проверку в \_\_call\_\_:

>>> class SingletonMeta(type):

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

... cls.instance = None

... def \_\_call\_\_(cls, \*args, \*\*kw):

... if cls.instance is None:

... cls.instance = super(SingletonMeta, cls).\_\_call\_\_(\*args, \*\*kw)

... return cls.instance

...

>>> class C(object):

... \_\_metaclass\_\_ = SingletonMeta

...

Проверяем, что все работает как надо.

>>> C() is C()

True

>>> a = C()

>>> b = C()

>>> a.x = 42

>>> b.x

42

>>>

Вызываемый (callable) тип в качестве метакласса

Метаклассом может быть не только объект типа type, но и вообще любой вызываемый (callable) тип.

Достаточно просто создать функцию, в которой создается класс с помощью метакласса type.

>>> def mymeta(name, bases, attrs):

... attrs['foo'] = 'bar'

... return type(name, bases, attrs)

...

>>> class D(object):

... \_\_metaclass\_\_ = mymeta

...

>>> D()

<\_\_main\_\_.D object at 0x7fafa9abc090>

>>> d = D()

>>> d.foo

'bar'

>>> d.\_\_dict\_\_

{}

>>> D.\_\_dict\_\_

<dictproxy object at 0x7fafa9b297f8>

>>> dict(D.\_\_dict\_\_)

{'\_\_module\_\_': '\_\_main\_\_', '\_\_metaclass\_\_': <function mymeta at 0x7fafa9b3a9b0>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'D' objects>, 'foo': 'bar', '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'D' objects>, '\_\_doc\_\_': None}

Определения класса

Конструкция (statement) определения класса — это просто конструкция. Также как и любое statement оно может появляться где угодно в коде программы.

>>> if True:

... class A(object):

... def foo(self):

... print 42

...

>>> A

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

>>> A().foo()

42

>>>

В конструкции 'class' любые определенные «внутри» переменные, функции, классы, накапливаются в \_\_dict\_\_. А в определении можно использовать любые другие конструкции — циклы, if' :.

Поэтому можно делать так:

>>> class A(object):

... if 1 > 2:

... def foo(self):

... print '1>2'

... else:

... def bar(self):

... print 'else'

...

>>>

>>> A()

<\_\_main\_\_.A object at 0x7fafa9abc150>

>>> A().foo()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'A' object has no attribute 'foo'

>>> A().bar()

else

или так

>>> class A(object):

... if 1 > 2:

... x = 1

... def foo(self):

... print 'if'

... else:

... y = 1

... def bar(self):

... print 'else'

...

>>> A.x

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: type object 'A' has no attribute 'x'

>>> A.y

1

>>> A.foo

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: type object 'A' has no attribute 'foo'

>>> A.bar

<unbound method A.bar>

>>> A.bar()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: unbound method bar() must be called with A instance as first argument (got nothing instead)

>>> A().bar()

else

>>>

Можно вкладывать одно определение в другое.

>>> class A(object):

... class B(object):

... pass

...

...

>>> A()

<\_\_main\_\_.A object at 0x7fafa9abc2d0>

>>> A.\_\_dict\_\_

<dictproxy object at 0x7fafa9b340f8>

>>> dict(A.\_\_dict\_\_)

{'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'A' objects>, '\_\_module\_\_': '\_\_main\_\_', 'B': <class '\_\_main\_\_.B'>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'A' objects>, '\_\_doc\_\_': None}

>>> A.B()

<\_\_main\_\_.B object at 0x7fafa9abc310>

Или же динамически создавать методы класса:

>>> FIELDS=['a', 'b', 'c']

>>> class A(object):

... for f in FIELDS:

... locals()[f] = lambda self: 42

...

>>> a = A()

>>> a.a()

42

>>> a.b()

42

>>> a.c()

42

>>> a.d()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'A' object has no attribute 'd'

>>>

Хотя конечно такое наверняка крайне не рекомендуется делать в обычной практике, и лучше воспользоваться более идиоматичными средствами.

Name mangling

И еще про определения класса. Про name mangling.

Любой атрибут внутри определения класса classname вида ".\_\_{attr}" (attr при этом имеет не более одного \_ в конце) подменяется на "\_{classname}\_\_{attr}". Таким образом, внутри классов можно иметь «скрытые» приватные атрибуты, которые не «видны» наследникам и экземплярам класса.

>>> class A(object):

... \_\_private\_foo=1

...

>>> A.\_\_private\_foo

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: type object 'A' has no attribute '\_\_private\_foo'

Увидеть переменную можно так:

>>> A.\_A\_\_private\_foo

1

Ну и храниться она в \_\_dict\_\_ класса:

>>> dict(A.\_\_dict\_\_)

{'\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'A' objects>, '\_A\_\_private\_foo': 1, '\_\_module\_\_': '\_\_main\_\_', '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'A' objects>, '\_\_doc\_\_': None}

>>>

Наследники доступа не имеют:

>>> class B(A):

... def foo(self):

... print self.\_\_private\_foo

...

>>> B().foo()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in foo

AttributeError: 'B' object has no attribute '\_B\_\_private\_foo'

В принципе обеспечить доступ внешний доступ к атрибутам типа \_\_{attr} внутри определения класса, т.е. обойти name\_mangling, можно с помощью \_\_dict\_\_.

>>> class C(object):

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

... self.\_\_dict\_\_['\_\_value'] = 1

...

>>> C().\_\_value

1

>>>

Однако, такие вещи крайне не рекомендуется делать из-за того, что доступ к таким атрибутам будет невозможен внутри определения любого другого класса из-за подмены ".\_\_{attr}" на ".\_{classname}\_\_{attr}" вне зависимости к какому объекту или классу они относятся, т.е.

>>> class D(object):

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

... self.c = C().\_\_value

...

>>> D()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 3, in \_\_init\_\_

AttributeError: 'C' object has no attribute '\_D\_\_value'

>>> C().\_\_value

1

>>>

Хотя С().\_\_value прекрасно отработает вне определения класса. Чтобы обойти также придется использовать \_\_dict\_\_['\_\_value'].