Заметки об объектной системе языка Python ч.2

Python\*

Вторая часть заметок об объектной системе python'a (первая часть тут). В этой статье рассказывается, что такое классы, метаклассы, type, object и как происходит поиск атрибутов в классе.

Классы

Классы (типы) — это объектные фабрики. Их главная задача — создавать объекты, обладающие определенным поведением.

Классы определяют поведение объектов с помощью своих атрибутов (которые хранятся в \_\_dict\_\_ класса): методов, свойств, классовых переменные, дескрипторов, а также с помощью атрибутов, унаследованных от родительских классов.

Инстанцирование обычного объекта происходит в 2 этапа: сначала его создание, потом инициализация. Соответственно, сначала запускается метод класса \_\_new\_\_, который возвращает объект данного класса, потом выполняется метод класса \_\_init\_\_, который инициализирует уже созданный объект.

def \_\_new\_\_(cls, ...) — статический метод (но его можно таковым не объявлять), который создает объект класса cls.

def \_\_init\_\_(self, ...) — метод класса, который инициализирует созданный объект.

Например, объявим класс:

>>> class A(object):

... pass

...

>>>

Для класса A не определены ни \_\_new\_\_, ни \_\_init\_\_. В соответствии с алгоритмом поиска атрибутов для класса (типа), который не стоит путать с алгоритмом поиска атрибутов для обычных объектов, когда класс не найдет их в своем\_\_dict\_\_, он будет искать эти методы в \_\_dict\_\_ своих базовых (родительских) классах.

Класс А имеет в качестве родителя встроенный класс object. Таким образом он будет их искать в object.\_\_dict\_\_

И найдет:

>>> object.\_\_dict\_\_['\_\_init\_\_']

<slot wrapper '\_\_init\_\_' of 'object' objects>

>>> object.\_\_dict\_\_['\_\_new\_\_']

<built-in method \_\_new\_\_ of type object at 0x82e780>

>>>

Раз есть такие методы, значит, получается, что a = A() аналогичен последовательности вызовов:

a = object.\_\_new\_\_(A)

object.\_\_init\_\_(a)

В общем виде, используя super, который как раз и реализует алгоритм поиска атрибутов по родительским классам [1]:

a = super(A, A).\_\_new\_\_(A)

super(A, A).\_\_init\_\_(a)

Пример.

>>> class A(object):

... def \_\_new\_\_(cls):

... obj = super(A, cls).\_\_new\_\_(cls)

... print 'created object', obj

... return obj

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

... print 'initing object', self

...

>>> A()

created object <\_\_main\_\_.A object at 0x1620ed0>

initing object <\_\_main\_\_.A object at 0x1620ed0>

<\_\_main\_\_.A object at 0x1620ed0>

>>>

Singleton v.1

Понимая, как происходит создание объекта, можно написать реализацию паттерна одиночка.

Мы должны гарантировать, что у класса есть только один экземпляр. Т.е. при вызове конструктора класса, всегда возвращаем один и тот же экземпляр класса.

А это значит, что при вызов метода \_\_new\_\_ должен возвращать каждый раз один и тот же объект. Хранить сам объект можно, например, в классовой переменной instance.

В результате получаем:

>>> class C(object):

... instance = None

... def \_\_new\_\_(cls):

... if cls.instance is None:

... cls.instance = super(C, cls).\_\_new\_\_(cls)

... return cls.instance

...

>>> C() is C()

True

>>> C().x = 1

>>> c = C()

>>> d = C()

>>> c.x

1

>>> d.x

1

>>> c.x=2

>>> d.x

2

>>> c.x

2

Классы и метаклассы.

Для класса (типа), так же как и для обычного объекта, существует класс (тип), который создает классы и определяет поведение класса. Этот класс называется метаклассом.

Создание класса, как и обычного объекта происходит с помощью вызова конструктора, но т.к. в классе есть несколько дополнительных специальных атрибутов, которые должны быть инициализированы, в конструктор передаются и соответствующие обязательные параметры.

XClass = XMetaClass(name, bases, attrs)

Тогда, сразу после создания

XClass.\_\_name\_\_ равно name,

XClass.\_\_bases\_\_ равен bases,

XClass.\_\_dict\_\_ равен attrs, а

XClass.\_\_class\_\_ равен XMetaClass

По умолчанию для всех определяемых классов метаклассом является type.

Таким образом.

>>> class A(object):

... pass

...

Эквивалентно, по аналогии с обычными объектами:

>>> type('A', (object,), {})

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

А это:

>>> class B(A):

... def foo(self):

... 42

...

эквивалентно

>>> type('B', (A,), {'foo': lambda self: 42})

При определении класса, можно задать свой метакласс с помощью

классовой переменной \_\_metaclass\_\_:

>>> class A(object):

... \_\_metaclass\_\_ = Meta

...

>>>

Что равносильно: A = Meta('A', (object,), {})

О type и object

Прежде всего type и object — это объекты. И, как у всех порядочных объектов, у них есть специальные атрибуты \_\_class\_\_ и \_\_dict\_\_:

>>> object.\_\_class\_\_

<type 'type'>

>>> type.\_\_class\_\_

<type 'type'>

>>> object.\_\_dict\_\_

<dictproxy object at 0x7f7797a1cf30>

>>> type.\_\_dict\_\_

<dictproxy object at 0x7f7797a1cfa0>

Более того object, и type — это объекты типа (классы), и у них тоже есть специальные атрибуты \_\_name\_\_, \_\_bases\_\_\_:

>>> object.\_\_name\_\_

'object'

>>> type.\_\_name\_\_

'type'

>>> object.\_\_bases\_\_

()

>>> type.\_\_bases\_\_

(<type 'object'>,)

>>>

Экземпляры типа или класса object — это объекты (любые). Т.е. любой объект — экземпляр класса object:

>>> isinstance(1, object)

True

>>> isinstance(setattr, object)

True

>>> isinstance("foo", object)

True

>>> isinstance(A, object)

True

Даже функция является объектом:

>>> def bar():

... pass

...

>>> isinstance(bar, object)

True

Кроме того, класс object сам является своим экземпляром:

>>> isinstance(object, object)

True

type тоже является его экземпляром:

>>> isinstance(type, object)

True

Инстанцирование — object() возвращает самый простой и общий объект:

>>> o = object()

У которого даже \_\_dict\_\_ нет, есть только \_\_class\_\_.

>>> o.\_\_dict\_\_

Traceback (most recent call last):

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

AttributeError: 'object' object has no attribute '\_\_dict\_\_'

>>> o.\_\_class\_\_

<type 'object'>

>>>

Экземпляры класса или типа type — это только другие классы или другие типы:

Число — это не класс

>>> isinstance(1, type)

False

Строка тоже

>>> isinstance("foo", type)

False

Встроенная функция setattr тоже не класс.

>>> isinstance(setattr, type)

False

Класс — это класс.

>>> isinstance(A, type)

True

Тип строки — это класс.

>>> isinstance("foo".\_\_class\_\_, type)

True

Т.к. object и type — тоже классы, то они являются экземплярами класса type:

>>> isinstance(object, type)

True

>>> isinstance(type, type)

True

>>>

Т.к. множество классов (типов) являются подмножеством множества объектов, то логично предположить, что type является подклассом object, т.е.

>>> issubclass(type, object)

True

>>> issubclass(object, type)

False

type — это просто класс, экземплярами которого являются другие классы. (т.е. метакласс). А сами классы можно считать расширением простых, обычных объектов.

Таким образом, когда мы наследуем класс от object, этот класс автоматически наследует поведение класса object, т.е. при инстанцировании он будет возвращать обычный объект. А когда мы наследуем от класса type, мы также автоматически наследуем поведение класса type, т.е. при инстацированни будет создаваться класс. А класс, который создает класс, называется метаклассом.

Значит, чтобы определить просто класс, нужно наследовать его от object, чтобы определить метакласс — наследуем его от type.

И еще: не нужно путать type(a) и type(name, bases, attrs).

type(a) — вызов с одним аргументом, возвращает тип объекта,

a type(name, bases, attrs) — вызов с тремя аргументами — это вызов конструктора класса.

О поиске атрибутов в классе

Как уже было отмечено, алгоритм поиска атрибутов в обычном объекте, но есть некоторые тонкости, т.к. у типов (классов) есть \_\_bases\_\_ — родительские классы (типы).

Если атрибут есть в \_\_dict\_\_ возвращается он, затем идет поиск по базовым классам из \_\_bases\_\_, а потом идет обращение к \_\_dict\_\_ \_\_class\_\_'а (т.е. фактически метакласса) и его (метакласса) родительских классов (метаклассов).

Небольшой пример:

>>> class Ameta(type):

... def foo(cls):

... print 'Ameta.foo'

...

>>> class A(object):

... \_\_metaclass\_\_ = Ameta

...

>>> A.foo()

Ameta.foo

Все что определяется в метаклассе доступно для класса, но не доступно для экзмепляров класса — обычных объектов, т.к. поиск атрибутов в обычном объекте ведется только по \_\_dict\_\_ словарям класса.

>>> a = A()

>>> a.foo

Traceback (most recent call last):

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

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

В A.\_\_dict\_\_ 'foo' нет:

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

Traceback (most recent call last):

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

KeyError: 'foo'

Зато он есть в метаклассе, поэтому:

>>> A.foo()

Ameta.foo

>>> class B(A):

... @classmethod

... def foo(cls):

... print 'B.foo'

...

>>> B.foo # т.к. foo есть B.\_\_dict\_\_ вернется значение B.\_\_dict\_\_['foo']

<bound method Ameta.foo of <class '\_\_main\_\_.B'>>

>>> B.foo()

B.foo

>>> class C(B):

... pass

...

>>> C.foo() # вернет значение из базового класса B.

B.foo

Экземпляр класса C также вызовет метод foo из класса B.

>>> c= C()

>>> c.foo()

B.foo

>>> class D(A):

... pass

...

>>> D.foo()

Ameta.foo

А экземпляр D не найдет:

>>> d = D()

>>> d.foo()

Traceback (most recent call last):

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

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

Метаклассы

Метаклассы являются фабриками классов (или типов). Инстанцирование класса тоже проходит в 2 этапа — создание объекта типа (класса) и его инициализация. Это также делается с помощью двух методов метакласса. Сначала вызывается метод \_\_new\_\_ метакласса с параметрами, необходимыми для создания класса — name, bases, attrs, а потом \_\_init\_\_ с теми же параметрами и уже созданным классом.

Пример.

>>> class Meta(type):

... pass

...

>>> Meta('A', (object,), {})

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

В начале метакласс Meta ищет метод \_\_new\_\_ у себя в словаре \_\_dict\_\_, не находит его там и начинает искать в \_\_dict\_\_ своих родительских классах (т.е. метаклассах, в данном случае type), т.е. происходит обычный поиск атрибута в классе. В результате исполнения \_\_new\_\_ с соответствующими параметрами получает новый класс, который потом инициализируется вызовом \_\_init\_\_ метода метакласса.

В совсем развернутом виде получается:

cls = type.\_\_dict\_\_['\_\_new\_\_'](Meta, 'A', (object,), {})

type.\_\_dict\_\_['\_\_init\_\_'](cls, 'A', (object,), {})

Или с помощью super

cls = super(Meta, Meta).\_\_new\_\_(Meta, 'A', (object,), {})

super(Meta, Meta).\_\_init\_\_(cls, 'A', (object,), {})

Стоит отметить, что в отличие от инстанцирования обычных объектов, используется не object.\_\_new\_\_ и object.\_\_init\_\_, а type.\_\_new\_\_ и type.\_\_init\_\_. У object.\_\_new\_\_ и type.\_\_new\_\_ разные сигнатуры, и object.\_\_new\_\_ возвращает обычный объект (regular object), а type.\_\_new\_\_ — объект типа (typeobject), т.е. класс.

Посмотрим, как это все работает на примере.

>>> class Meta(type):

... def \_\_new\_\_(mcls, name, bases, attrs):

... print 'creating new class', name

... return super(Meta, mcls).\_\_new\_\_(mcls, name, bases, attrs)

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

... print 'initing new class', name

...

...

>>> class A(object):

... \_\_metaclass\_\_ = Meta

...

creating new class A

initing new class A

Во время инстанцирования просто объекта, никаких надписей не выводится.

>>> a = A()

>>>

Кроме того, соответственно, во методах \_\_new\_\_ и \_\_init\_\_ метакласса можно менять все: имя, список суперклассов, атрибуты.