Лекция 10: Классы 2

Сергей Лебедев

sergei.a.lebedev@gmail.com

16 ноября 2015 г.

Дескрипторы

- Механизм свойств в Python позволяет контролировать доступ, изменение и удаление атрибута.
- Очень безопасный класс запрещает неотрицательные значения для атрибута `x`:

```
class VerySafe:
    def _get_attr(self):
        return self._x

    def _set_attr(self, x):
        assert x > 0, "non-negative value required"
        self._x = x

    def _del_attr(self):
        del self._x

    x = property(_get_attr, _set_attr, _del_attr)
```

Проверим, что всё работает:

```
>>> very_safe = VerySafe()
>>> very_safe.x = 42
>>> very_safe.x
42
>>> very_safe.x = -42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in _set_attr
AssertionError: non-negative value required
>>> very_safe.y = -42
>>> very_safe.y
-42
```

Вопрос

Как добавить аналогичные проверки для атрибута `y`?

- Дескриптор — это:
  - экземпляр класса, реализующего протокол дескрипторов,
  - свойство, которое можно переиспользовать[1].
- Пример: дескриптор `NonNegative`, который делает то же самое, что написанное нами ранее свойство.

```
class NonNegative:
    def __get__(self, instance, owner):
        return magically_get_value(...)

    def __set__(self, instance, value):
        assert value >= 0, "non-negative value required"
        magically_set_value(...)

    def __delete__(self, instance):
        magically_delete_value(...)
```

[1] https://bit.ly/descriptors-demystified

```
>>> class VerySafe:
...     x = NonNegative()
...     y = NonNegative()
...
>>> very_safe = VerySafe()
>>> very_safe.x = 42
>>> very_safe.x
42
>>> very_safe.x = -42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __set__
AssertionError: non-negative value required
>>> very_safe.y = -42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __set__
AssertionError: non-negative value required
```

Протокол дескрипторов: `__get__`

- Метод `__get__` вызывается при доступе к атрибуту.
- Метод принимает два аргумента:
  - `instance` — экземпляр класса или **None**, если дескриптор был вызван в результате обращения к атрибуту у класса
  - `owner` — класс, “владеющий” дескриптором.
- Пример:

```
>>> class Descr:
...     def __get__(self, instance, owner):
...         print(instance, owner)
...
>>> class A:
...     attr = Descr()
...
>>> A.attr
None <class '__main__.A>
>>> A().attr
<__main__.A object at [...]> <class '__main__.A>
```

Протокол дескрипторов: __get__ и owner

```
>>> class A:
...     attr = Descr()
...
>>> class B(A):
...     pass
...
>>> A.attr
None <class '__main__.A'>
>>> A().attr
<__main__.A object at [...]> <class '__main__.A'>
>>> B.attr
None <class '__main__.B'>
>>> B().attr
<__main__.B object at [...]> <class '__main__.B'>
```

Протокол дескрипторов: `__set__`

- Метод `__set__` вызывается для изменения значения атрибута.
- Метод принимает два аргумента:
  - `instance`, экземпляр класса, “владеющего” дескриптором,
  - `value` — новое значение атрибута.
- Пример:

```
>>> class Descr:
...     def __set__(self, instance, value):
...         print(instance, value)
...
>>> class A:
...     attr = Descr()
...
>>> instance = A()
>>> instance.attr = 42
<__main__.A object at [...]> 42
>>> A.attr = 42
???
```

Протокол дескрипторов: `__delete__`

- Метод `__delete__` вызывается при удалении атрибута.
- Метод принимает один аргумент — экземпляр класса, “владеющего” дескриптором.
- Пример:

```
>>> class Descr:
...     def __delete__(self, instance):
...         print(instance)
...
>>> class A:
...     attr = Descr()
...
>>> del A().x
<__main__.A object at [...]>
>>> del A.x
???
```

Вопрос

Почему метод называется `__delete__`, а не `__del__`?

“Семантика” протокола дескрипторов

- Пусть
  - `instance` — экземпляр класса `cls`,
  - атрибут `attr` которого — дескриптор, и
  - `descr = cls.__dict__["attr"]` — непосредственно сам дескриптор.
- Тогда

```
cls.attr                 descr.__get__(None, cls)
instance.attr           ≈descr.__get__(instance, cls)
instance.attr = value    descr.__set__(instance, value)
del instance.attr        descr.__delete__(instance)
```

- Всё то же самое справедливо для ситуации, когда дескриптор объявлен где-то в иерархии наследования.

- Дескриптор может определять любое сочетание методов `__get__`, `__set__` и `__delete__`.
- Все дескрипторы можно поделить на две группы:
  - дескрипторы данных *aka* data descriptors, определяющие как минимум метод `__set__`, и
  - остальные *aka* non-data descriptors.
- Полезные дескрипторы определяют ещё и метод `__get__`.
- Отличие между группами в том, как они взаимодействуют с `__dict__` экземпляра.

- Пример:

```
class A:
    attr = Descr()


A().attr
```

- Обращение к атрибуту `attr` будет перенаправлено к методу `Descr.__get__` если:
  1. `Descr` — это дескриптор данных, реализующий метод `__get__`, или
  2. `Descr` — это дескриптор, реализующий только метод `__get__`, и в `__dict__` экземпляра нет атрибута `attr`.
- Во всех остальных случаях сработает стандартная машинерия поиска атрибута: сначала в `__dict__` экземпляра, затем в `__dict__` класса и рекурсивно во всех родительских классах.

Пример: дескриптор данных с методом __get__

```
>>> class Descr:
...     def __get__(self, instance, owner):
...         print("Descr.__get__")
...
...     def __set__(self, instance, value):
...         print("Descr.__set__")
...
>>> class A:
...     attr = Descr()
...
>>> instance = A()
>>> instance.attr
Descr.__get__
>>> instance.__dict__["attr"] = 42
>>> instance.attr
Descr.__get__
```

Пример: дескриптор с единственным методом __get__

```
>>> class Descr:
...     def __get__(self, instance, owner):
...         print("Descr.__get__")
...
>>> class A:
...     attr = Descr()
...
>>> instance = A()
>>> instance.attr
Descr.__get__
>>> instance.__dict__["attr"] = 42
>>> instance.attr
42
```

```
class Proxy:
    def __get__(self, instance, owner):
        # вернём значение атрибута для
        # переданного экземпляра.

    def __set__(self, instance, value):
        # сохраним новое значение атрибута
        # для переданного экземпляра.

    def __delete__(self, instance)
        # удалим значение атрибута для переданного
        # экземпляра.
```

Вопрос

Как реализовать методы дескриптора `Proxy` так, чтобы значение атрибута было специфично для переданного экземпляра?

Хранение данных в дескрипторах: атрибут дескриптора

- Можно хранить данные в атрибутах самого дескриптора:

```
class Proxy:
    def __get__(self, instance, owner):
        return self.value

    def __set__(self, instance, value):
        self.value = value

    # __delete__ аналогично
```

- Как это будет работать в случае нескольких экземпляров класса Something?

```
>>> class Something:
...     attr = Proxy()
...
>>> some = Something()
>>> some.attr = 42
>>> other = Something()
>>> other.attr
???
```

Хранение данных в дескрипторах: словарь

- Можно хранить данные для каждого экземпляра в словаре:

```
class Proxy:
    def __init__(self):
        self.data = {}

    def __get__(self, instance, owner):
        if instance is None:
            return self
        if instance not in self.data:
            raise AttributeError
        return self.data[instance]

    def __set__(self, instance, value):
        self.data[instance] = value

    # __delete__ аналогично
```

- Очевидный минус такого подхода — экземпляр должен быть *hashable*, но есть и ещё одна проблема, какая?

Хранение данных в дескрипторах: атрибут экземпляра

- Наименьшее из зол — хранить данные непосредственно в самом экземпляре:

```
class Proxy:
    def __init__(self, label):
        self.label = label  # метка дескриптора

    def __get__(self, instance, owner):
        # ...
        return instance.__dict__[self.label]

    # __set__ и __delete__ аналогично
```

- Пример использования:

```
>>> class Something:
...     attr = Proxy("attr")
...
>>> some = Something()
>>> some.attr = 42
>>> some.attr
42
```

Примеры дескрипторов: @property

```
class property:
    def __init__(self, get=None, set=None, delete=None):
        self._get = get
        self._set = set
        self._delete = delete

    def __get__(self, instance, owner):
        if self._get is None:
            raise AttributeError("unreadable attribute")
        return self._get(instance)

    # __set__ и __delete__ аналогично


class Something:
    @property
    def attr(self):
        return 42
```

- Напоминание: методы — это обычные функции, объявленные в теле класса.

```
>>> class Something:
...     def do_something(self):
...         pass
...
>>> Something.do_something
<function Something.do_something at [...]>
>>> Something().do_something
<bound method Something.do_something of [...]>
```

- Как это работает? Дескрипторы![2]

```
from types import MethodType

class Function:
    def __get__(self, instance, owner):
        if instance is None:
            return self
        else:
            return MethodType(self, instance, owner)
```

[2] https://docs.python.org/3/howto/descriptor.html

- Декоратор `staticmethod` позволяет объявить статический метод, то есть просто функцию, внутри класса:

```
>>> class SomeClass:
...     @staticmethod
...     def do_something():
...         pass
...
>>> SomeClass.do_something()
```

- Для объявления методов класса используется похожий декоратор `classmethod`. Первый аргумент метода класса — непосредственно сам класс, а не его экземпляр.

```
>>> class Settings:
...     @classmethod
...     def read_from(cls, path):
...         return cls()  # noop
...
>>> Settings.read_from("./settings.ini")
<__main__.Settings at 0x10f558208>
```

Примеры дескрипторов: @staticmethod и @classmethod

```
>>> class staticmethod:
...     def __init__(self, method):
...         self._method = method
...
...     def __get__(self, instance, owner):
...         return self._method
...
>>> class Something:
...     @staticmethod
...     def do_something():
...         print("I'm busy, alright?")
...
>>> Something().do_something()
I'm busy, alright?
```

Вопрос

Как будет выглядеть декоратор `classmethod`, реализованный через механизм дескрипторов?

- Как и свойства, дескрипторы позволяют контролировать чтение, изменение и удаление атрибута, но, в отличие от свойств, дескрипторы можно переиспользовать.
- Дескриптор — это экземпляр класса, реализующего любую комбинацию методов `__get__`, `__set__` и `__delete__`.
- Мы поговорили про:
  - использование дескрипторов,
  - семантику протокола дескрипторов,
  - подводные камни при написании собственных дескрипторов,
  - декораторы `staticmethod` и `classmethod`.

Метаклассы

- Все классы в Python — это экземпляры класса `type`.

```
>>> class Something:
...     attr = 42
...
>>> Something
<class '__main__.Something'>
>>> type(Something)
<class 'type'>
```

- Чтобы избежать путаницы с обычными классами, класс `type` называют *метаклассом*, то есть классом, экземпляры которого тоже классы.

```
>>> name, bases, attrs = "Something", (), {"attr": 42}
>>> Something = type(name, bases, attrs)
>>> Something
<class '__main__.Something'>
>>> some = Something()
>>> some.attr
42
```

При объявлении класса можно указать для него метакласс, отличный от `type`:

```
>>> class Meta(type):
...     def some_method(cls):
...         return "foobar"
...
>>> class Something(metaclass=Meta):
...     attr = 42
...
>>> type(Something)
<class '__main__.Meta'>
>>> Something.some_method
<bound method Meta.some_method of <class '[...].Something'>>
>>> Something().some_method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Something' object has no attribute [...]
```

- Создание классов в Python — многоэтапный процесс.

```
class Something(Base, metaclass=Meta):
    def __init__(self, attr):
        self.attr = attr

    def do_something(self):
        pass
```

- Первым делом определим метакласс.
  - Для класса `Something` всё просто: метакласс указан явно.
  - В общем случае интерпретатору пришлось бы обойти все родительские классы и поинтересоваться их метаклассом.
  - Метакласс по умолчанию для всех классов — `type`.

Как создаются классы: `__dict__` и `exec`

- Подготовим `__dict__` для будущего класса.
  - По умолчанию это просто словарь, но метакласс может изменить такое поведение, определив метод класса `__prepare__`.
  - Для класса `Something`:
    `clsdict = Meta.__prepare__("Something", (Base, ))`
- Вычислим тело класса, используя `clsdict` для хранения локальных переменных:

```
body = """
def __init__(self, attr):
    self.attr = attr

def do_something(self):
    pass
"""

exec(body, globals(), clsdict)
```

- После вызова `exec`, словарь `clsdict` содержит все методы и атрибуты класса.

- Создадим объект класса, вызвав метакласс с тремя аргументами:
  - `name` — имя класса,
  - `bases` — кортеж родительских классов,
  - `clsdict` — словарь атрибутов и методов класса.

```
Something = Meta("Something", (Base, ), clsdict)
```

- Более подробное описание с множеством примеров можно найти в PEP-3115[3].

[3] https://www.python.org/dev/peps/pep-3115

- Кроме метода `__init__`, который инициализирует уже созданный экземпляр, у каждого класса в Python есть метод `__new__`.
- Метод `__new__` создаёт экземпляр до инициализации.

```
>>> class Noop:
...     def __new__(cls, *args, **kwargs):
...         print("Creating instance with {} and {}"
...               .format(args, kwargs))
...         instance = super().__new__(cls)  # self
...         return instance
...
...     def __init__(self, *args, **kwargs):
...         print("Initializing with {} and {}"
...               .format(args, kwargs))
...
>>> noop = Noop(42, attr="value")
Creating instance with (42,) and {'attr': 'value'}
Initializing with (42,) and {'attr': 'value'}
```

Пример бесполезного метакласса[4]

```
>>> class UselessMeta(type):
...     def __new__(metacls, name, bases, clsdict):
...         print(type(clsdict))
...         print(list(clsdict))
...         cls = super().__new__(metacls, name, bases,
...                               clsdict)
...         return cls
...
...     @classmethod
...     def __prepare__(metacls, name, bases):
...         return OrderedDict()
...
>>> class Something(metaclass=UselessMeta):
...      attr = "foo"
...      other_attr = "bar"
...
<class 'collections.OrderedDict'>
['__module__', '__qualname__', 'attr', 'other_attr']
```

[4] Примеры **полезных** метаклассов: `enum.EnumMeta`, `abc.ABCMeta`. Ещё один пример будет в домашнем задании

- Области применения декораторов классов и метаклассов пересекаются: и те, и другие используются для изменения поведения классов.
- При этом метаклассы
  - могут *временно* подменять тип `__dict__`,
  - сохраняются при наследовании.
- Пример:

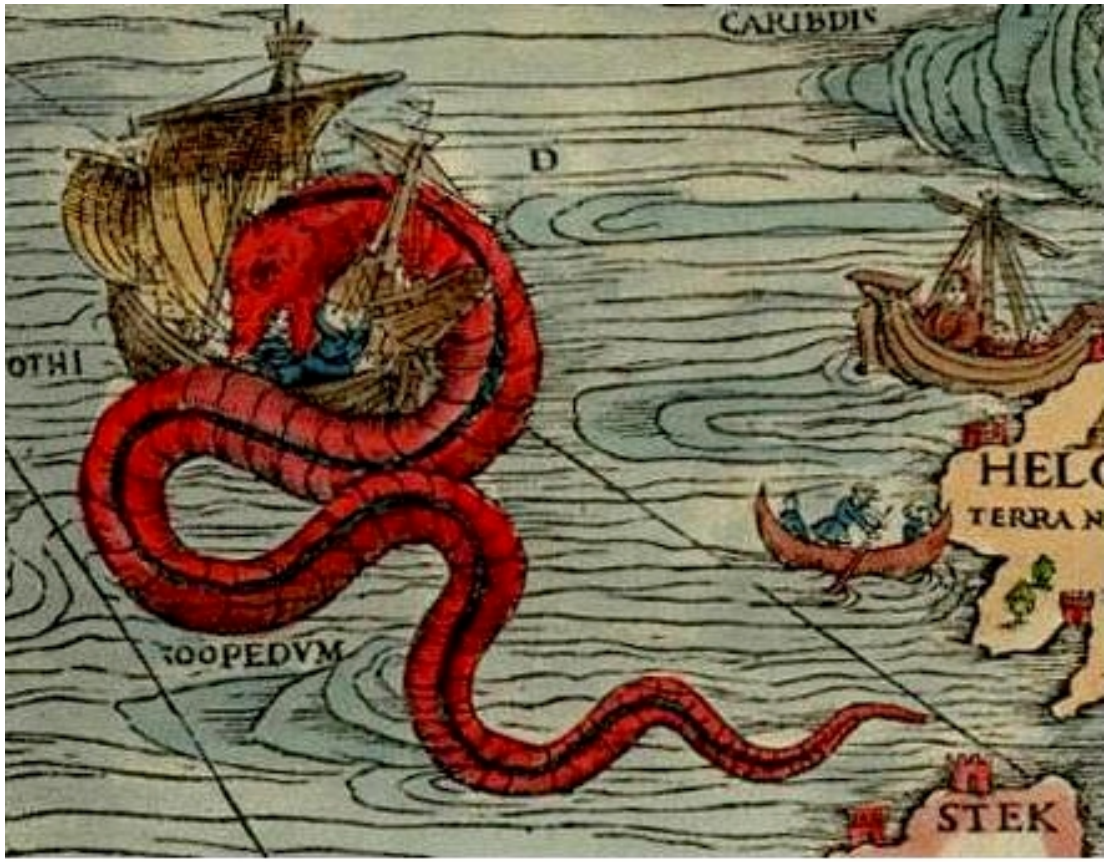
```
class Base(metaclass=Meta):
    pass

class Something(Base):
    pass  # имеет метакласс
          # Meta, а не type
```

```
@meta
class Base:
    pass

# vvv нужно явно
#     декорировать
#     каждый раз
@meta
class Something:
    pass
```

Метаклассы: резюме[5]



[5] https://blogs.cisco.com/manufacturing/here-be-dragons

Модуль abc

- Модуль `abc` содержит метакласс `ABCMeta`, который позволяет объявлять **абстрактные базовые классы** *aka* ABC.
- Класс считается абстрактным, если:
  - его метакласс — `ABCMeta`,
  - хотя бы один из абстрактных методов не имеет конкретной реализации.

```
>>> import abc
>>> class Iterable(metaclass=abc.ABCMeta):
...     @abc.abstractmethod
...     def __iter__(self):
...         pass
...
>>> class Something(Iterable):
...     pass
...
>>> Something()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Something \
           with abstract methods __iter__
```

```
>>> class MemorizingDict(dict):
...     def __init__(self, *args, **kwargs):
...         super().__init__(*args, **kwargs)
...         self._history = deque(maxlen=10)
...
...     def __setitem__(self, key, value):
...         self._history.append(key)
...         super().__setitem__(key, value)
...
...     def get_history(self):
...         return self._history
...
>>> d = MemorizingDict({"foo": 42})
>>> d.setdefault("bar", 24)
24
>>> d["baz"] = 100500
>>> print(d.get_history())
???
```

- В CPython `list`, `dict`, `set` и др. — это структуры языка C, к которым можно обращаться через конструкции языка Python.
- Они не предполагают расширение: `dict` — это не какой-нибудь словарь, описанный в терминах `__getitem__`, `__setitem__` и др., а вполне конкретная его реализация.
- Поэтому для `dict` перегруженный метод `MemorizingDict.__setitem__` не существует:
  - он **не будет** вызван в конструкторе при инициализации словаря,

    ```
    >>> d = MemorizingDict({"foo": 42})
    ```

  - и в методе `setdefault`.

    ```
    >>> d.setdefault("bar", 24)
    24
    ```

- Модуль `collections.abc` содержит абстрактные базовые классы для коллекций на все случаи жизни.
- Например, чтобы реализовать `MemorizingDict`, нужно унаследовать его от `MutableMapping` и реализовать пять методов:
  - `__getitem__`, `__setitem__`, `__delitem__`,
  - `__iter__` и
  - `__len__`.
- `MutableMapping` выражает все остальные методы `dict` в терминах этих пяти абстрактных методов.

Ещё немного “магических” методов

```
instance[key]           instance.__getitem__(key)
instance[key] = value   instance.__setitem__(key, value)
del instance[key]       instance.__delitem__(key)
```

Модуль `collections.abc` и встроенные коллекции

- Все встроенные коллекции являются наследниками ABC из модуля `collections.abc`:

```
>>> from collections import abc
>>> issubclass(list, abc.Sequence)
True
>>> isinstance({}, abc.Hashable)
False
```

- Это позволяет компактно проверять наличие у экземпляра необходимых методов:

```
>>> def flatten(obj):
...     for item in obj:
...         if isinstance(item, abc.Iterable):  # ?
...             yield from flatten(item)
...         else:
...             yield item
...
>>> list(flatten([[1, 2], 3, [], [4]]))
[1, 2, 3, 4]
```

- Модуль `abc` позволяет классам в Python объявлять абстрактные методы.
- Мы также поговорили про модуль `collections.abc` и его использование для:
  - написания своих коллекций и
  - проверки наличия методов у объекта.