Лекция 11
Метаклассы

27 апреля 2017 г.

> Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).

Python Guru Tim Peters

Классы и метаклассы

Класс — объект, создающий другие объекты.

```
x = int
print x
y = x()
print y
```

```
<type 'int'>
0
```

Классы и метаклассы

Класс — объект, создающий другие объекты.

```
x = int
print x
y = x()
print y
```

```
<type 'int'>
0
```

Метакласс — объект, создающий классы.

Метаметаклассы...?

```
def f():
    return f

f
f()
f()()
f()()()
```

```
<function f at 0x7f99a456e5f0>
<function f at 0x7f99a456e5f0>
<function f at 0x7f99a456e5f0>
<function f at 0x7f99a456e5f0>
```

Пример (Django)

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=70)
```

Пример (Django)

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=70)

s = Student(name='Alexey')
print s.name
```

Пример (Django)

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=70)


s = Student(name='Alexey')
print s.name
```

Экземпляр `CharField`?

Пример (Django)

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=70)


s = Student(name='Alexey')
print s.name
```

Экземпляр `CharField`?

```
'Alexey'
```

Пример (Django)

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=70)

s = Student(name='Alexey')
print s.name
```

Экземпляр `CharField`?

```
'Alexey'
```

(Скрытые обращения к базе данных)

Метакласс `type`

`type(name, bases, attrs)`

- Возвращает класс с именем `name`,
- родителями `bases`,
- атрибутами `attrs`.

Метакласс `type`

`type(name, bases, attrs)`

- Возвращает класс с именем `name`,
- родителями `bases`,
- атрибутами `attrs`.

```
C = type('MyClass',
         (object,),
         {'f': lambda self: 'abc'})

c = C()
print c.f()
```

```
abc
```

Универсальный метакласс

```
x = 1
print type(x)
```

```
<type 'int'>
```

Универсальный метакласс

```
x = 1
print type(x)
```

```
<type 'int'>
```

```
# type(type(x))
print type(int)
```

```
<type 'type'>
```

Универсальный метакласс

```
x = 1
print type(x)
```

```
<type 'int'>
```

```
# type(type(x))
print type(int)
```

```
<type 'type'>
```

```
# type(type(type(x)))
print type(type)
```

```
<type 'type'>
```

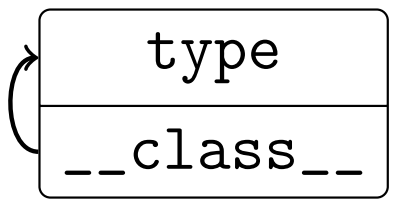
Атрибут `__class__`

(класс $\approx$ тип)

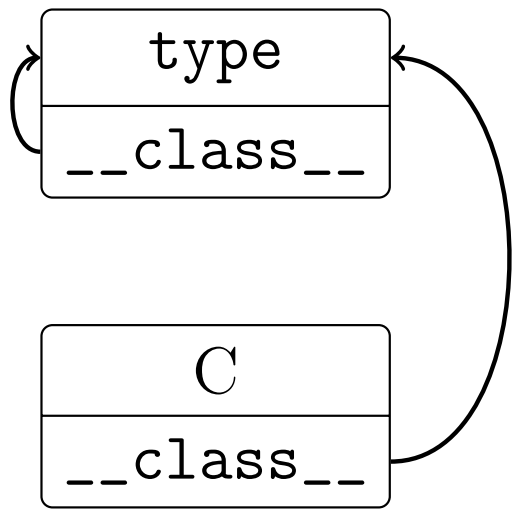
```
class C(object):
    pass

x = C()
print x.__class__
print C.__class__
print type.__class__
```

```
<class '__main__.C'>
<type 'type'>
<type 'type'>
```

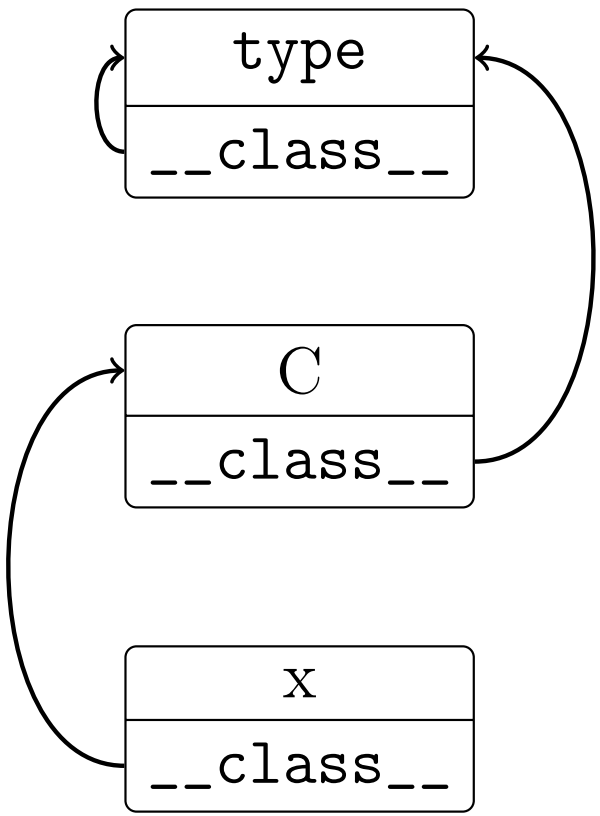
Иерархия объектов



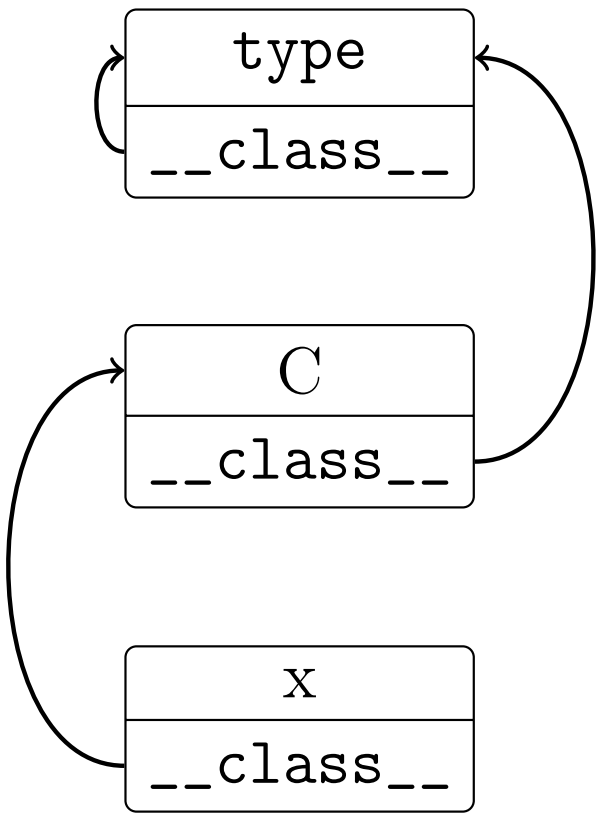
Иерархия объектов



Иерархия объектов

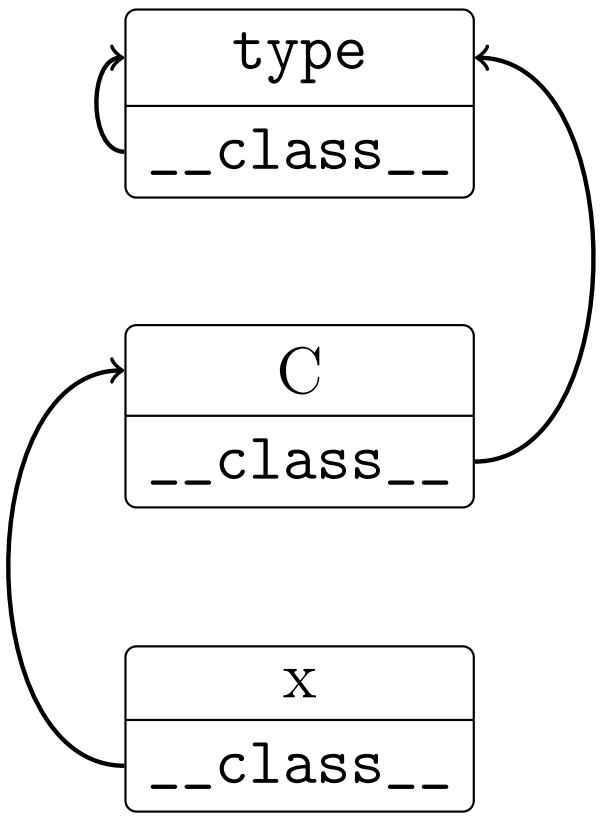


Иерархия объектов



- Объект

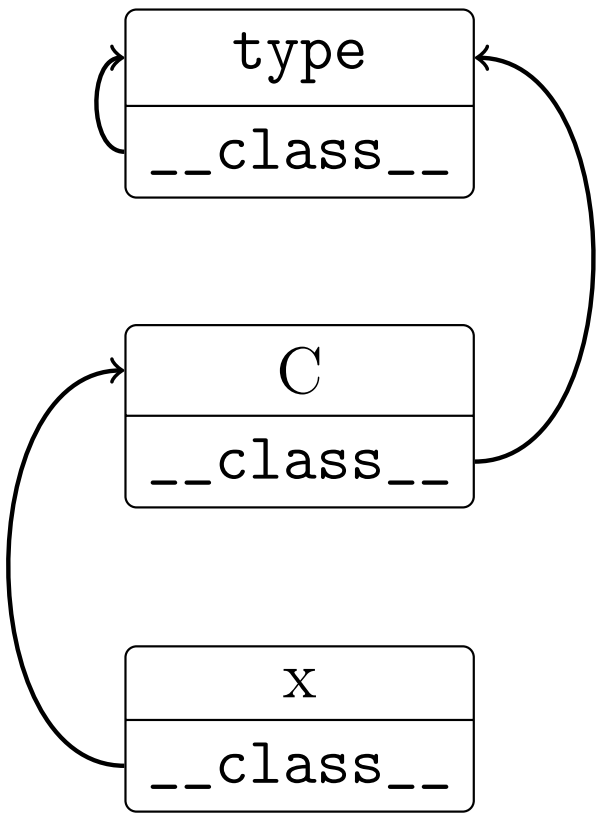
Иерархия объектов



- Класс
- Объект

- Объект

Иерархия объектов



- Метакласс
- Класс
- Объект

- Класс
- Объект

- Объект

Механизм создания классов

```
class C(object):
    def f(self):
        print 'abc'
    X = 1
    ...
```

Механизм создания классов

```
class C(object):
    def f(self):
        print 'abc'
    X = 1
    ...
```

- Создаются `f`, `X`, ... (рекурсивно)

Механизм создания классов

```
class C(object):
    def f(self):
        print 'abc'
    X = 1
    ...
```

- Создаются `f`, `X`, ... (рекурсивно)
- Вызывается
  `C = type('C', (object,), {'f': f, 'X': X, ...})`

Механизм создания классов

```
class C(object):
    def f(self):
        print 'abc'
    X = 1
    ...
```

- Создаются `f`, `X`, ... (рекурсивно)
- Вызывается
  `C = type('C', (object,), {'f': f, 'X': X, ...})`
- (`type` делает все внутренние регистрации и т.п.)

Механизм создания классов

```
class C(object):
    def f(self):
        print 'abc'
    X = 1
    ...
```

- Создаются `f`, `X`, ... (рекурсивно)
- Вызывается
  `C = type('C', (object,), {'f': f, 'X': X, ...})`
- (`type` делает все внутренние регистрации и т.п.)
- `type` — метакласс по умолчанию

Изменение метакласса

```
class A(object): pass

def creator(classname, bases, attrs):
    return A

class C(object):
    __metaclass__ = creator
    def f(self): print 'hi'
```

Изменение метакласса

```
class A(object): pass

def creator(classname, bases, attrs):
    return A

class C(object):
    __metaclass__ = creator
    def f(self): print 'hi'

print C
```

```
<class '__main__.A'>
```

Изменение метакласса

```
class A(object): pass

def creator(classname, bases, attrs):
    return A

class C(object):
    __metaclass__ = creator
    def f(self): print 'hi'

print C
```

```
<class '__main__.A'>
```

```
c = C()
c.f()
```

```
AttributeError: 'A' object has no attribute 'f'
```

Метаклассы в общем виде

Особые функции: `__call__`, `__init__` (и `__new__`).

Метаклассы в общем виде

Особые функции: `__call__`, `__init__` (и `__new__`).

```
class Meta(type):
    def __init__(cls, name, base, attrs):
        super(Meta, cls).__init__(
            name, base, attrs)
        cls.f = lambda self: 'qwerty'

class A(object):
    __metaclass__ = Meta
```

Метаклассы в общем виде

Особые функции: `__call__`, `__init__` (и `__new__`).

```
class Meta(type):
    def __init__(cls, name, base, attrs):
        super(Meta, cls).__init__(
            name, base, attrs)
        cls.f = lambda self: 'qwerty'

class A(object):
    __metaclass__ = Meta

a = A()
print a.f()
```

```
qwerty
```

Процесс создания класса, `type`

```
C = type(name, bases, attrs)
```

Процесс создания класса, `type`

```
C = type(name, bases, attrs)
```

```
C = type.__call__(name, bases, attrs)
```

- `C = type.__new__(metacls, name, bases, attrs)` (конструирование)
- `type.__init__(C, name, bases, attrs)` (инициализация)

Процесс создания класса, метакласс

```
C = Meta(name, bases, attrs)
```

```
C = Meta.__call__(name, bases, attrs)
```

- `C = Meta.__new__(Meta, name, bases, attrs)` (конструирование)
- `Meta.__init__(C, name, bases, attrs)` (инициализация)

Метаклассы, наследование, разрешение имен

- Метаклассы наследуются от `type`

Метаклассы, наследование, разрешение имен

- Метаклассы наследуются от `type`
- При наследовании классов метакласс сохраняется

Метаклассы, наследование, разрешение имен

- Метаклассы наследуются от `type`
- При наследовании классов метакласс сохраняется
- Разрешение имен в классах: сам класс и его метакласс (с учетом наследования)

Метаклассы, наследование, разрешение имен

- Метаклассы наследуются от `type`
- При наследовании классов метакласс сохраняется
- Разрешение имен в классах: сам класс и его метакласс (с учетом наследования)
- Разрешение имен в экземплярах: экземпляр и его класс (с учетом наследования)

Метаклассы, наследование, разрешение имен

- Метаклассы наследуются от `type`
- При наследовании классов метакласс сохраняется
- Разрешение имен в классах: сам класс и его метакласс (с учетом наследования)
- Разрешение имен в экземплярах: экземпляр и его класс (с учетом наследования)

```
class M(type):
    def f(cls): pass
class C(object):
    __metaclass__ = M
c = C()
c.f()
```

```
AttributeError: 'C' object has no attribute 'f'
```

```
C.f()
```

Пример метакласса

Задача: для каждого метода в классе добавлять для него короткий псевдоним (первая буква в верхнем регистре).

```
class A(object):
    __metaclass__ = AddShortNames
    def foo(self):
        print('Called foo')
    def bar(self):
        print('Called bar')

a = A()
a.foo()
a.bar()
a.F()
a.B()
```

Пример метакласса

```
class AddShortNames(type):
    def __init__(cls, classname, bases, attrs):
        super(AddShortNames, cls).__init__(
            classname, bases, attrs)
        for attr_name, attr in attrs.items():
            if callable(attrs):
                short = attr_name[0].upper()
                setattr(cls, short, attr)
```

Абстрактные атрибуты

Запрет на создание экземпляров классов-потомков, которые не переопределяют какие-то атрибуты.

Абстрактные атрибуты

Запрет на создание экземпляров классов-потомков, которые не переопределяют какие-то атрибуты.

```
from abc import ABCMeta, abstractmethod

class Shape(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def area(self):
        pass
```

Абстрактные атрибуты

```
class Rect(Shape):
    pass

t = Rect()
```

```
TypeError: Can't instantiate abstract class
Square with abstract methods area
```

Абстрактные атрибуты

```
class Rect(Shape):
    pass

t = Rect()
```

```
TypeError: Can't instantiate abstract class
Square with abstract methods area
```

```
class Rect(Shape):
    ...
    def area(self):
        return self.w * self.h

t = Rect()
```

Ничего не понятно, что делать?

Материалы про метаклассы:

- Ссылка на статью на хабре.
- М. Лутц. Изучаем Python (4-е издание). Гл. 39.