Лекция 3
Объектно-ориентированное программирование в Python

25 февраля 2016 г.

Пользовательские классы

Классы

Класс — тип данных, представляющий модель какой-то сущности.

Объект — конкретная реализация какого-то класса (*экземпляр*).

Класс `int` — тип данных, моделирующий целые числа. `1`, `2`, `15` — объекты этого класса.

Минимальный класс в Python

```
class Empty:
    pass
```

Минимальный класс в Python

```
class Empty:
    pass
```

```
>>> x = Empty()
>>> x
<__main__.Empty instance at ...>
```

Методы

```
class Greeter:
    def greet(self):
        print "hi!"
```

Методы

```
class Greeter:
    def greet(self):
        print "hi!"

>>> x = Greeter()
>>> x
<__main__.Greeter instance at ...>
>>> x.greet()
hi!
```

Конструктор и атрибуты объектов

```
class NamedGreeter:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print "hi, my name is", self.name
```

Конструктор и атрибуты объектов

```
class NamedGreeter:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print "hi, my name is", self.name

x = NamedGreeter('Guido')
x.greet()
```

```
hi, my name is Guido
```

Создание атрибутов

```
class CreativeGreeter:
    def invent_name(self):
        self.name = "Tom"
    def greet(self):
        print "hi, my name is", self.name
```

Создание атрибутов

```
class CreativeGreeter:
    def invent_name(self):
        self.name = "Tom"
    def greet(self):
        print "hi, my name is", self.name

x = CreativeGreeter()
x.greet()
```

```
...
AttributeError: CreativeGreeter instance
has no attribute 'name'
```

Создание атрибутов

```
class CreativeGreeter:
    def invent_name(self):
        self.name = "Tom"
    def greet(self):
        print "hi, my name is", self.name

x = CreativeGreeter()
x.invent_name()
x.greet()
```

```
hi, my name is Tom
```

Атрибуты классов

```
class NamedGreeter2:
    def __init__(self, name):
        self.name = name
    PREFIX = "hi, my name is"
    def greet(self):
        return self.PREFIX, self.name
```

Атрибуты классов

```
class NamedGreeter2:
    def __init__(self, name):
        self.name = name
    PREFIX = "hi, my name is"
    def greet(self):
        return self.PREFIX, self.name
```

```
>>> NamedGreeter2.PREFIX
'hi, my name is'
>>> x = NamedGreeter2('Guido')
>>> x.PREFIX
'hi, my name is'
>>> x.greet()
'hi, my name is Guido'
```

Объекты в Python

Все является объектами

```
>>> x = int
>>> x
<type 'int'>
>>> x(3.5)
3
```

Все является объектами

```
>>> x = int
>>> x
<type 'int'>
>>> x(3.5)
3

>>> y = 'abc'.split
>>> y('b')
['a', 'c']
```

Все является объектами

```
>>> d = {}
>>> d[int] = 'test'
>>> d[str] = 1
>>> d
{<type 'int'>: 'test', <type 'str'>: 1}
```

Все является объектами

```
>>> d = {}
>>> d[int] = 'test'
>>> d[str] = 1
>>> d
{<type 'int'>: 'test', <type 'str'>: 1}

>>> def f():
        pass
>>> l = [f, len, lambda x: x + 1]
>>> l
[<function f at ...>,
 <built-in function len>,
 <function <lambda> at ...>]
```

Классы и их экземпляры

```
>>> type(1)
<type 'int'>
```

Классы и их экземпляры

```
>>> type(1)
<type 'int'>

>>> type(int)
<type 'type'>
```

Классы и их экземпляры

```
>>> type(1)
<type 'int'>

>>> type(int)
<type 'type'>

>>> type(type)
<type 'type'>
```

(Подробнее — в теме “Метаклассы”)

Атрибуты

Функции и методы

- Функция — объект, который можно “вызвать”, используя круглые скобки.
- Метод — атрибут-функция, которая принимает экземпляр класса в качестве первого аргумента.

Атрибуты и методы

```
def method(self):
    self.x = 1
class MyClass:
    f = method
```

Атрибуты и методы

```
def method(self):
    self.x = 1
class MyClass:
    f = method

>>> c = MyClass()
>>> c.x
...
AttributeError: MyClass instance has no
   attribute 'x'
>>> c.f()
>>> c.x
1
```

Доступ к атрибутам

```
class NamedGreeter:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print "hi, my name is", self.name

x = NamedGreeter('Guido')
print x.name
```

```
Guido
```

Доступ к атрибутам

```
class NamedGreeter:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print "hi, my name is", self.name

x = NamedGreeter('Guido')
print x.name
```

```
Guido
```

```
x.name = 'Tom'
x.greet()
```

```
hi, my name is Tom
```

Доступ к атрибутам

```
class Empty:
    pass


>>> x = Empty()
>>> x
<__main__.Empty instance at ...>
>>> x.smth = 1
>>> x.smth
1
```

Изменение атрибутов извне

Преимущества

- Легче отлаживать и проверять код.
  Больше возможностей у IDE.
- Легче писать группы классов, связанных друг с другом.
- Изменение атрибутов можно “перехватывать”

Изменение атрибутов извне

Преимущества

- Легче отлаживать и проверять код.
  Больше возможностей у IDE.
- Легче писать группы классов, связанных друг с другом.
- Изменение атрибутов можно “перехватывать”

Отсутствие “защиты от дурака”

- Всегда можно все сломать, если есть желание
- Неявная типизация $\leftrightarrow$ документация

“Защищенные” атрибуты

Атрибут только для использования внутри класса.

```
class SecureGreeter:
    def __init__(self, name):
        self._name = name
    def greet(self):
        print 'hi, my name is', self._name
```

“Защищенные” атрибуты

Атрибут только для использования внутри класса.

```
class SecureGreeter:
    def __init__(self, name):
        self._name = name
    def greet(self):
        print 'hi, my name is', self._name
```

- Атрибут все равно можно использовать
- Название предупреждает, что это запрещено делать

“Защищенные” атрибуты

Преимущества

- Легче отлаживать и проверять код.
  Больше возможностей у IDE.
- Легче писать группы классов, связанных друг с другом.
- Изменение атрибутов можно “перехватывать”

“Защищенные” атрибуты

Преимущества

- Легче отлаживать и проверять код.
  Больше возможностей у IDE.
- Легче писать группы классов, связанных друг с другом.
- Изменение атрибутов можно “перехватывать”

Отсутствие “защиты от дурака”

- Всегда можно все сломать, если есть желание
- Многие IDE предупреждают в случае использования
- Неявная типизация $\leftrightarrow$ документация

Переопределение операторов

Имена вида `__*__` переопределяют операторы.

Например, `__init__` переопределяет создание объекта.

Переопределение арифметики

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector2D(self.x + other.x,
                        self.y + other.y)
    def __mul__(self, scalar):
        return Vector2D(self.x * scalar,
                        self.y * scalar)

x = Vector2D(1, 2)
y = x * 2
```

Другие арифметические операторы

- `__sub__` — вычитание
- `__div__` — деление

- `__eq__` — сравнение на равенство
- `__ne__` — сравнение на неравенство

Оператор “квадратные скобки”

```
class Vector2D:
    ...
    def __getitem__(self, index):
        print "Get", index
    def __setitem__(self, index, value):
        print "Set", index, value
```

Оператор “квадратные скобки”

```
class Vector2D:
    ...
    def __getitem__(self, index):
        print "Get", index
    def __setitem__(self, index, value):
        print "Set", index, value
```

```
>>> v = Vector2D(1, 2)
>>> v[0]
Get 0
>>> v[0] = 1
Set 0 1
```

Оператор “квадратные скобки”

```
class Vector2D:
    ...
    def __getitem__(self, index):
        print "Get", index
    def __setitem__(self, index, value):
        print "Set", index, value
```

```
>>> v = Vector2D(1, 2)
>>> v[0]
Get 0
>>> v[0] = 1
Set 0 1

>>> v['abc'] = 1
Set abc 1
```

Оператор “квадратные скобки”

```
class Vector2D:
    ...
    def __getitem__(self, index):
        if index == 0:
            return self.x
        elif index == 1:
            return self.y
    def __setitem__(self, index, value):
        if index == 0:
            self.x = value
        elif index == 1:
            self.y = value
```

Другие операторы

- `__len__` — длина объекта.
  Функция `len()` пользуется этим оператором.
- `__str__` — строковое представление объекта.
  Функция `str()` пользуется этим оператором.
- `__call__` — оператор “круглые скобки”.
  Может принимать любые аргументы.

Использование ООП в Python

Классы на разных языках

Утка на C++	Утка на Python
Клюв	Крякает
Перья	Летает
Лапы	Плавает

Неявная типизация

(также “Утиная типизация”)

Объект определяется своим набором методов и свойств.

(Внутренняя структура не имеет значения)

Пример неявной типизации

```
def count_and_call(function, n):
    for i in range(n):
        function(i)
```

Пример неявной типизации

```
def count_and_call(function, n):
    for i in range(n):
        function(i)

def printer(x):
    print x,
count_and_call(printer, 5)
```

```
0 1 2 3 4
```

Пример неявной типизации

```
def count_and_call(function, n):
    for i in range(n):
        function(i)

def printer(x):
    print x,
count_and_call(printer, 5)
```

```
0 1 2 3 4
```

```
x = []
count_and_call(x.append, 5)
print x
```

```
[0, 1, 2, 3, 4]
```

Еще пример неявной типизации

```
def add(a, b):
    return a + b

>>> add(1, 2)
3
>>> add('a', 'b')
'ab'
>>> add([1, 2], [3])
[1, 2, 3]
```

Использование ООП

Когда нужно использовать классы, а когда — функции?

Функции

- Действия (глаголы)
- Абстрагирование от внутреннего устройства действий

Классы

- Объекты (существительные)
- Объединение данных и операций над ними “в одном месте”
- Абстрагирование от внутреннего устройства данных и операций над ними

Разные способы проектирования

```
class Rational:
    def __init__(self, num, denom=1):
        self.num = num
        self.denom = denom

    def __mul__(self, other):
        return Rational(
            self.num * other.num,
            self.denom * other.denom)
    ...

r1 = Rational(1, 5)
r2 = Rational(2, 3)
r1 * r2
```

Разные способы проектирования

```
def mult_rational(first, second):
    return [first[0] * second[0],
            first[1] * second[1]]

r1 = [1, 5]
r2 = [2, 3]
mult_rational(r1, r2)
```

Разные способы проектирования

```
def gamma(x):
    ...

def beta(x, y):
    ...

def log(x, base=2):
    ...

gamma(3.5)
```

Разные способы проектирования

```
class MathFunctions:
    def gamma(self, x):
        ...

    def beta(self, x, y):
        ...

    def log(self, x, base=2):
        ...

math = MathFunctions()
math.gamma(3.5)
```

Использование ООП в Python

Особенности для Python

Использование ООП в Python

Особенности для Python

- Очень мощные типы `list` и `dict`

Использование ООП в Python

Особенности для Python

- Очень мощные типы `list` и `dict`
- Просто = хорошо

Использование ООП в Python

Особенности для Python

- Очень мощные типы `list` и `dict`
- Просто = хорошо
- Утиная типизация

Документирование кода

Докстринги

```
>>> int.__doc__
"int(x[, base]) -> integer\n\nConvert a
string or number to an integer, ..."

import sys
>>> sys.__doc__
"This module provides access to some
objects used or maintained by ..."
```

Создание докстрингов

Строковый литерал в самом начале объявления (модуля, функции, класса).

```
>>> def f():
...     "Some doc."
...
>>> f.__doc__
'Some doc.'
```

Функции

```
def f(x, y, z):
    """Do stuff.

    This function does this and that.

    """
```

Классы

```
class C(object):

    """Short class description.

    This class works with this stuff and
    that stuff.

    """

    ...
```

Модули

```
"""Short module description.

Some things can be done with this module.

"""

import sys

...
```

Однострочная документация

```
def sum(a, b):
    """Return sum of two values."""
    ...
```

Немного наследования

Отношение “Являться частным случаем”

A является частным случаем B

A наследуется от B

Наследование в Python

```
class Animal:
    pass

class Dog(Animal):
    pass

class Cat(Animal):
    pass
```

Отношение “Являться экземпляром”

Кот Боб является экземпляром кота.
Кот Боб является экземпляром животного.
Кот Боб не является экземпляром собаки.

Отношение “Являться экземпляром”

Кот Боб является экземпляром кота.
Кот Боб является экземпляром животного.
Кот Боб не является экземпляром собаки.

```
>>> isinstance(bob, Cat)
True
>>> isinstance(bob, Animal)
True
>>> isinstance(bob, Dog)
False
```

Универсальный класс

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

>>> isinstance(1, object)
True
>>> isinstance(int, object)
True
>>> isinstance(object, object)
True
```

Наследование от универсального класса

```
class MyClass(object):
    ...
```

Наследование от универсального класса

```
class MyClass(object):
    ...
```

Рекомендуется так делать

- Единообразие
- Больше возможностей