Классы в Python'е

Владимир Борисенко

Мехмат МГУ

vladibor2016@yandex.ru
vladimir_borisen@mail.ru

Классы в Python'е: отличие от C++

Для программистов, привыкшим к объектно ориентированным языкам со *статической типизацией*, таким как C++ или Java, классы в Python'е при первом знакомстве кажутся очень непривычными. Главное отличие:

- в языке C++ *описание класса* — это как бы чертеж объекта, включающий описание его составляющих элементов (данные-члены класса) и методов для работы с ними. Все создаваемые потом *объекты класса* имеют одинаковое устройство, соответствующее его описанию;
- в Python'е *описание класса* вообще не содержит описаний данных-членов класса. *Данные-члены класса* создаются при вызове его методов, причем разные *объекты* класса могут содержать разные данные-члены, т.е. иметь различное устройство.

Это различие отражает главную особенность Python'а как *языка с динамической типизацией* — типы объектов и выражений определяются в общем случае только в процессе работы программы.

Задание методов класса, аргумент *self*

Классы в Python'е задаются с помощью ключевого слова *class*. Можно указать, из какого класса наследуется описываемый класс:

```
class A(B):
    . . .
```

Задан класс *A*, который наследует все методы базового класса *B*. Если базовый класс не указан (нет части в скобках), то по умолчанию все классы наследуются из базового класса *object*.
Важно! У всех методов класса (обычных, не статических) первым аргументом является ссылка на объект, к которому метод применяется. По соглашению между всеми программистами этот аргумент называется *self*, хотя это правило не формальное. Объект класса создается с помощью конструктора, который в Python'е называется $\_\_init\_\_$:

```
class R2Vector:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

Здесь мы описываем класс *R2Vector*, представляющий вектор на плоскости $\mathbb{R}^2$. При выполнении конструктора внутри объекта создаются две переменные x, y, которым присваиваются значения, переданные как параметры конструктору.
Переменная *self* в Python'е выполняет ту же роль, что и *this* в C++, но, в отличие от C++, должна указываться явно как первый параметр любого метода. Например, определим евклидову норму вектора (его длину) как метод *norm*:

```
class R2Vector:
    . . .
    def norm(self):
        return (self.x*self.x + self.y*self.y) ** 0.5
```

Вызов обычных методов для объектов класса

Методы в Python'е вызываются точно так же, как и в других объектно ориентированных языках: указывается имя объекта, затем точка и имя метода, после чего в скобках указывается список параметров, который может быть и пустым — но скобки обязательны! Параметр *self* при вызове не указывается. Например, в интерпретаторе Python'а можно набрать следующие команды:

```
>>> v = R2Vector(1, 1)
>>> n = v.norm()
>>> print(n)
1.4142135623730951
```

Здесь в выражении `v.norm()` мы применяем метод *norm* класса *R2Vector* к объекту *v*, в качестве аргумента *self* передается ссылка на *v*.
Кстати, и такой вызов допустим (но так никто не делает):

```
>>> R2Vector.norm(v)
1.4142135623730951
```

Полиморфизм

Вернемся к определению конструктора класса *R2Vector*.

```
def __init__(self, x = 0, y = 0):
    self.x = x; self.y = y
```

Недостаток такого решения в том, что мы определили только конструктор по двум координатам вектора, которые заданы как два отдельных аргумента. Но хотелось бы иметь также и конструктор по одному аргументу, который является либо списком, либо кортежем (tuple), содержащим пару чисел. В C++ можно задать несколько конструкторов с разными наборами аргументов, это называется *полиморфизмом*. В Python'е полиморфизма нет, но он и не нужен, поскольку при выполнении можно определить тип аргумента с помощью встроенной функции *type*:

```
def __init__(self, x = 0, y = 0):
    if type(x) == tuple or type(x) == list:
        self.x = x[0]; self.y = x[1]
    else:
        self.x = x; self.y = y
```

Теперь можно создавать векторы разными способами:

```
>>> u = R2Vector(1, 1)
>>> v = R2Vector((1, 2))
>>> w = R2Vector([3, 4])
```

Подобный “полиморфизм” возможен не только в случае конструктора, но и для любых других методов класса. Например, когда мы умножаем два вектора на плоскости, выполняется операция скалярного произведения векторов, результатом которой является число; но также можно умножить вектор на число, в результате получается вектор. Операция умножения * реализуется с помощью метода $\_\_mul\_\_$, в котором мы рассматриваем два разных случая по типу второго операнда:

```
    def __mul__(self, v):
        if type(v) == R2Vector:
            # Scalar product of 2 vectors
            return self.x * v.x + self.y * v.y
        else:
            # Multiply a vector by a number
            return R2Vector(self.x * v, self.y * v)
```

Реализация арифметических операций для класса

Четыре арифметические операции $+$, $-$, $*$, $/$ реализуются с помощью методов `__add__`, `__sub__`, `__mul__`, `__div__`. Например,

```
def __add__(self, v):
    return R2Vector(self.x + v.x, self.y + v.y)
```

В результате выполнения операции сложения двух векторов получается третий вектор, равный их сумме:

```
>>> u = R2Vector(1, 2); v = R2Vector(3, 4)
>>> w = u + v
>>> print(u, v, w)
(1, 2) (3, 4) (4, 6)
```

Но, помимо операции сложения двух векторов, есть еще операция увеличения первого вектора на второй +=, которая реализуется с помощью метода `__iadd__` (буква *i* от слов in-place). В результате не создается новый объект, равный сумме двух векторов, а изменяется первый аргумент сложения:

```
def __iadd__(self, v):
    self.x += v.x; self.y += v.y
```

Пример применения метода *__iadd__*:

```
>>> u = R2Vector(1, 2); v = R2Vector(3, 4)
>>> u += v
>>> print(u)
(4, 6)
```

Таким образом, для определения четырех арифметических операций для класса надо реализовать 8 методов, по 2 для каждой операции: *__add__* и *__iadd__* для операции сложения (addition), *__sub__* и *__isub__* для вычитания (subtraction) и т.д.

Другие стандартные методы

Помимо арифметических операций, обычно нужно реализовать и другие стандартные методы. Рассмотрим их на примере класса *R2Vector*.

- **Преобразование к текстовой форме.** Метод называется *__str__*, он применяется к объекту v как функция $str(v)$.

```
def __str__(self):
    return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Кроме функции $str(v)$, надо реализовать еще функцию $repr(v)$ (от representation — представление):

```
def __repr__(self):
    return "R2Vector("+repr(self.x)+", "+repr(self.y)+")"
```

Пример применения:

```
>>> v = R2Vector(3, 4)
>>> str(v)
'(3, 4)'
>>> repr(v)
'R2Vector(3, 4)'
```

- **Работа с объектом класса как с массивом.** К координатам x, y вектора v можно обращаться как $v[0]$, $v[1]$. Для этого надо реализовать два метода $\_\_getitem\_\_$ и $\_\_setitem\_\_$, выполняющие чтение и запись элемента массива с заданным индексом:

```
def __getitem__(self, idx):
    if i == 0:
        return self.x
    else:
        return self.y
def __setitem__(self, idx, value):
    if i == 0:
        self.x = value
    else:
        self.y = value
```

- **Сравнение объектов класса.** Для возможности сравнения объектов надо реализовать 6 методов $<$, $\leq$, $>$, $\geq$, $=$, $\neq$. В Python'е они называются *__lt__*, *__le__*, *__gt__*, *__ge__*, *__eq__*, *__ne__* (от слов greater than, greater or equal, less than, less or equal, equal, not equal).

```
def __lt__(self, v):
    return self.x < v.x or (
        self.x == v.x and self.y < v.y)
def __le__(self, v):
    return self.x < v.x or (
        self.x == v.x and self.y <= v.y)
def __gt__(self, v):
    return not self <= v
def __ge__(self, v):
    return not self < v
def __eq__(self, v):
    return self.x == v.x and self.y == v.y
def __ne__(self, v):
    return not self == v
```

Отметим, что, если для класса определены операторы == и !=, то объекты сравниваются по значению, а не по ссылке, т.е. это могут быть разные объекты, содержащие одно и то же значение. Если нужно сравнить ссылки, то вместо оператора == следует использовать ключевое слово *is*:

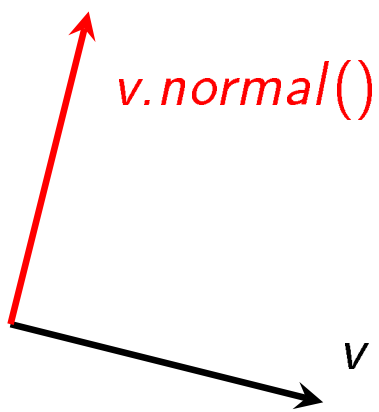
```
>>> u = R2Vector(3, 5)
>>> v = R2Vector(3, 5)
>>> id(u)
139785146657144
>>> id(v)
139785146657200
>>> u == v
True
>>> u is v
False
```

Содержательные методы класса R2Vector

Реализуем ряд содержательных методов класса *R2Vector*. Они нам понадобятся для решения самых разных геометрических задач на плоскости.

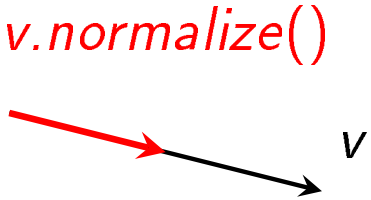
- **Нормаль к вектору.** Метод *normal* возвращает вектор, перпендикулярный данному, имеющий такую же длину и получающийся из данного вектора поворотом на 90° против часовой стрелки:

```
def normal(self):
    return R2Vector(-self.y, self.x)
```



- **Нормализация вектора.** Метод *normalize* делает длину вектора равной единице, сохраняя его направление:

```
def normalize(self):
    n = self.norm()
    if n > 0:
        self.x /= n
        self.y /= n
```



- В отличие от метода *normalize*, метод *v.normalized*() оставляет вектор *v* нетронутым, возвращая другой нормализованный вектор единичной длины, по направлению совпадающий с *v*:

```
def normalized(self):
    n = self.norm()
    if n > 0.:
        return R2Vector(self.x/n, self.y/n)
    else
        return R2Vector(self.x, self.y)
```

- **Скалярное произведение векторов.** Оператор умножения * реализует как скалярное произведение двух векторов, так и умножение вектора на число:

```
def __mul__(self, v):
    if type(v) == R2Vector:
        # Scalar product of 2 vectors
        return self.x * v.x + self.y * v.y
    else:
        # Multiply a vector by a number
        return R2Vector(self.x*v, self.y*v)
```

Метод возвращает либо число (скалярное произведение), либо новый вектор:

```
>>> u = R2Vector(3, 1)
>>> v = R2Vector(1, -1)
>>> u*v
2
>>> u*0.5
R2Vector(1.5, 0.5)
```

Класс R2Point

Класс *R2Vector* определен в модуле *R2Graph*, заданным в файле "R2Graph.py". Помимо класса *R2Vector*, этот модуль содержит также определение класса *R2Point*, который представляет точку на плоскости $\mathbb{R}^2$. В целом описание этого класса похоже на класс *R2Vector*. Отметим важные отличия:

- операция сложения точек не определена — геометрически сложение координат точек бессмысленно, т.к. результат зависит от выбора начала координат;
- однако *разность* двух точек $q - p$ имеет смысл — это *вектор*, направленный от точки p к точке q. Также к точке можно прибавить или отнять вектор:

```
class R2Point:
    def __sub__(self, p):
        if type(p) == R2Point:
            return R2Vector(self.x - p.x, self.y - p.y)
        else:
            assert type(p) == R2Vector
            return R2Point(self.x - p.x, self.y - p.y)
```

```
def __add__(self, v):
    assert type(v) == R2Vector
    return R2Point(self.x + v.x, self.y + v.y)
```