

# Неделя 3. Часть 1. ООП в Python

## Цели лекции

- ознакомиться с принципами ООП, особенностями применения в Python, понять преимущества ООП;
- научиться создавать пользовательские типы данных – классы, а также использовать их на практике;

## План лекции

План лекции .....	1
<b>ООП in Python</b> .....	2
ООП .....	2
СОЗДАНИЕ КЛАССОВ .....	2
ПРОСТРАНСТВО ИМЕН .....	3
При создании экземпляра (инстанса) класса: .....	3
Изменяемость (mutability) и ООП .....	4
ИНСТАНЦИРОВАНИЕ. СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА .....	5
МЕТОДЫ ЭКЗЕМПЛЯРОВ КЛАССОВ .....	6
ООП ПОДХОД К РЕШЕНИЮ КВАДРАТНОГО УРАВНЕНИЯ .....	7
Класс QuadraticEquation: .....	7
КТО ПОЛЬЗУЕТСЯ МОИМ КЛАССОМ .....	7
ИНКАПСУЛЯЦИЯ .....	8
НАСЛЕДОВАНИЕ .....	10
ПОЛИМОРФИЗМ .....	10
ОБРАЩЕНИЕ К ФУНКЦИЯМ РОДИТЕЛЯ, МЕТОД SUPER .....	11
<b>Абстрактные классы. Статические методы. Методы класса.</b> .....	12
"ОБЫЧНЫЕ" МЕТОДЫ .....	12
ЛИСТИНГ № 1. Класс: .....	12
ЛИСТИНГ № 3 .....	12
СТАТИЧЕСКИЕ МЕТОДЫ .....	13
ЛИСТИНГ № 5 .....	13
МЕТОДЫ КЛАССА .....	13
ЛИСТИНГ № 6 .....	13
<b>АБСТРАКТНЫЕ КЛАССЫ</b> .....	14
Листинг № 7 .....	14
Листинг № 8 .....	14
Листинг № 9 .....	14
<b>ПОЛЕЗНЫЕ ССЫЛКИ</b> .....	15

## OOP in Python

### ООП

ООП — аббревиатура объектно-ориентированного программирования.

Любая программа - это данные плюс алгоритмы, обрабатывающие эти данные

ООП способ организации структуры программы, чтоб данные и функционал хранились в рамках единой сущности. Эта сущность есть объект. Объект в ООП описывается атрибутами-свойствами и функциями-методами, которые выполняют какие-либо действия над данными атрибутами. Какими именно атрибутами и методами может обладать объект - описывается в классе. Т.е. сам класс - это новый тип данных, а объект - это переменная данного типа.

Определение по Гради Бучу:

*Объектно-ориентированное программирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.*

Например, тип данных строка `str` - класс. Переменная типа `str` это объект. Также переменные определенного типа называют инстансом (другими словами экземпляром, объектом) данного типа или инстансом данного класса.

```
str # тип данных, класс
```

```
s = str()
```

```
s # переменная данного типа или инстанс(экземпляр) класса
```

Все в мире можно описать терминами объекта

Существует много терминологии вокруг ООП: абстракция, агрегация, инстанцирование, инкапсуляция, наследование, полиморфизм.

**Более подробно и академично, но очень доступно, можно почитать здесь**

[http://kpolyakov.spb.ru/download/ch11-7\\_python.pdf](http://kpolyakov.spb.ru/download/ch11-7_python.pdf)

( главы об ООП учебника «Информатика. Углублённый уровень» для 10-11 классов К.Ю. Полякова и Е.А. Еремина.)

## СОЗДАНИЕ КЛАССОВ

**Создавая новый класс, создаем новый тип данных.**

Для создания классов предусмотрена инструкция `class`. Это составная инструкция, которая состоит из строки заголовка и тела. Заголовок состоит из ключевого слова `class`, имени класса (имя принято писать CamelCase) и названий суперклассов в скобках. Тело класса состоит из блока различных инструкций. Тело должно иметь отступ (как и любые вложенные конструкции в языке Python, четыре пробела).

Синтаксис объявления класса

```
class ClassName:
    ''' docstring '''
    <body of class>
```

```
s = str() # переменная данного типа или инстанс (экземпляр) класса
```

```
isinstance(s, str)
type(str), type(ClassName)
instance = ClassName()
isinstance(instance, ClassName)
```

Схематично класс можно представить следующим образом:

```
class ClassName(object):
    ''' docstring '''
    attribute_name = 'value'

    def method_name(self, ...):
        self.a = ...
```

Пример (создание класса – сущности с описание его атрибутов и методов, область видимости переменных, обращение к ним):

```
class SomeClass:
    """Some help text"""
    a = 10
    b = 20
    def some_func(self):
        print("Hello")
```

```
In [1]: class SomeClass:
...:     """Some help text"""
...:     a = 10
...:     b = 20
...:     def some_func(self):
...:         print("Hello")
...:

In [2]: str
Out[2]: str

In [3]: print(str)
<class 'str'>

In [4]: print(int)
<class 'int'>

In [5]: print(SomeClass)
<class '__main__.SomeClass'>

In [5]: print(SomeClass)
<class '__main__.SomeClass'>

In [6]: SomeClass.a
Out[6]: 10

In [7]: SomeClass.b
Out[7]: 20

In [8]: SomeClass.b = 30

In [9]: SomeClass.b
Out[9]: 30
```

## ПРОСТРАНСТВО ИМЕН

### При создании экземпляра (инстанса) класса:

- создает пространство имен, к которому можно обратиться через точку как из самого объекта класса, так и через инстанс;
- у инстанса свое независимое пространство имен, но он имеет доступ к пространству класса.

Описание класса создает локальное пространство имен класса. Инстанс класса – это конкретный экземпляр класса. Переменная данного типа – это объект данного типа. Инстанс класса также создает собственное пространство имен. Класс у нас уникален, а конкретных сущностей (инстансов, экземпляров класса) может быть создано сколько угодно много. Процедура создания экземпляра класса – инстансирования выглядит следующим образом: `rectangle = Rectangle()`

```
class Rectangle(object):
    a, b = 10, 20
    c = [1]

print(Rectangle.a, Rectangle.b)
rectangle = Rectangle()
rectangle.a = 30
print(rectangle.a, Rectangle.a)
print(Rectangle.c)
```

Инстанс, экземпляр класса получает все атрибуты и и методы как и базовый класс. При обращении к атрибуту класса от инстанса, происходит поиск переменной вначале в объекте, а потом в классе. При обращении к атрибуту по имени из объекта, если переменная с таким именем в пространстве имен объекта не найдена, то поиск продолжается в пространстве имен класса. В момент присвоения значения атрибуту, при обращении к нему из объекта, происходит создание отдельной локальной переменной в пространстве имен объекта. Чем-то напоминает ситуацию с передачей в функцию изменяемых объектов в качестве параметра по умолчанию.

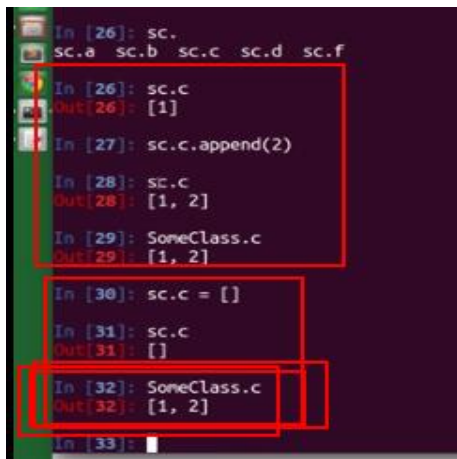
### Изменяемость (mutability) и ООП

- при присвоении значения изменяемому объекту мы создаем локальную переменную,
- с изменяемым значением мы правим исходный объект.

```
class Rectangle(object):  
    a, b = 10, 20  
    c = [1]  
  
print(Rectangle.c)  
rectangle.c.append(2)  
print(rectangle.c, Rectangle.c)  
print(Rectangle.c)
```

У **инстанса** (экземпляра класса, объекта) есть свое независимое пространство имен, но он имеет доступ к пространству класса. К пространству имен (атрибуту) можно обратиться через точку (дот-нотация) как из самого **объекта класса**, так и через инстанс.

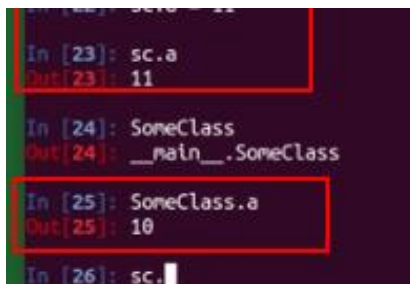
Если меняем список, присущий инстансу, то меняется и список типа данных (класса).



```
In [26]: sc.  
sc.a sc.b sc.c sc.d sc.f  
  
In [26]: sc.c  
Out[26]: [1]  
  
In [27]: sc.c.append(2)  
  
In [28]: sc.c  
Out[28]: [1, 2]  
  
In [29]: SomeClass.c  
Out[29]: [1, 2]  
  
In [30]: sc.c = []  
  
In [31]: sc.c  
Out[31]: []  
  
In [32]: SomeClass.c  
Out[32]: [1, 2]  
  
In [33]:
```

Если же переписываем список, присущий инстансу, то меняется только список инстанса; список типа остается неизменным.

Инстанс и тип:



```
In [23]: sc.a  
Out[23]: 11  
  
In [24]: SomeClass  
Out[24]: __main__.SomeClass  
  
In [25]: SomeClass.a  
Out[25]: 10  
  
In [26]: sc.
```

Присвоение атрибута классу не влияет на значение атрибута инстанса класса:

```

AttributeError                                Traceback (most recent call last)
<ipython-input-18-e902e35abab0> in <module>()
----> 1 SomeClass.d

AttributeError: type object 'SomeClass' has no attribute 'd'

In [19]: SomeClass.f = 50

In [20]: sc.f
Out[20]: 50

```

**Итого:** при присвоении значения изменяемому объекту мы создаем локальную переменную; с изменяемым значением мы правим исходный объект класса.

Тип инстанса:

```

In [8]: SomeClass
Out[8]: __main__.SomeClass

In [9]: sc
Out[9]: <__main__.SomeClass at 0xb6b4c04c>

In [10]: type(SomeClass)
Out[10]: type

In [11]: type(sc)
Out[11]: __main__.SomeClass

```

Для определения принадлежности инстанса к определенному классу можно использовать функцию **isinstance**.

## ИНСТАНЦИРОВАНИЕ. СОЗДАНИЕ ЭКЗЕМПЛЯРА КЛАССА

Классы имеют специальный метод, который автоматически вызывается при создании объекта. Т.е. вызывать данный метод не нужно, т.к. он сам запускается при вызове класса. Такой метод называется **конструктором класса** и в языке программирования Python носит имя **\_\_init\_\_**. (В начале и конце по два знака подчеркивания.)

**Основные моменты:**

- При инстанцировании каждый раз вызывается специальный метод `_init_`
- Все функции первым параметром получают `self`, инстанс для которого эта функция вызывается
- Если при инстанцировании класс вызвать с параметрами, то именно метод `_init_` получит эти параметры

Демонстрируем работу метода `__init__` (вариант с print)

```

In [60]: class Rectangle:
...:     a = 10
...:     b = 20
...:     def __init__(self):
...:         print(self)
...:         print(type(self))
...:
In [61]: r = Rectangle()
<__main__.Rectangle object at 0x1043d3588>
<class '__main__.Rectangle'>

In [62]: print(r)
<__main__.Rectangle object at 0x1043d3588>

In [63]: print(type(r))
<class '__main__.Rectangle'>

```

```

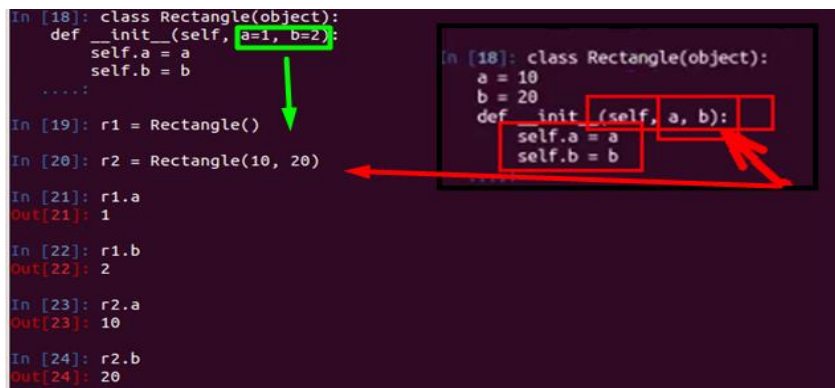
In [54]: class Rectangle:
...:     a = 10
...:     b = 20
...:     def __init__(self):
...:         print("In init!")
...:
In [55]: r = Rectangle()
In init!

```

Более традиционный вариант использования метода `__init__`: атрибуты класса, задаваемые при создании экземпляров класса правильно выносить в `__init__`. Первым параметром, как и у любого другого метода, у `__init__` является `self`, на место которого подставляется объект в момент его создания. Второй и последующие


параметры заменяются аргументами, переданными в конструктор при вызове класса. На уровне пространства имен класса мы будем определять константы, которые не будут меняться.

Так же как обычная функция может иметь обязательные и необязательные параметры:



```
In [18]: class Rectangle(object):
...:     def __init__(self, a=1, b=2):
...:         self.a = a
...:         self.b = b
...:
In [19]: r1 = Rectangle()
In [20]: r2 = Rectangle(10, 20)
In [21]: r1.a
Out[21]: 1
In [22]: r1.b
Out[22]: 2
In [23]: r2.a
Out[23]: 10
In [24]: r2.b
Out[24]: 20
```

Пример бизнес-логики в `__init__` - передача необязательного параметра дает возможность задать квадрат вместо прямоугольника:

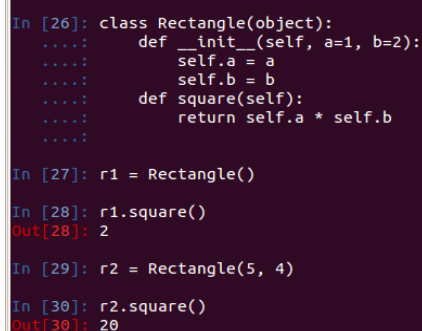


```
In [85]: class Rectangle:
...:     a = 10
...:     b = 20
...:     def __init__(self, a, b=None):
...:         self.a = a
...:         if b is None:
...:             self.b = a
...:         else:
...:             self.b = b
...:
In [86]: r = Rectangle(60)
In [87]: r.a
Out[87]: 60
In [88]: r.b
Out[88]: 60
```

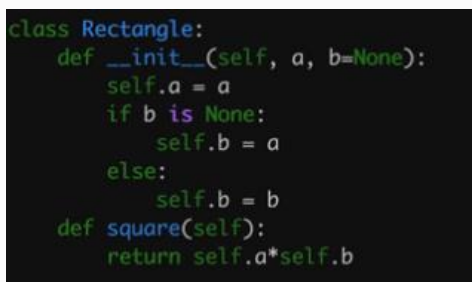
## МЕТОДЫ ЭКЗЕМПЛЯРОВ КЛАССОВ

- В теле класса можно описывать функции - методы класса
- Все функции первым параметром получают `self`, через данную переменную в функции можно обращаться к атрибутам инстанса и функциям класса
- методы класса ведут себя как обычные функции за исключением первого параметра, т.е. можно передавать другие параметры, возвращать значения и т.д.

Методы - это обычные функции, которые создаются инструкциями `def` в теле инструкции `class`. С точки зрения программирования методы работают точно также, как и обычные функции, с одним важным исключением: в первом аргументе метода всегда **передается подразумеваемый объект экземпляра**. Этот аргумент обычно называется `self`, в соответствии с общепринятыми соглашениями (с технической точки зрения само имя не играет никакой роли, значение имеет позиция аргумента).



```
In [26]: class Rectangle(object):
...:     def __init__(self, a=1, b=2):
...:         self.a = a
...:         self.b = b
...:     def square(self):
...:         return self.a * self.b
...:
In [27]: r1 = Rectangle()
In [28]: r1.square()
Out[28]: 2
In [29]: r2 = Rectangle(5, 4)
In [30]: r2.square()
Out[30]: 20
```



```
class Rectangle:
    def __init__(self, a, b=None):
        self.a = a
        if b is None:
            self.b = a
        else:
            self.b = b
    def square(self):
        return self.a*self.b
```

Пример, когда «self уже не нужен»; обращаемся внутри функции класса к другой функции класса:

```
In [5]: class Rectangle(object):
def __init__(self, a, b=None):
    self.a = a
    if b is None: self.b = a
    else: self.b = b
def square(self):
    return self.a * self.b
...:     def half_square(self):
...:         return self.square()
```

## ООП ПОДХОД К РЕШЕНИЮ КВАДРАТНОГО УРАВНЕНИЯ

- Описываем атрибутами класса переменные уравнения
- Упаковываем функции внутрь класса

Класс QuadraticEquation:

```
class QuadraticEquation:
    """ Quadratic Equation solution """
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def calc_descr(self):
        self.d = self.b ** 2 - 4 * self.a * self.c
    def get_desc(self):
        return self.d
    def calc_root(self, order = 1):
        if order == 1:
            x = (-self.b + self.d ** (1 / 2)) / 2 * self.a
        else:
            x = (-self.b - self.d ** (1 / 2)) / 2 * self.a
        return x

qe = QuadraticEquation(1, 2, 1) # создание экземпляра класса
print(qe.calc_descr()) # обращение к методу класса
```

## КТО ПОЛЬЗУЕТСЯ МОИМ КЛАССОМ

Пользователь класса:

1. **Вы**, если продолжаете использовать в программе разработанный вами класс
2. Один разработчик описывает класс, опираясь, на некие спецификации, а **другой разработчик** работает над бизнес логикой взаимодействия с пользователем. В этом случаи *пользователь класса* тот, кто разрабатывает бизнес логику
3. В процессе работы с библиотекой (например, math) вы могли использовать скрытые классы. В этом случае **вы** являетесь *пользователем класса*

**Пользователь класса** – тот программист, который будет пользоваться ваши классом через интерфейс, который вы ему предоставили.



- реализовано на уровне соглашения: имена функций и атрибутов начинаются с "\_" приватные и обращаться можно к ним только из тела класса

- сильно приватные имена функций и атрибутов начинаются "\_\_" (двойного подчеркивания), доступ к ним тоже можно получить, но лучше этого не делать

Инкапсуляция является одним из ключевых понятий ООП.

Скрытие информации о внутреннем устройстве объекта выполняется в Python на уровне соглашения между программистами о том, какие атрибуты относятся к общедоступному интерфейсу класса, а какие — к его внутренней реализации. Одиночное подчеркивание в начале имени атрибута говорит о том, что метод не предназначен для использования вне методов класса (или вне функций и классов модуля), **однако, атрибут все-таки доступен по этому имени. Два подчеркивания в начале имени дают несколько большую защиту: атрибут будет доступным под именем вида** `_ИмяКласса_ИмяАтрибута`

```
In [39]: class Rectangle(object):
        def _private(self):
            return 'private'
        def __really_private(self):
            return '__really_private'
        ....:

In [40]: r = Rectangle()

In [41]: r._private()
Out[41]: 'private'

In [42]: r._Rectangle__really_private()
Out[42]: '__really_private'
```

В Питоне нет приватных атрибутов класса как таковых (нет модификаторов доступа public/private/protected, свойственных, например, для Java, C#). «Задачи установления приватности» выполняет символ нижнего подчеркивания перед наименованием переменной. На скриншоте ниже - `_d`; можно прописать **пользовательскую** функцию-геттер (`get_discr` в листинге ниже), которая будет вызывать «приватное» поле:

```
class QuadraticEquation(object):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def calc_discr(self):
        self._d = self.b ** 2 - 4 * self.a * self.c

    def get_discr(self):
        return self._d

    def get_eq_root(self, order=1):
        if order==1:
            x = (-self.b + self._d ** (1/2.0)) / 2*self.a
        else:
            x = (-self.b - self._d ** (1/2.0)) / 2*self.a
        return x
```

При этом важно понимать, что **напрямую, непосредственно мы не должны менять d, однако можем**; символ «\_» применяется на уровне соглашения, как «сигнал важности атрибута» другому разработчику). Два подчеркивания частично решают проблему:



```

def __init__(self):
    self.__b = 10
...:     def set_b(self):
...:         self.b = 1
...:
In [6]: class A(object):
def __init__(self):
    self.__b = 10
def set_b(self, b):
    self.__b = b
...:

In [7]: a = A()

In [8]: a.__b
Out[8]: 10

In [9]: a.set_b(20)

In [10]: a.__b
Out[10]: 20

```

Служебные функции:

```

Out[11]:
['_A__b',
 '__class__',
 '__delattr__',
 '__dict__',
 '__doc__',
 '__format__',
 '__getattribute__',
 '__hash__',
 '__init__',
 '__module__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'set_b']

```

Пример переопределения функции len (обратите внимание: возможны два варианта вызова функции):

```

In [12]: class A(object):
def __init__(self):
    self.__b = 10
def set_b(self, b):
    self.__b = b
...:     def __len__(self):
...:         return self.__b
...:

In [13]: a = A()

In [14]: len('adsfas')
Out[14]: 6

In [15]: len([])
Out[15]: 0

In [16]: len(a)
Out[16]: 10

In [17]: a.__len__()
Out[17]: 10

```

```

In [12]: class A(object):
def __init__(self):
    self.__b = 10
def set_b(self, b):
    self.__b = b
...:     def __len__(self):
...:         return self.__b
...:

In [13]: a = A()

```

```

In [16]: len(a)
Out[16]: 10

In [17]: a.__len__()
Out[17]: 10

```

## НАСЛЕДОВАНИЕ

- указав при определении класса родителя, текущий класс наследует от родителя все атрибуты и функции

**Наследование** - это механизм создания новых классов, призванный настроить или изменить поведение существующего класса. Оригинальный класс называют базовым или суперклассом. Новый класс называют производным классом или подклассом. Когда новый класс создается с использованием механизма наследования, он наследует атрибуты базовых классов. Однако производный класс может переопределить любой из этих атрибутов и добавить новые атрибуты.

Наследование определяется перечислением в инструкции `class` имен базовых классов через запятые. (интерфейс множественного наследования) Порядок поиска атрибутов в базовых классах при множественном наследовании можно увидеть, если вывести содержимое атрибута `__mro__` класса. (см.скриншот!)

```
In [9]: class A(object):pass
In [10]: class B(A): pass
In [11]: class C(A): pass
In [12]: class D(B, C): pass
In [13]: D.__mro__
Out[13]: (_main__.D, _main__.B, _main__.C, _main__.A, object)
In [14]: class A(object):pass
In [15]: class B(A): pass
In [16]: class C(object): pass
In [17]: class D(B, C): pass
In [18]: D.__mro__
Out[18]: (_main__.D, _main__.B, _main__.A, _main__.C, object)
```

Класс `RaceCat` наследует все атрибуты и методы от `Car`:

```
class RaceCar(Car):
    pass
```

## ПОЛИМОРФИЗМ

- в наследнике можно переопределить функцию с тем же именем в итоге будет вызываться разная функция с одинаковым именем для экземпляров разных классов

**Полиморфизм** - разное поведение одного и того же метода в разных классах. Например, мы можем сложить два числа, и можем сложить две строки. При этом получим разный результат, так как числа и строки являются разными классами.

```
>>> 1 + 1
>>> 2
>>> '1' + '1'
>>> '11'
```

**Пример:** Переопределение одноименного метода в классе наследника:

```

class Rectangle(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def square(self):
        return self.a * self.b
    def perimeter(self):
        return self.a*2 + self.b*2

class Square(Rectangle):
    def __init__(self, a):
        self.a = self.b = a
    def perimeter(self):
        return self.a*4

```

## ОБРАЩЕНИЕ К ФУНКЦИЯМ РОДИТЕЛЯ, МЕТОД SUPER

Можем обращаться к своему родителю:

```

class Rectangle(object):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def square(self):
        return self.a * self.b

    def perimeter(self):
        return self.a*2 + self.b*2

class Square(Rectangle):
    def __init__(self, a):
        b = a
        super(Square, self).__init__(a, b)

    def perimeter(self):
        return self.a*4

r = Rectangle(10, 20)
print r.square()

s = Square(20)
print s.square()

```

## Абстрактные классы. Статические методы. Методы класса.

Данная тема - дополнение к лекционному материалу, по ней не будет ни тестовых, ни контрольных заданий, ее цель - продемонстрировать больше возможностей Python.

### "ОБЫЧНЫЕ" МЕТОДЫ

Напомним, что метод — это функция, являющаяся атрибутом класса. Прежде мы работали только с методами, привязанными к **объекту**, т. е. **инстансу** (bound to instance): если в аргументы данной функции передан параметр `self`, то метод можно вызвать только для конкретного инстанса класса, а не для класса:

#### ЛИСТИНГ № 1. Класс:

```
class FantasticVehicle(object):
    def __init__(self, can_fly, can_swim):
        self.can_fly = can_fly
        self.can_swim = can_swim
    def get_features(self):
        return self.can_fly, self.can_swim
```

#### ЛИСТИНГ № 2. Попытка вызова метода инстанса как атрибут класса (unbound method):

```
>>> FantasticVehicle.get_features # вызов метода-объекта — получение информации
<unbound method FantasticVehicle.get_features>
>>> FantasticVehicle.get_features() # собственно, попытка
Traceback (most recent call last):
  File "abstract_class.py", line 8, in <module>
    FantasticVehicle.get_features()
TypeError: unbound method get_features() must be called with FantasticVehicle instance as
first argument (got nothing instead)
```

Метод `get_features`, при его объявлении, получает `self` — значит, это **метод инстанса**, он должен быть привязан к инстансу (**bound method**). Привязать можно так:

#### ЛИСТИНГ № 3

```
>>> v = FantasticVehicle(True, True) # создание экземпляра класса — инстанса
FantasticVehicle.get_features(v)
(True, True)
```

а лучше — так (так мы делали в предыдущих лекциях и будем делать на практиках):

#### ЛИСТИНГ № 4

```
>>> v.get_features()
(True, True)
v.get_features # вызов метода-объекта — получение информации
<bound method FantasticVehicle.get_features of <__main__.FantasticVehicle object at
0x7f41b5ea9390>>
```

**Статические методы** не зависят от инстанса, не зависят от состояния объекта. Выполняются быстрее, чем методы, которые должны быть связаны с инстансом: функции не нужно передавать **self**. Для вызова статического метода не нужно: создавать инстанс, привязываться к инстансу, инстанцировать метод, привязанный к инстансу. При объявлении статических методов в Python используется декоратор **@staticmethod** (декораторы — это тема курса Advanced Python; пока используем декоратор как данность).

## ЛИСТИНГ № 5

```
class FantasticVehicle(object):
    # ...
    @staticmethod
    def define_signals(a):
        return 'My signal is: <{}>'.format(a)

    def signalize(x, y):
        return self.define_signals(self.signal)

>> FantasticVehicle.define_signals # info
<function define_signals at 0x7f60584b8c08>
>> v.define_signals # info
<function define_signals at 0x7f60584b8c08>
>> FantasticVehicle.define_signals('B-e-e-e-p!')
'My signal is: "B-e-e-e-p!'"
>> v.define_signals('B-e-e-e-p!')
'My signal is: "B-e-e-e-p!'"
```

Как бы мы ни вызывали метод класса (как атрибут инстанса или как атрибут класса), мы получаем результат работы статического метода - функции **define\_signals**.

## МЕТОДЫ КЛАССА

**Метод класса** привязывается к классу, а не к объекту (инстансу).

При объявлении статических методов в Python используется декоратор **@classmethod**.

## ЛИСТИНГ № 6

```
class FantasticVehicle(object):
    # ...

    planet_of_origin = 'Jupiter'

    @classmethod
    def get_planet_of_origin(any_class):
        return any_class.planet_of_origin

v = FantasticVehicle(True, True) # создание экземпляра класса - инстанса

>> FantasticVehicle.get_planet_of_origin # info
<bound method type.get_planet_of_origin of <class '__main__.FantasticVehicle'>>
>> v.get_planet_of_origin # info
<bound method type.get_planet_of_origin of <class '__main__.FantasticVehicle'>>
>> FantasticVehicle.planet_of_origin # class attribute
Jupiter
>> FantasticVehicle.get_planet_of_origin()
Jupiter
>> v.get_planet_of_origin()
```

Как бы мы ни вызывали метод класса (как атрибут инстанса или как атрибут класса), мы получаем результат работы метода класса - функции `get_planet_of_origin`., возвращающей значение атрибута класса `planet_of_origin`.

## АБСТРАКТНЫЕ КЛАССЫ

Мы уже работали с перегрузкой метода родителя в классе наследника. Python дает возможность определить базовый класс, который будет **обязывать** разработчика перегрузить его методы в классе-наследнике. При этом, в базовом классе может и не быть реализации методов (**implementation**).

В Python рекомендуется использовать модуль `abc`.

### Листинг № 7

```
import abc

class BaseVehicle(object):
    __metaclass__ = abc.ABCMeta # need to indicate!

    @abc.abstractmethod # need to indicate!
    def have_a_ride(self):
        pass

class Car(BaseVehicle): # inherits from abstract class!
    def __init__(self, sound):
        self.sound = sound

    def have_a_ride(self):
        return 'I\'m {}-ing'.format(self.sound)

>> car = Car('v-z-z-z')
>> car.have_a_ride()
'I'm v-z-z-z-ing'
```

Мы обязаны переопределить метод абстрактного класса, иначе получим ошибку:

### Листинг № 8

```
class Bicycle(BaseVehicle):
    def init (self, sound):
        self.sound = sound

    # we forget to declare have_a_ride

>> bicycle = Bicycle('silent')
TypeError: Can't instantiate abstract class Bicycle with abstract methods have_a_ride
Нельзя создать экземпляр абстрактного класса:
```

### Листинг № 9

```
>> vehicle = BaseVehicle()
TypeError: Can't instantiate abstract class BaseVehicle with abstract methods have_a_ride
```

## ПОЛЕЗНЫЕ ССЫЛКИ

- Больше примеров — в блоге: <https://julien.danjou.info/blog/2013/guide-python-static-class-abstract-methods>

Документация — статические методы и методы класса:

<https://docs.python.org/3/howto/descriptor.html#static-methods-and-class-methods>

- Abstract Base Classes in Python <https://dbader.org/blog/abstract-base-classes-in-python>