



# Основы объектно-ориентированного программирования - 2

## Технологии и языки программирования

Юдинцев В. В.

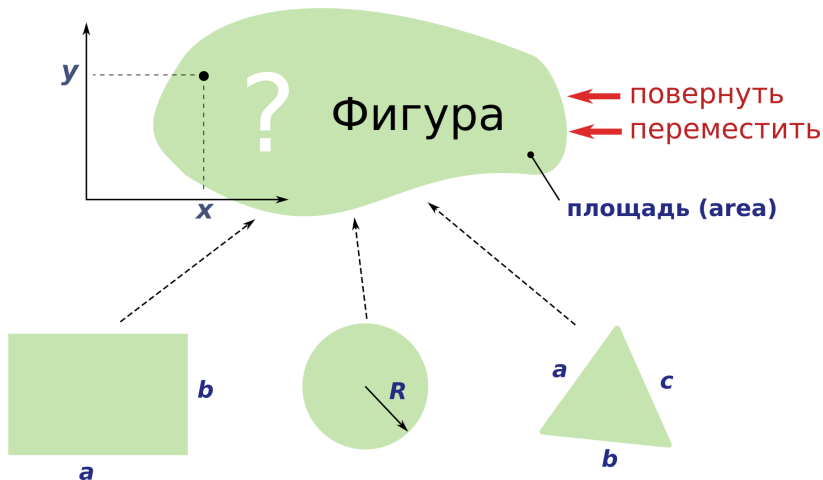
Кафедра теоретической механики



**САМАРСКИЙ** УНИВЕРСИТЕТ  
SAMARA UNIVERSITY

# Класс = данные + методы

Класс – тип данных, состоящий из набора **атрибутов** (свойств) и **методов** – функций для работы с этими атрибутами.



# Объявление класса в Python

Точка с координатами x и y (атрибуты).

```
1 class Point :
2     # Конструктор
3     def __init__(self , coordinates) :
4         self.x = coordinates [0]
5         self.y = coordinates [1]
6     # Переместить точку
7     def move(self , delta) :
8         self.x = self.x + delta [0]
9         self.y = self.y + delta [1]
```

Метод `move` перемещает точку на заданные расстояния вдоль осей x и y.

# Конструктор

Метод класса, вызываемый при создании объекта – представителя класса.

```
1 class Point :
2     # Конструктор
3     def __init__(self, coordinates):
4         self.x = coordinates[0]
5         self.y = coordinates[1]
```

`self` – это ссылка на создаваемый в памяти компьютера объект.

```
6 p1 = Point([1, 3])
7 print(p1.x, p1.y)
```

1 3

При создании объекта (строка 6) вызывается конструктор `__init__` (строка 3), в который передается ссылка на создаваемый объект (`p1`) и список `[1, 3]`.

## Вызов метода класса

```
1 class Point :
2     # Конструктор
3     def __init__(self , coordinates ) :
4         self.x = coordinates [0]
5         self.y = coordinates [1]
6     # Переместить точку
7     def move(self , delta ) :
8         self.x = self.x + delta [0]
9         self.y = self.y + delta [1]
```

```
p1 = Point ([1 , 3])
```

```
p1.move ([2 , 3])
```

```
print (p1.x , p1.y)
```

# Инкапсуляция

# Ограничение доступа к полям класса

Разрешение прямого доступа к свойствам объекта может нарушать его целостность

```
1 class Circle :
2     def __init__(self, x, y, r):
3         self.x = x
4         self.y = y
5         self.r = r
6         self.area = math.pi*r*r
```

```
1 circle = Circle(0, 0, 1)
2 print(circle.area)
```

3.14159265358979

```
1 circle.r = 10
2 print(circle.area)
```

3.14159265358979

# Скрытые свойства класса

Атрибуты, объявленные с одним подчёркиванием **не предназначены** для использования вне класса

```
1 class Circle :
2     def __init__(self , radius) :
3         self._radius = radius
4         self._area = math.pi*radius**2
```

Это просто **соглашение**. Атрибут доступен вне класса:

```
c1 = Circle(10.0)
print(c1._radius)
```

10.0

```
c1._radius = 5
```



# Скрытые свойства класса

Свойства, объявленные с двойным подчёркиванием перед именем недоступны вне класса:

```
1 class Circle :  
2     def __init__(self , radius ) :  
3         self.__radius = radius  
4         self.__area = math.pi*radius**2
```

```
c1 = Circle(10.0)  
print(c1.__radius)
```

AttributeError: 'Circle' object has no attribute '\_\_radius'

# Скрытые свойства класса

Свойства, объявленные с двойным подчёркиванием перед именем недоступны вне класса:

```
1 class Circle :  
2     def __init__(self , radius) :  
3         self.__radius = radius  
4         self.__area = math.pi*radius**2
```

```
c1 = Circle(10.0)  
print(c1.__radius)
```

AttributeError: 'Circle' object has no attribute '\_\_radius'

Если очень хочется:

```
print(c1._Circle__radius)
```

# Доступ при помощи методов

```
1 class Circle :
2     def __init__(self , radius) :
3         self.__radius = radius
4         self.__area = math.pi*radius**2
5         # Получить значение радиуса
6     def get_radius(self) :
7         return self.__radius
8         # Установить значение радиуса и площади
9     def set_radius(self , radius) :
10        self.__radius = radius
11        self.__area    = math.pi*radius**2
12        # Получить значение площади
13    def get_area(self) :
14        return self.__area
```

## Доступ при помощи методов

```
c = Circle(10.0)
```

```
print(c.get_area())
```

314.1592653589793

```
c.set_radius(5)
```

```
print(c.get_area())
```

78.53981633974483

Слишком много скобок...

## Упрощаем: синтаксис `property`

```
1 class Circle :
2     def __init__(self, radius):
3         self.__radius = radius
4         self.__area = math.pi*radius**2
5
6     def get_radius(self):
7         return self.__radius
8     def set_radius(self, radius):
9         self.__radius = radius
10        self.__area = math.pi*radius**2
11
12    def get_area(self):
13        return self.__area
14
15    radius = property(get_radius, set_radius)
16    area = property(get_area)
```

## Синтаксис `property`

```
| c = Circle(10.0)
```

Вызывается `get_radius`

```
| print(c.area)
```

314.1592653589793

Вызывается `set_radius` и `get_radius`

```
| c.radius = 5  
| print(c.area)
```

78.53981633974483

```
| c.area = 0
```

**AttributeError: can't set attribute**

# Синтаксис @property

```
1 class Circle :
2     def __init__(self , radius) :
3         self.__radius = radius
4         self.__area = math.pi*radius**2
5     @property
6     def area(self) :
7         return self.__area
8     @property
9     def radius(self) :
10        return self.__radius
11    @radius.setter
12    def radius(self , radius) :
13        self.__radius = radius
14        self.__area = math.pi*radius**2
```

## Синтаксис @property

```
| c = Circle(10.0)
```

Вызывается `radius(self)`

```
| print(c.area)
```

314.1592653589793

Вызывается `radius(self, radius)` и `radius(self)`

```
| c.radius = 5  
| print(c.area)
```

78.53981633974483

```
| c.area = 0
```

AttributeError: can't set attribute



# Вычисляемые атрибуты (свойства)

```
1 class Circle :
2     def __init__(self , radius):
3         self.__radius = radius
4
5     @property
6     def area(self) :
7         return math.pi*self.__radius**2
8
9     @property
10    def radius(self) :
11        return self.__radius
12
13    @radius.setter
14    def radius(self , radius) :
15        self.__radius = radius
```

## Синтаксис `property`

```
c = Circle(10.0)
```

Вызывается `area(self)`

```
print(c.area)
```

314.1592653589793

Вызывается `radius(self, radius)` и `area(self)`

```
c.radius = 5  
print(c.area)
```

78.53981633974483

# Переопределение атрибутов

```
1 import math
2
3 class Shape:
4
5     def __init__(self, x, y):
6         self.x = x
7         self.y = y
8
9     def move(self, dx, dy):
10        self.x += dx
11        self.y += dy
12
13    @property
14    def area(self):
15        raise NotImplementedError()
```

# Rectangle

```
1 class Rectangle(Shape):  
2  
3     def __init__(self, x, y, a, b):  
4         super().__init__(x, y)  
5         self.a = a  
6         self.b = b  
7  
8     @property  
9     def area(self):  
10        return self.a*self.b
```

# Circle

```
1 class Circle(Shape):  
2  
3     def __init__(self, x, y, r):  
4         super().__init__(x, y)  
5         self.r = r  
6  
7     @property  
8     def area(self):  
9         return math.pi * self.r**2
```

# Фигуры

```
1 figures = []  
2  
3 figures.append( Rectangle(0,0,2,4) )  
4 figures.append( Rectangle(1,15,4,2) )  
5  
6 figures.append( Circle(5,10,5) )  
7 figures.append( Circle(6,7,3) )  
8  
9 sum(fig.area for fig in figures)
```

122.81415022205297

# Полиморфизм

# Перегрузка операторов

Перегрузка операторов – один из способов реализации полиморфизма, когда различные операторы (+, -, /, ...) имеют различный смысл в зависимости от типов аргументов.

- $1 + 3 = 4$   
результат сложения двух целых чисел – их арифметическая сумма
- "1" + "3" = "13"  
результат сложения двух строк – конкатенация (склейка) строк

Python позволяет определять правила выполнения операций для своих типов (классов).



# Перегрузка операторов

Класс `list2` создаётся на основе класса `list`, переопределяя операцию сложения:

```
1 class list2 ( list ) :  
2  
3     def __add__( self , other ) :  
4         return list2 ( [ i [ 0 ] + i [ 1 ] for i in \  
5                             zip ( self , other ) ] )
```

```
1 a = list2 ( [ 1 , 2 , 3 ] )  
2 b = list ( [ 7 , 4 , 1 ] )  
3 c = a + b  
4 print ( c )
```

[8, 6, 4]

# Перегрузка операторов

Класс `list2` создаётся на основе класса `list`, переопределяя операцию сложения и вычитания:

```
1 class list2 ( list ) :
2     def __add__ ( self , other ) :
3         return list2 ( [ i [ 0 ] + i [ 1 ] for i in \
4                         zip ( self , other ) ] )
5     def __sub__ ( self , other ) :
6         return list2 ( [ i [ 0 ] - i [ 1 ] for i in \
7                         zip ( self , other ) ] )
```

```
a = list2 ( [ 1 , 2 , 3 ] )
b = list ( [ 7 , 4 , 1 ] )
c = a - b
print ( c )
```

`[-6, -2, 2]`

# Перегрузка операторов

Перегрузка оператора сравнения (равенство):

```
1 class Circle :
2     def __init__(self , x, y, r):
3         self.x = x
4         self.y = y
5         self.r = r
6     def __eq__(self , other):
7         return self.r == other.r
```

Две окружности равны, если равны их радиусы

```
1 o1 = Circle(0, 0, 2)
2 o2 = Circle(1, 0, 2)
3 o1 == o2
```

True

# Перегрузка операторов

Если операция сравнения для объектов типа `Circle` не определена:

```
1 class Circle :  
2     def __init__(self, x, y, r):  
3         self.x = x  
4         self.y = y  
5         self.r = r
```

Окружности не равны:

```
1 o1 = Circle(1, 0, 2)  
2 o2 = Circle(1, 0, 2)  
3 o1 == o2
```

False

# Перегрузка преобразования в текст

```
1 class Circle :
2     def __init__(self , x, y, r):
3         self.x = x
4         self.y = y
5         self.r = r
6     def __str__(self) :
7         return "Окружность радиуса " + str(self.r)
```

```
1 o1 = Circle(0, 0, 2)
2 print(o1)
```

Окружность радиуса 2

## Перегрузка операторов сравнения

```
<      __lt__(self, other)
<=     __le__(self, other)
==     __eq__(self, other)
!=     __ne__(self, other)
>=     __ge__(self, other)
>      __gt__(self, other)
```

# Перегрузка арифметических операторов

```
+      __add__(self, other)
-      __sub__(self, other)
*      __mul__(self, other)
//     __floordiv__(self, other)
/      __truediv__(self, other)
%      __mod__(self, other)
**     __pow__(self, other)
```

# Классы, модули, структуры?

Советы от Гвидо ван Россума:

- Избегайте усложнения структур данных.
- Кортежи лучше объектов (можно воспользоваться именованными кортежами).
- Предпочитайте простые поля функциям, геттерам и сеттерам.
- Используйте больше чисел, строк, кортежей, списков, множеств, словарей.
- Взгляните также на библиотеку `collections`, особенно на класс `deque`.



# Именованные кортежи

Если необходима структура данных со свойствами (атрибутами, полями) без сложного поведения (вычисляемые поля), необходимости наследования, то лучше использовать более простые типы, например `namedtuple`:

```
1 import collections
2
3 Book = collections.namedtuple( 'Book' ,
4     [ 'name' , 'cost' , 'pages' ])
5
6 book = Book( 'Python cookbook' , 25.0 , 500)
```

```
print( book . pages )
```

500

# Структура

Эмуляция структуры в стиле языка Си:

```
class DataStructure :  
    pass
```

```
a = DataStructure ()
```

```
a.x = 1
```

```
a.y = 2
```

```
print (a.x)
```

# Структура

Создание структуры с функцией инициализации:

```
1 class Struct :
2     def __init__(self , **kwargs) :
3         for k, v in kwargs.items() :
4             setattr(self , k, v)
5
6 class MyStruct ( Struct ) :
7     pass
```

```
ms = MyStruct ( foo = 10, bar = "abc" )
print ( ms.foo )
```

10