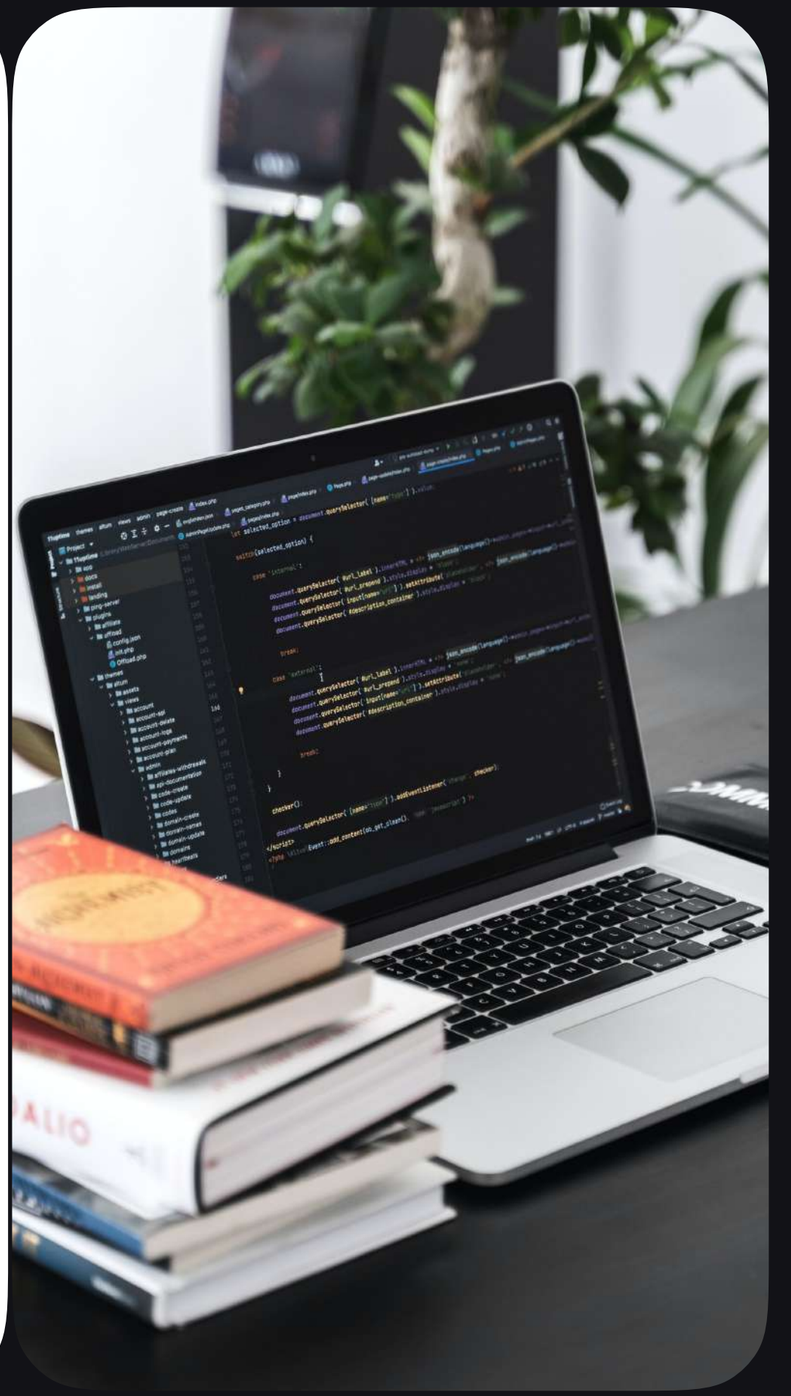
Модуль 2 Занятие 3

Методы классов



Методы классов

Параметр self

Методы как атрибуты класса

Классметоды

Статические методы

Свойства и методы

Класс

Данные

Действия

Класс имеет данные (свойства класса) и действия (методы класса).

Свойства — это данные класса

Методы — это набор функций для работы с классом или объектом данного класса (действия)

Метод класса

```
class Point:
    size = 1
    color = 'black'

    def set_coordinates():
        print('Вызов метода set_coordinates')

print(Point.set_coordinates)
Point.set_coordinates()
```

Метод — это функция, объявленная внутри класса

Добавим метод и вызовем его через класс

Вывод:

```
<function Point.set_coordinates at 0x0000017314497B50>
Вызов метода set_coordinates
```

Метод класса

```
class Point:
    size = 1
    color = 'black'

    def set_coordinates():
        print('Вызов метода set_coordinates')

my_point = Point()
my_point.set_coordinates()
```

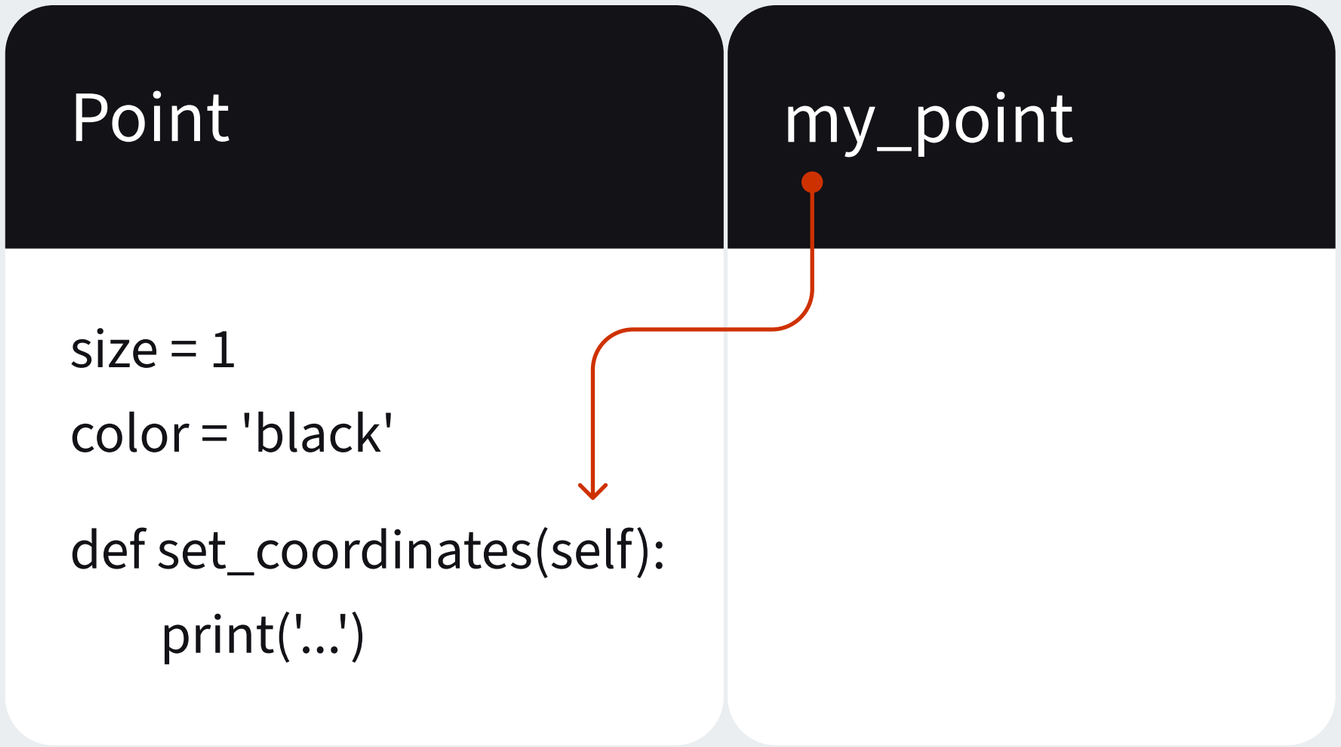
Если вызвать этот метод через объект — возникает ошибка

Почему возникла ошибка?

Ошибка:

```
TypeError: Point.set_coordinates() takes 0 positional arguments but 1 was given
```

Параметр self



При вызове метода через объект, интерпретатор автоматически передает ссылку на этот объект как первый параметр метода. Для имени этого параметра используется имя self.

```
my_point.set_coordinates()
или
Point.set_coordinates(my_point)
```

Метод класса

```
class Point:
    size = 1
    color = 'black'

    def set_coordinates(self):
        print('Вызов метода set_coordinates')

my_point = Point()
my_point.set_coordinates()
```

Исправим ошибку и первым параметром метода класса укажем параметр self

Вывод:

```
Вызов метода set_coordinates
```

Параметр self

```
class Point:
    size = 1
    color = 'black'

    def set_coordinates(self):
        print(id(self))

my_point = Point()
print(id(my_point))
my_point.set_coordinates()
```

Self — это ссылка на экземпляр класса, через который был вызван данный метод.

Вывод:

```
1693132349904
1693132349904
```

Пример

```
class Point:
  size = 1
  color = 'black'

  def set_coordinates(self, x, y):
      self.x = x
      self.y = y

my_point = Point()
my_point.set_coordinates(3, 5)
print(my_point.__dict__)
```

Вывод:

```
{'x': 3, 'y': 5}
```

Так как **self** — это ссылка на конкретный объект **my_point**, то, добавляя атрибуты, мы добавляем атрибуты в конкретный объект **my_point**.

Благодаря параметру **self**, используя один метод, мы можем работать с атрибутами разных объектов.

Пример

```
class Point:
    size = 1
    color = 'black'

    def set_coordinates(self):
        print('Вызов метода set_coordinates')

Point.set_coordinates()
```

Использовать этот метод через класс уже не получится. Вызывая метод через класс мы не передаем никаких ссылок на объект и поэтому получаем ошибку: не был передан один обязательный аргумент.

Ошибка:

```
TypeError: Point.set_coordinates() missing 1 required positional argument: 'self'
```

Метод класса

```
class Point:
    size = 1
    color = 'black'

    def set_coordinates(self, x, y):
        self.x = x
        self.y = y

    def get_coordinates(self):
        return self.x, self.y

my_point = Point()
my_point.set_coordinates(3, 5)
print(my_point)
print(my_point.get_coordinates())
```

Добавим метод get_coordinates(), который будет возвращать значения атрибутов объекта.

Вывод:

```
<__main__.Point object at 0x00000128C53C7E20>
(3, 5)
```

Метод класса

```
class Point:
    size = 1
    color = 'black'

    def set_coordinates(self, x, y):
        self.x = x
        self.y = y

    def get_coordinates(self):
        return self.x, self.y

my_point = Point()
my_point.set_coordinates(3, 5)
print(getattr(my_point, 'get_coordinates')())
```

Методы — тоже атрибуты,
то есть доступ к методам можно получить через функцию getattr, как мы делали ранее.

Вывод:

```
(3, 5)
```

@classmethod и @staticmethod

- **@classmethod** — это метод, работающий с атрибутами класса (а не объекта), через который он был вызван, привязан к классу, а не к объекту.
- **@staticmethod** —метод, который ничего не знает о классе или экземпляре, через который он был вызван, по сути это функция внутри класса, которую можно вызывать без создания экземпляра класса.
- Для создания этих методов используются декораторы **@classmethod** и **@staticmethod.**

Декораторы

Декоратор — это функция-обертка, которая позволяет изменить поведение другой функции, не внося изменения в код.

```
def decorator_function(func):
    res = func()
    return res.upper()

@decorator_function
def hello_world():
    return 'hello world!'

print(hello_world)
```

Функция **decorator_function** меняет поведение функции **hello_world**.
Для этого перед функцией **hello_world** указывается запись:

@decorator_function

Вывод:

```
(3, 5)
```

@classmethod

```
class Point:
    MAX_SIZE = 10

    @classmethod
    def validate_size(cls, size):
        return size <= cls.MAX_SIZE

    def set_size(self, size):
        if self.validate_size(size):
            self.size = size
        else:
            self.size = self.MAX_SIZE

print(Point.validate_size(15))
my_point = Point()
my_point.set_size(15)
print(my_point.size)
```

Метод @classmethod работает только с атрибутами класса. Первым параметром метода указывается параметр cls — это ссылка на текущий класс.

Теперь этот метод можно вызывать через класс и использовать в других методах по необходимости.

Вывод:

```
False
10
```

@staticmethod

```
import math

class Point:
    MAX_SIZE = 10

    @staticmethod
    def get_distance(x, y):
        return math.sqrt(x ** 2 + y ** 2)

print(Point.get_distance(-2, -2))
```

Статический метод не связан
с атрибутами класса и объекта
и поэтому не принимает никаких
ссылок.

Итоги

- **def set_coordinates(self, x, y)**: … — такой метод вызывается через экземпляр класса и работает с атрибутами класса и объекта
- **@classmethod**

  def validate_size(cls, size): … – такой метод вызывается через класс и работает только с атрибутами класса
- **@staticmethod**

  def get_distance(x, y):... – такой метод вызывается через класс и не работает ни с какими с атрибутами