ЯЗЫК ПРОГРАММИРОВАНИЯ PYTHON

Объектно-ориентированное программирование в Python

Крашенинников Роман Сергеевич
Главный специалист отдела системного администрирования РХТУ им. Д.И. Менделеева, ведущий программист кафедры информационных компьютерных технологий

ТЕМЫ

- Основные термины и понятия ООП
- Классы
- Наследование классов
- Магические методы и их переопределение
- Жизненный цикл объекта
- Сокрытие данных в Python
- Свойства, методы класса и статические методы.

ПОЛЕЗНЫЕ РЕСУРСЫ

- Официальная документация
- Основы ООП Python
- Магические методы

ЧТО ТАКОЕ ООП?

Объектно-ориентированное программирование (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Класс — тип, описывающий устройство объектов.

Объект — это экземпляр класса. Класс можно сравнить с чертежом, по которому создаются объекты.

Python соответствует принципам объектно-ориентированного программирования. Кроме того, стоит отметить, что всё в Python является объектами: строки, списки, числа и тд.

КЛАССЫ (СОЗДАНИЕ)

Классы оформляются с помощью ключевого слова class и в виде блока с отступом, содержащего атрибуты класса (которые являются переменными) и методы класса (которые являются функциями).

Общий вид класса

```
class <имя_класса>:
    <атрибуты_класса>
    <методы_класса>
```

Класс собака

```
class Dog:
    breed = "Доберман"

    def bark(self):
        print("Гав-гав")
```

КЛАССЫ (ОБЪЕКТЫ)

Для того, чтобы создать объект класса, используются функции, называемые конструкторами. Они по названию совпадают с названием класса. Иногда конструкторам нужно передать некие параметры (об этом дальше), иногда это не требуется. У объектов можно изменять и получать поля, а так же вызывать методы класса.

Конструктор без параметров

```
my_dog = Dog()
print(my_dog.breed)
my_dog.bark()
```

Конструктор с параметрами

```
my_dog = Dog("Гончая")
print(my_dog.breed)
my_dog.bark()
```

КЛАССЫ (SELF)

self - это ни в коем случае не зарезервированное слово. Это просто название переменной. В методах класса первый параметр функции **по соглашению** именуют **self**, и это ссылка на сам объект этого класса.

```
class Dog:
    breed = "Доберман"

    def bark(self):
        print("Гав-гав")

    def write_breed(self):
        print(self.breed)
```

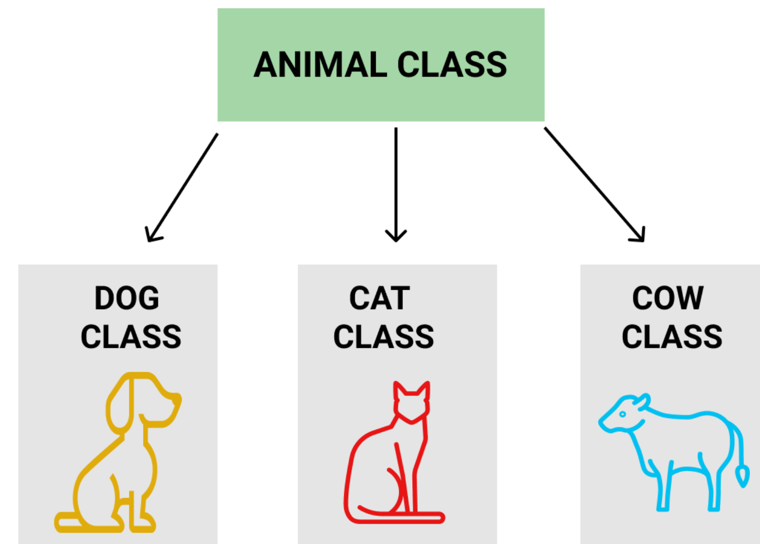
```
my_dog = Dog()
my_dog.breed = "Гончая"
my_dog.write_breed()
```

```
Гончая

Process finished with exit code 0
```

НАСЛЕДОВАНИЕ КЛАССОВ

Наследование классов — очень мощная возможность в объектно-ориентированном программировании. Оно позволяет создавать производные классы (классы наследники), взяв за основу все методы и элементы базового класса (класса родителя). Таким образом экономится масса времени на написание и отладку кода новой программы.

```
1   class Animal:
2       def __init__(self, name, color):
3           self.name = name
4           self.color = color
5
6   class Cat(Animal):
7       def purr(self):
8           print("Purr...")
9
10  class Dog(Animal):
11      def bark(self):
12          print("Woof!")
13
14  fido = Dog("Fido", "brown")
15  print(fido.color)
16  fido.bark()
```



МАГИЧЕСКИЕ МЕТОДЫ

Магические методы, если говорить простыми словами, это такие функции класса, которые переопределяют какое-либо базовое поведения класса. Такое поведение, как создание объекта этого класса, сложение объектов класса, строковое представление объектов и тд. Магические методы начинаются с __ и ими же заканчиваются.

МАГИЧЕСКИЕ МЕТОДЫ (__INIT__)

__init__ - Этот метод используется для определения/инициализации экземпляра. Именно в методе __init__ обычно задаются начальные атрибуты создаваемого экземпляра.

```
class Dog:
    breed = "Доберман"

    def __init__(self, breed=None):
        if breed is not None:
            self.breed = breed

    def bark(self):
        print("Гав-гав")

    def write_breed(self):
        print(self.breed)
```

МАГИЧЕСКИЕ МЕТОДЫ (__STR__)

Метод __str__ может вернуть более описательные данные экземпляра. Следует отметить, что этот метод используется функцией print() для отображения информации экземпляра

```
class Dog:

    def __init__(self, breed=None, age=5):
        self.breed = breed
        self.age = age

    def __str__(self):
        return "{}, {} лет".format(self.breed, self.age)

    def bark(self):
        print("Гав-гав")
```

МАГИЧЕСКИЕ МЕТОДЫ (ЛОГИЧЕСКИЕ МЕТОДЫ)

`__eq__(self, other)`
Определяет поведение оператора равенства, `==` .

`__ne__(self, other)`
Определяет поведение оператора неравенства, `!=` .

`__lt__(self, other)`
Определяет поведение оператора меньше, `<` .

`__gt__(self, other)`
Определяет поведение оператора больше, `>` .

`__le__(self, other)`
Определяет поведение оператора меньше или равно, `<=` .

`__ge__(self, other)`
Определяет поведение оператора больше или равно, `>=` .

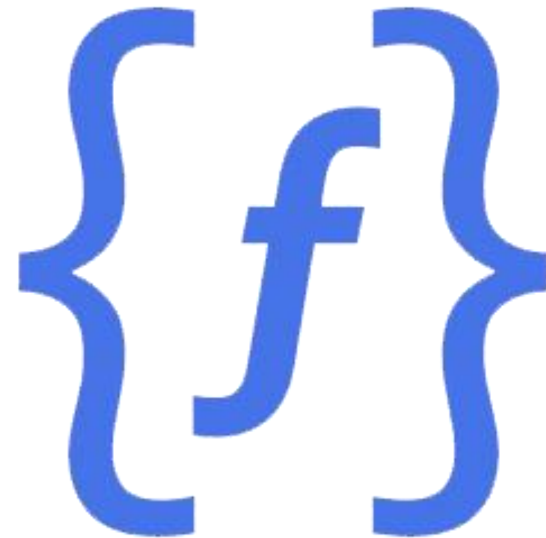
```
class Dog:

    def __init__(self, breed=None, age=5):
        self.breed = breed
        self.age = age

    def __eq__(self, other):
        return self.age == other.age

    def __gt__(self, other):
        return self.age > other.age

    def __ge__(self, other):
        return self.age >= other.age
```

```
my_dog1 = Dog("Доберман", 5)
my_dog2 = Dog(age=5)
print(my_dog2 >= my_dog1)
```

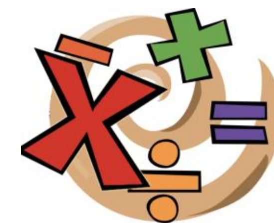
МАГИЧЕСКИЕ МЕТОДЫ (__CALL__)

Позволяет любому экземпляру вашего класса быть вызванным как-будто он функция. __call__ принимает произвольное число аргументов; то есть, вы можете определить __call__ так же как любую другую функцию, принимающую столько аргументов, сколько вам нужно.

```
class Test:

    def __init__(self, n = 1):
        self.n = n

    def __call__(self, a, b):
        return self.n * (a + b)


t = Test(5)

print(t(3, 2))
```



МАГИЧЕСКИЕ МЕТОДЫ (МАТЕМАТИЧЕСКИЕ МЕТОДЫ)



- `__add__(self, other)`
  Сложение.

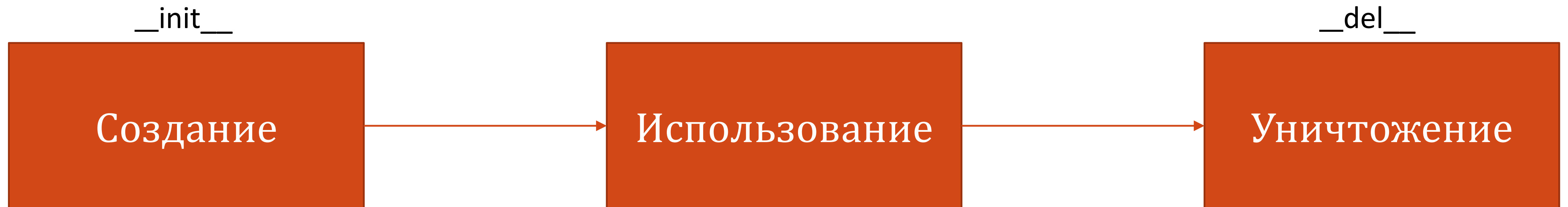
- `__sub__(self, other)`
  Вычитание.

- `__mul__(self, other)`
  Умножение.

- `__div__(self, other)`
  Деление, оператор `/`.

```python
class Rect:

    def __init__(self, a, b):

        self.a = a
        self.b = b

    def __str__(self):

        return f"a={self.a} b={self.b}"

    def __add__(self, other):
        return Rect(self.a + other.a, self.b + other.b)

r1 = Rect(1, 2)
r2 = Rect(2, 3)

r3 = r1 + r2

print(r3)
# a=3 b=5
```

ЖИЗНЕННЫЙ ЦИКЛ



СОКРЫТИЕ ДАННЫХ

Сокрытие данных - понятие, суть которого в том, что детали реализации класса должны быть скрыты, и чистый стандартный интерфейс должен быть представлен тем, кто будет использовать класс. В других языках программирования это достигается с использованием частных методов и атрибутов, которые закрывают доступ извне к определенным методам и атрибутам класса.

Идеология Python несколько иначе. В сообществе Python часто звучит фраза «мы все взрослые и по своему согласию здесь», что означает, что не следует устанавливать свои ограничения на доступ к отдельным частям класса. Так как все равно невозможно обеспечить строгую частность метода или атрибута.

СОКРЫТИЕ ДАННЫХ

Условно частные методы и атрибуты оформляются с единым подчеркиванием в начале имени. Это частные методы, которые не должны взаимодействовать со внешней частью программы. Но часто это правило условно; внешняя часть программы может получить к ним доступ.
Реальная особенность этих методов лишь в том, что from module_name import * не будет импортировать переменные, которые начинаются с единого подчеркивания.

Строго частные методы и атрибуты оформляются с двойным подчеркиванием в начале имени. Таким образом их имена искажаются, и внешняя часть программы не может получить к ним доступ. Но это делается не для того, чтобы обеспечить их частность, а чтобы избежать ошибок, если где-либо в коде есть подклассы, которые имеют методы или атрибуты с такими же именами.

СОКРЫТИЕ ДАННЫХ

Пример частных атрибутов

```python
class MyClass:

    _secret_field = 5
    __super_secret_field = 15

mc = MyClass()

print(mc._secret_field)
# 5
mc._secret_field = 10
print(mc._secret_field)
# 10

print(mc._MyClass__super_secret_field)
# 15
mc._MyClass__super_secret_field = 20
print(mc._MyClass__super_secret_field)
# 20

print(mc.__super_secret_field)
# AttributeError
```

СВОЙСТВА

Свойства позволяют нам создавать вычисляемые поля класса и ограничивать доступ к их изменению извне. Для этого используется декоратор property, при вызове мы обращаемся к свойству по имени функции, но не используем скобки (как у атрибутов).

```
class Rect:
    def __init__(self, h, w):
        self.h = h
        self.w = w

    @property
    def s(self):
        return self.h * self.w
```

МЕТОДЫ КЛАССА

Методы класса несколько отличаются от методов объекта класса: они вызываются классом, который передается параметру cls метода. Чаще всего это используется в фабричных методах: создается экземпляр класса, при этом используются иные параметры, чем те, которые обычно передаются в конструктор класса. Методы класса оформляются с декоратором classmethod.

```
class Rect:
    def __init__(self, h, w):
        self.h = h
        self.w = w

    @classmethod
    def square(cls, h):
        return cls(h, h)

    @property
    def s(self):
        return self.h * self.w
```

МЕТОДЫ КЛАССА

Статические методы похожи на методы класса с тем отличием, что они не берут никаких дополнительных аргументов; они аналогичны обычным функциям класса. Они оформляются с декоратором staticmethod.

```
class Rect:
    def __init__(self, h, w):
        self.h = h
        self.w = w

    @staticmethod
    def print_msg(name):
        print("Я прямоугольник по имени {}".format(name))

Rect.print_msg("Ваня")
```

ЗАДАНИЕ ДЛЯ САМОКОНТРОЛЯ

Задание 1

Используя объектно-ориентированный подход создать программу на языке Python, позволяющую определить температуру смеси по данным двух смешивающихся потоков. Исходя из того, что температуру можно вычислить по формуле:

$$t_m = \frac{m_1 c_{p1} t_1 + m_2 c_{p2} t_2}{m_1 c_{p1} + m_2 c_{p2}}$$

Где m – масса, c_p – удельная теплоёмкость, t – температура.

Задание 2

Попытаться решить задачу с 4-й лекции (про 4 народа), используя объектно-ориентированный подход

СПАСИБО ЗА ВНИМАНИЕ!