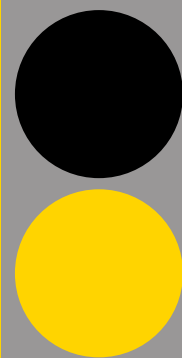


ООП В PYTHON

TEACHER: KSENYA STANISLAVOVNA

OCTOBER 3, 2023



Введение

Суть объектно-ориентированного программирования (ООП) очевидно раскрывается в его названии. Эта парадигма предлагает представлять все компоненты программы как объекты из реальной жизни. У каждого такого объекта есть характеристики и он выполняет определенные функции.

Например, объект может представлять собой человека с такими характеристиками, как имя, возраст и род занятий, а также поведением, таким как ходьба, разговор и бег. Это Дима, ему 18 лет, он спортсмен и сейчас бежит по стадиону, а когда устанет, будет ходить и разговаривать с тренером о том, как повысить выносливость.

Принципы ООП

Основные принципы ООП включают абстракцию, инкапсуляцию, наследование и полиморфизм. Есть также объекты и классы. Вместе они составляют принцип работы любого объектно-ориентированного языка программирования, в том числе Python.

• ИНКАПСУЛЯЦИЯ

Инкапсуляция - это принцип, который позволяет объединить данные и методы, работающие с этими данными, внутри класса. Таким образом, данные и методы становятся связанными и взаимодействуют друг с другом, обеспечивая безопасность и управление доступом к данным. Например, в классе **Person** есть данные **name** и **age**, и метод **say_hello()**, который выводит приветствие с именем объекта **Person**. Вот пример:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```

def say_hello(self):
    print("Привет, меня зовут", self.name)

person = Person("Иван", 25)
person.say_hello()

```

• НАСЛЕДОВАНИЕ

Наследование - это принцип, который позволяет создавать новые классы на основе уже существующих классов. Класс, который наследует свойства и методы другого класса, называется подклассом, а класс, от которого наследуются свойства и методы, называется суперклассом или родительским классом. Подкласс может добавлять новые свойства и методы или переопределять уже существующие. Например, в классе **Dog** мы наследуем от класса **Animal** и переопределяем метод **speak()**, чтобы собака говорила по-своему. Вот пример:

```

class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("Животное говорит")

class Dog(Animal):
    def speak(self):
        print("Собака говорит")

dog = Dog("Бобик")
dog.speak()

```

• ПОЛИМОРФИЗМ

Полиморфизм - это принцип, который позволяет использовать один интерфейс для разных типов данных. Это означает, что объекты разных классов могут обрабатываться одинаково, если они поддерживают общий интерфейс. Например, у нас есть функция **print_sound()**, которая принимает объект **animal** и вызывает его метод **speak()**. Мы можем передать в эту функцию экземпляры классов **Dog** и **Cat**, и они будут издавать разные звуки. Вот пример:

```

def print_sound(animal):

```

```
def print_sound(animals):  
    animal.speak()  
  
dog = Dog("Бобик")  
cat = Cat("Мурзик")  
  
print_sound(dog)  
print_sound(cat)
```

КЛАССЫ

Классы создаются очень просто. Достаточно прописать инструкцию `class` и добавить имя.

```
1 class Car:  
2     pass  
3
```

Для именования классов в Python используют нотацию **CamelCase** — каждое слово начинается с заглавной буквы.

В примере выше тело класса состоит из одного оператора `pass`. Он нужен для того, чтобы интерпретатор Python не выдавал ошибку. Фактически это пустой класс, у которого нет ни атрибутов (характеристик), ни методов. Может возникнуть вопрос: как их добавить? Давайте разбираться.

ДОБАВЛЕНИЕ МЕТОДОВ И АТТРИБУТОВ

Конструктор - это метод, который вызывается при создании классов. Благодаря ему, у объектов изначально есть какие-то значения.

В Python конструктором является метод `__init__()`. В нем указываются все свойства, которыми должны обладать объекты класса **Car**. Каждый раз, когда создается новый объект **Car**, `__init__()` устанавливает начальное состояние, присваивая свойствам значения. То есть `__init__()` инициализирует каждый новый экземпляр класса.

Вы можете указать в `__init__()` любое количество параметров, но первым параметром всегда будет переменная с именем **self**. Когда создается новый экземпляр класса, он автоматически передается self параметру в `__init__()`, чтобы можно было определить новые атрибуты объекта.

Важно: self — не зарезервированное имя. У первого параметра при инициализации может быть и другое название. Однако есть соглашение, что его называют self. Это помогает унифицировать создание классов в разных проектах.

Вот пример класса в Python:

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
```

В теле `__init__()` есть три оператора, использующих переменную self:

- `self.brand = brand` создает атрибут с именем brand и присваивает ему значение параметра brand .
- `self.model = model` создает атрибут с именем model и присваивает ему значение параметра model .
- `self.year = year` создает атрибут с именем year и присваивает ему значение параметра year .

Важно соблюдать отступы. Они показывают, что метод `__init__` принадлежит классу Car, а атрибуты `self.brand`, `self.model` и `self.year` принадлежат методу `__init__`.

Атрибуты экземпляра и атрибуты класса

Атрибуты, созданные в `__init__()`, называются атрибутами экземпляра. Их значения зависят от конкретного экземпляра класса. У всех объектов есть бренд, модель и год. Но значения атрибутов **brand**, **model**, **year** будут различаться в зависимости от объекта.

Чтобы не запутаться, давайте посмотрим примеры. Вернемся к нашему классу Car. Вот он:

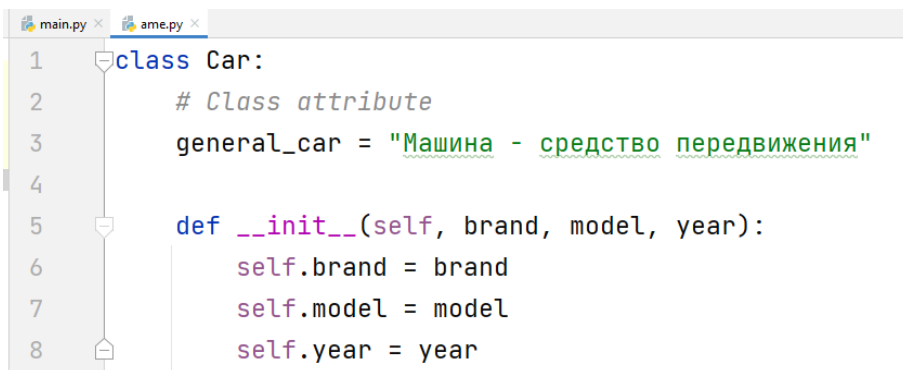
```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
```

```
        self.model = model
    )
    self.year = year
```

Создадим на его основе два объекта — экземпляра класса Car. Первый объект — машина Toyota Camry 2020 года. Второй объект — машина Hyundai Creta 2021. У обоих объектов есть бренд, модель и год. Но значения у них разные. Это достигается благодаря ключевому слову **self**.

Self — указатель на текущий экземпляр класса. Он позволяет работать с отдельными объектами, а не всеми экземплярами, которые принадлежат классу. Благодаря **self** мы можем указать, что марка одной машины Toyota, а второй — Hyundai, что одна машина — 2020 года выпуска, а другая — 2021 года.

Кроме атрибутов экземпляра существуют атрибуты класса. Они имеют одинаковое значение для всех объектов. Вы можете определить атрибут класса, присвоив значение имени переменной за пределами метода `__init__()`. Например, если бы мы создавали класс для кошек, общий атрибут для них было бы сообщение “Кошки милые”. Для машин, общий атрибут может быть таким:



```
1 class Car:
2     # Class attribute
3     general_car = "Машина - средство передвижения"
4
5     def __init__(self, brand, model, year):
6         self.brand = brand
7         self.model = model
8         self.year = year
```

Атрибуты класса определяются под первой строкой имени класса и имеют отступ в четыре пробела. Им всегда должно быть присвоено начальное значение. При создании экземпляра класса автоматически создаются атрибуты класса, которым присваиваются их начальные значения.

Теперь вы можете создать две машины с разными моделями, брендами и годами выпуска. Но их будет объединять атрибут класса **general_car** — обе они средства передвижения. Этот атрибут класса будет появляться во всех экземплярах класса, которые вы будете создавать.

МЕТОДЫ

Кроме `__init__()` могут быть и другие методы. Они делятся на три группы:

- методы экземпляра класса;

- методы класса;
- статические методы.

Чтобы лучше понять разницу между ними, посмотрим пример:

```
class MyClass:
    def method(self):
        return 'instance method called', self
    @classmethod
    def classmethod(cls):
        return 'class method called', cls
    @staticmethod
    def staticmethod():
        return 'static method called'
```

В этом классе есть все три метода.

Первый — метод экземпляра класса. Он принимает параметр `self`, но, как вы видели выше на примере атрибутов, ему можно передавать любые другие параметры. Методы экземпляра класса — самые распространенные, обычно приходится работать именно с ними.

```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Благодаря параметру `self` методы экземпляра класса имеют доступ к атрибутам и методам объектов. Это позволяет изменять состояние объектов.

Второй тип — метод класса. В примере выше он определен с помощью декоратора `@classmethod`. Декораторы — это обертки, которые позволяют менять назначение функций.

Методы класса принимают параметр `cls`. При вызове метода этот параметр указывает не на объект, а на сам класс. Такие методы не могут менять объекты по отдельности — как вы теперь знаете, для этого нужен параметр `self`. Но они могут менять состояние класса в целом, что влияет на все объекты этого класса. Пример метода класса:

```
class Cat:
    def speak(cls):
        print('Мяу!')
```

Теперь вы можете создать экземпляры этого класса — конкретных кошек. Все они будут уметь говорить «Мяу».

Если вы измените метод класса, то это модифицирует всех кошек. Например:

```
class Cat:  
    def speak(cls):  
        print('Гав!')
```

Теперь все кошки будут лаять, а не мяукать.

Статический метод определен с помощью декоратора `@staticmethod`. Он принимает любые параметры в любых количествах, кроме `self` и `cls`.

Статические методы не меняют состояние ни класса, ни объекта, поэтому используются не так часто. Они помещаются в класс для того, чтобы быть в пространстве имен, их используют для организационных целей.

Объекты или экземпляры класса

Чем объекты отличаются от классов

В то время как класс является планом (схемой, чертежом — можно использовать разные сравнения), экземпляр представляет собой объект, созданный из класса и содержащий реальные данные. Экземпляр класса `Cat` — это уже не схема. Это настоящая кошка с именем и возрастом.

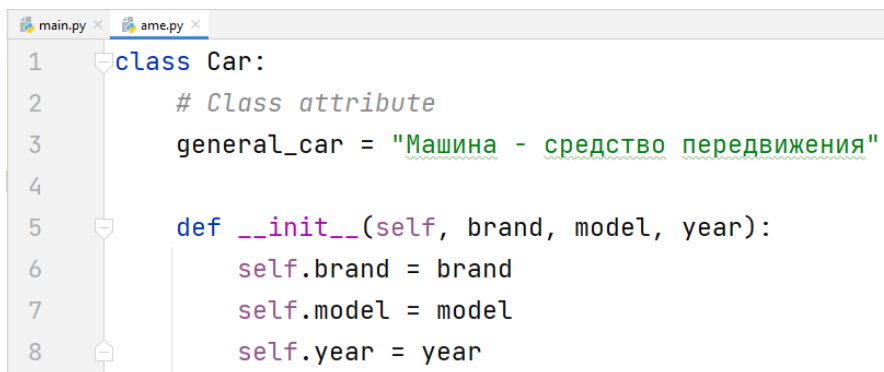
Класс похож на форму или анкету. Экземпляр подобен форме, которая была заполнена информацией. Подобно тому, как многие люди могут заполнять одну и ту же форму со своей собственной уникальной информацией, многие экземпляры могут быть созданы из одного класса.

То есть, основное отличие между классами и объектами заключается в том, что классы являются абстрактными сущностями, определяющими структуру и поведение объектов, в то время как объекты являются конкретными экземплярами классов, которые имеют конкретные значения атрибутов и могут выполнять методы.

Как создать объект класса в Python

Сначала нужно создать класс. На его основе вы будете создавать объекты — разные экземпляры класса.

Допустим, есть класс Car. В нем определены параметры, которые должны быть у каждого объекта этого класса.



```
1 class Car:
2     # Class attribute
3     general_car = "Машина - средство передвижения"
4
5     def __init__(self, brand, model, year):
6         self.brand = brand
7         self.model = model
8         self.year = year
```

Чтобы создать объект или несколько объектов, достаточно передать бренд, модель и год — параметры, которые указаны в классе.

```
car_1 = Car("Toyota", "Camry", 2020)
car_2 = Car("Hyundai", "Creta", 2021)
```

Есть и неочевидный момент. Метод класса `__init__()` имеет четыре параметра. Почему же в примере ему передаются только три аргумента? Никакой магии нет. Когда вы создаете экземпляр объекта, Python создает новый объект и передает его первому параметру `__init__()`. Это по существу удаляет `self`, поэтому вам нужно беспокоиться только о параметрах `brand`, `model` и `year`.

После создания экземпляров вы можете получить доступ к их атрибутам, используя запись через точку. Например:

```
print(car_2.model)
print(car_1.brand, car_1.model)
```