Введение в классы

План урока

1. Введение
2. Основные понятия
3. Классы
4. Примеры классов
5. Соглашения об именовании, вызов методов атрибутов

Аннотация

*Одна из самых распространённых методик разработки программных продуктов — объектно-ориентированное программирование (ООП). При работе с функциями мы уже использовали принцип модульности (сокрытие сложного алгоритма за вызовом функций) и библиотеки (упаковка функций, решающих схожие задачи, в одно хранилище — библиотеку). В ООП появляется ещё один пример модульности — **объект**. Объекты хранят внутри себя и данные, и обрабатывающие их функции. Изучению парадигмы объектно-ориентированного программирования и посвящены следующие уроки.*

Введение

Язык программирования Python появился в 1991 году. К этому времени была разработана теоретическая база объектно-ориентированного программирования, появились исследовательские языки программирования, проверившие эти идеи на практике, и даже возникло первое поколение объектно-ориентированных языков для широкого круга программистов. Поэтому, ориентируясь на чужие успехи и неудачи, Гвидо ван Россум и его коллеги смогли спроектировать достаточно простую и мощную реализацию ООП. Python поддерживает ООП на сто процентов: все данные в нём являются объектами. Числа всех типов, строки, списки, словари, даже функции, модули, и наконец, сами типы данных — всё это объекты!

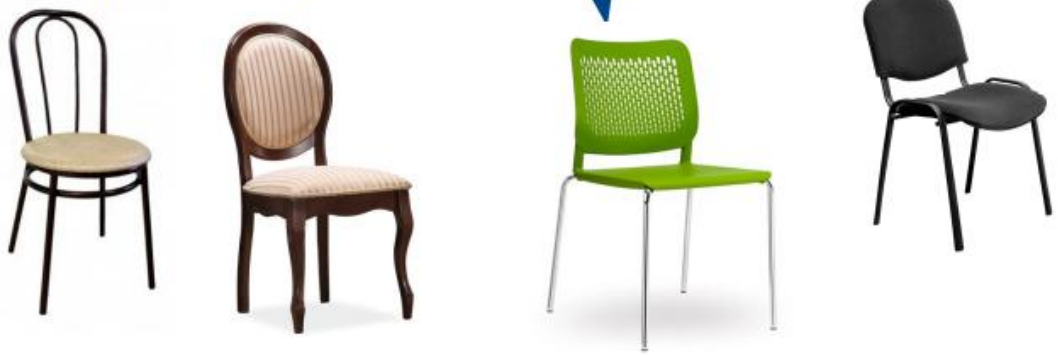
Давайте для начала рассмотрим несколько предметов из реального мира. Вообще говоря, прямой аналогии между объектами материального мира и объектами из мира программирования нет. Ведь в программировании есть объекты, которые обозначают какой-то процесс (например функции) или состояние процесса, или вообще произвольные абстрактные понятия. Например, массив или число с плавающей точкой сами по себе достаточно абстрактные понятия, которые имеют отдалённые аналоги в реальном мире. Но всё же давайте пока порассуждаем о реальных объектах. Представим себе комнату. В ней есть мебель: несколько столов, стулья, шкафы. Стулья могут отличаться цветом, формой, количеством ножек, но всё равно мы всегда сможем отличить стул от шкафа. Если задуматься, у каждого *объекта* есть набор *свойств* и *действия*, в которых он может участвовать. Основываясь на этих *свойствах* (наличие сиденья) и *действиях* (на стуле можно сидеть), мы классифицируем объекты, то есть относим их к тому или иному *классу*.

Класс

Стул

Свойства	Действия:
• число ножек • спинка • материалы	• можно сидеть

Объекты



Основные понятия

Давайте определим несколько терминов:

Класс

Описывает модель объекта, его свойства и поведение. Говоря языком программиста, класс — это такой тип данных, который создается для описания сложных объектов.

Экземпляр

Для краткости вместо «Объект, порождённый классом "Стул"» говорят «экземпляр класса "Стул"».

Объект

Хранит конкретные значения свойств и информацию о принадлежности к классу. Может выполнять методы.

Атрибут

Свойство, присущее объекту. Класс объекта определяет, какие атрибуты есть у объекта. Конкретные значения атрибутов — характеристика уже не класса, а конкретного экземпляра этого класса, то есть объекта.

Метод

Действие, которое объект может выполнять над самим собой или другими объектами.

Чтобы стало чуть понятнее, давайте разберём на примере: `1`, `2`, `3`, `"abc"`, `[10, 20, 30]` – объекты. А `int`, `str`, `list` — классы. Да, все типы данных, которые мы изучали ранее, на самом деле — классы. `1`, `2`, `3` — экземпляры класса `int`, `"abc"` — класса `str`. Объект `[10, 20, 30]` — экземпляр класса `list`, в который вложены экземпляры `int`.

Чтобы узнать, к какому классу относится тот или иной объект, можно воспользоваться функцией **type**.

Например,

```
>>> type(123)
```

```
<class 'int'>
```

```
>>> type([1, 2, 3])
```

```
<class 'list'>
```

Примеры классов

Давайте создадим простейший класс, который будет моделировать обычный фрукт. На языке Python это будет выглядеть так:

```
class Fruit:
    pass
```

Определение этого класса состоит из зарезервированного слова `class`, имени класса и пустой инструкции после отступа. Имена классов по стандарту именования PEP8 должны начинаться с большой буквы. Встроенные классы (`int`, `float`, `str`, `list` и другие) этому правилу не следуют, однако в вашем коде его лучше придерживаться — так делает большинство программистов на Python. Внутри класса с дополнительным уровнем отступов должны определяться его методы, но сейчас их нет. Однако хотя бы одна инструкция должна быть, поэтому приходится использовать пустую инструкцию-заглушку `pass`. Она предназначена как раз для таких случаев.

Описав класс, мы создали модель фрукта.

Теперь создадим два конкретных фрукта — экземпляры класса **Fruit**:

```
a = Fruit()
b = Fruit()
```

Переменные `a` и `b` содержат ссылки на два разных объекта — экземпляра класса **Fruit**, которые можно наделить разными атрибутами:

```
a.name = 'apple'
a.weight = 120
# теперь a - это яблоко весом 120 грамм
b.name = 'orange'
b.weight = 150
# a b - это апельсин весом 150 грамм
```

Атрибуты можно не только устанавливать, но и читать. При чтении ещё не созданного атрибута будет появляться ошибка `AttributeError`. Вы её часто увидите, допуская неточности в именах атрибутов и методов.

```
print(a.name, a.weight)  # apple 120
print(b.name, b.weight)  # orange 150
b.weight -= 10  # Апельсин долго лежал на складе и усох
print(b.name, b.weight)  # orange 140

c = Fruit()
c.name = 'lemon'
c.color = 'yellow'
# Атрибут color появился только в объекте c.
# Забыли добавить свойство weight и обращаемся к нему
print(c.name, c.weight)
# Ошибка AttributeError, нет атрибута weight
```

На данный момент мы пользуемся объектами только для хранения соответствий между именами атрибутов и их значениями. Но на самом деле возможности объектов значительно шире. Давайте рассмотрим, как можно запрограммировать объекты на выполнение определённых действий:

```
class Greeter:
    def hello_world(self):
        print("Привет, Мир!")

greet = Greeter()
greet.hello_world()  # выведет "Привет, Мир!"
```

После беглого осмотра этого кода видно, что внутри класса `Greeter` находится определение чего-то, похожего на функцию, печатающую фразу «Привет, Мир!». На самом деле мы написали метод, с синтаксисом вызова которого вы хорошо знакомы по методу строк `split` или методу списков `append`. Теперь мы можем создавать такие методы самостоятельно. Обратите внимание на два момента:

- Метод должен быть определён внутри класса (добавляется уровень отступов).
- У методов всегда есть хотя бы один аргумент, и **первый по счёту аргумент должен называться self**.

Аргументу `self` следует уделить особое внимание. В него передается тот объект, который вызвал этот метод. Поэтому `self` ещё часто называют «контекстным объектом». В примере выше это объект `greet`.

Вообще говоря **self** — это обычная переменная, которая может называться по-другому. Но так категорически не рекомендуется делать: соглашение об имени контекстного объекта — самое строгое из всех соглашений в мире Питона. Его выполняют 99,9% программистов. Если нарушить это соглашение, другие программисты просто не будут понимать ваш код. Кроме того, некоторые текстовые редакторы подсвечивают слово **self** цветом, и это удобно.

В примере выше мы не передавали нашему методу никаких аргументов. Это довольно скучно, ведь мы уже умеем передавать аргументы функциям. Давайте расширим пример и добавим в наш класс два новых метода:

```
class Greeter:
    def hello_world(self):
        print("Привет, Мир!")

    def greeting(self, name):
        ''' Поприветствовать человека с именем name.'''
        print("Привет, {}!".format(name))

    def start_talking(self, name, weather_is_good):
        ''' Поприветствовать и начать разговор с вопроса о погоде.'''
        print("Привет, {}!".format(name))
        if weather_is_good:
            print("Хорошая погода, не так ли?")
        else:
            print("Отвратительная погода, не так ли?")


greet = Greeter()
greet.hello_world()     # Привет, Мир!
greet.greeting("Петя")  # Привет, Петя!

greet.start_talking("Саша", True)
# Привет, Саша!
# Хорошая погода, не так ли?
```

Значение `self` автоматически получается из объекта, на котором сделан вызов метода, но мы его пока никак не используем.

Давайте попробуем запоминать информацию из предыдущих вызовов методов. Напишем класс «Машина», которую, как известно, надо сначала завести, а потом уже ехать:

```
class Car:
    def start_engine(self):
        engine_on = True  # К сожалению, не сработает

    def drive_to(self, city):
        if engine_on:  # Ошибка NameError
            print("Едем в город {}.".format(city))
        else:
            print("Машина не заведена, никуда не едем")

c = Car()
c.start_engine()
c.drive_to('Владивосток')
```

Итак, первая версия класса «Машина» специально сделана нерабочей, чтобы показать, что переменные внутри методов ведут себя точно так же, как и переменные функций. То есть, если мы инициализируем переменную внутри метода, то после его завершения все созданные таким образом переменные уничтожаются и оказываются недоступны как следующему вызову этого же метода, так и другим методам. Под «уничтожением» мы понимаем исчезновение самих переменных, а не объектов, на которые они ссылаются. Если ссылка на объект сохранилась где-нибудь (например, мы вернули объект с помощью `return`), он всё ещё доступен. Если ссылок не осталось, объект будет скоро переработан сборщиком мусора.

Напомним, что такие переменные называются **локальными**.

Но вернёмся к методам. Пора нашей машине наконец поехать — и в этом нам поможет контекстный объект `self`. Он общий для всех методов класса, и именно в нём мы с помощью атрибутов сохраним информацию о состоянии двигателя:

```
class Car:
    def start_engine(self):
        self.engine_on = True

    def drive_to(self, city):
        if self.engine_on:
            print("Едем в город {}.".format(city))
        else:
            print("Машина не заведена, никуда не едем.")
```

Теперь наша машина отлично заведётся и поедет:

```
car1 = Car()
car1.start_engine()
car1.drive_to('Владивосток')  # Едем в город Владивосток.
```

Однако одна проблема осталась. При попытке выехать на незаведённой машине

```
car2 = Car()
car2.drive_to('Лиссабон')
```

мы вместо красивого сообщения о том, что незаведённая машина не поедет, получим «падение» программы с ошибкой `AttributeError` (отсутствие атрибута или метода). Ещё бы, ведь атрибут создавался в методе `start_engine`, а мы не вызвали его для объекта `car2`.

Кроме того, стоит добавить метод `stop_engine`, чтобы не только заводить машину, но и глушить двигатель. Этот метод помог бы нам избежать вышеуказанной ошибки, но странно «глушить» ещё не заведенную машину, чтобы избежать ошибки: ведь интуитивно мы понимаем, что машина должна создаваться с выключенным двигателем.

Нет ли способа задать значение атрибута `engine_on` по умолчанию? Да. Есть метод `__init__`, который относится к группе так называемых *специальных методов*, которые имеют особое значение для интерпретатора Python. Особое значение метода `__init__` заключается в том, что если такой метод в классе определён, то интерпретатор *автоматически* вызывает его при создании каждого экземпляра этого класса для инициализации экземпляра. Давайте воспользуемся этим, чтобы при создании объекта создать атрибут `engine_on` и записать в него `False`.

```
class Car:
    def __init__(self):
        self.engine_on = False

    def start_engine(self):
        self.engine_on = True

    def drive_to(self, city):
        if self.engine_on:
            print("Едем в город {}.".format(city))
        else:
            print("Машина не заведена, никуда не едем.")

car1 = Car()
car1.start_engine()
car1.drive_to('Владивосток')  # Едем в город Владивосток.
car2 = Car()
car2.drive_to('Лиссабон')     # Машина не заведена, никуда не едем.
```

Метод `__init__` после `self` может получать параметры, передаваемые ему при создании экземпляра:

```
class Car:
    def __init__(self, color):
        self.engine_on = False
        self.color = color

    def start_engine(self):
        self.engine_on = True

    def drive_to(self, city):
        if self.engine_on:
            print("{} машина едет в город {}.".format(self.color, city))
        else:
            print("{} машина не заведена, никуда не едем.".format(self.color))

car1 = Car('красная')  # Создали машину красного цвета

car2 = Car('синяя')    # И ещё одну синего

car1.start_engine()
# Обратите внимание, что мы завели только одну машину,
# ту, на которую ссылается car1 (красную).

# car2 -- это другой объект, он не изменится.

car1.drive_to('Владивосток')
# красная машина едет в город Владивосток.

car2.drive_to('Лиссабон')
# синяя машина не заведена, никуда не едем.
```

Ещё раз обратите внимание на комментарии в тексте. Они показывают, что при записи атрибутов в `self` метод изменяет только свой контекстный объект.

Обсудим ещё один вопрос: зачем нам понадобился метод `start_engine`, ведь его можно было бы заменить строчкой `car.engine_on = True`? Казалось бы, это лишнее усложнение. На самом деле нет. При дальнейшей разработке нашей программы может оказаться, что завести двигатель можно только в машине, в которой есть бензин. Если бы мы в нескольких десятках мест программы написали `car.engine_on = True`, нам пришлось бы найти все эти места и вставить в них проверку на наличие бензина в баке. А с методом `start_engine` мы можем изменить только этот метод.

Такая технология сокрытия информации о внутреннем устройстве объекта за внешним интерфейсом из методов называется *инкапсуляцией*. Надо стараться делать интерфейс методов достаточно полным. Тогда вы, как и другие программисты, будете пользоваться этими методами, а изменения в атрибутах не будут расползаться по коду, использующему ваш класс. Кроме того, инкапсуляция позволяет шире использовать такое понятие, как полиморфизм. О нём мы поговорим на следующем уроке.

В некоторых языках программирования автор класса может закрыть доступ к атрибутам извне класса и заставить всех использующих его программистов работать только с методами. К сожалению, в Питоне так делать нельзя, однако стоит по возможности пользоваться только методами.

Соглашения об именовании, вызов методов атрибутов

Давайте разберёмся с ещё одним примером. Напишем класс робота-почтальона, который должен разносить письма в определённые дома и квартиры. (Для простоты, считаем, что робот обслуживает одну улицу, и не будем её указывать.) Класс назовём длинным именем `RoboticMailDelivery`, чтобы показать, как в Python принято называть классы с длинным составным именем. Имя должно начинаться с большой буквы, между словами не должно быть прочерка, каждое слово внутри имени должно начинаться с большой буквы.

Текст примера:

```
class RoboticMailDelivery:

    def __init__(self):
        self.house_flat_pairs = []

    def add_mail(self, house_number, flat_number):
        '''Добавить информацию о доставке письма по номеру дома
        house_number, квартира flat_number.'''
        self.house_flat_pairs.append((house_number, flat_number))

    def flat_numbers_for_house(self, house_number):
        '''Вернуть список квартир в доме house_number,
        в которые нужно доставить письма.'''
        flat_numbers = []
        for h, f in self.house_flat_pairs:
            if h == house_number:
                flat_numbers.append(f)
        return flat_numbers
```

Метод `add_mail` добавляет кортеж (номер_дома, номер_квартиры) в список-атрибут с помощью метода append. Как видно, вызовы методов для объектов-атрибутов производят обычным образом, вызов метода дописывается справа от объекта: `self.house_flat_pairs.append(...)`

В данном примере специально использованы длинные избыточные имена, чтобы показать, что для имён атрибутов и методов применяются те же правила, что и для имён переменных и функций. Имя должно быть записано в нижнем регистре, слова внутри имени разделяются подчёркиванием: `flat_numbers_for_house`, `house_flat_pairs`.

Документация с описанием методов записывается в `'''многострочных строках'''` перед первой инструкцией как в функциях, так и в методах.