Оглавление:

Классы в Python 2

Принципы ООП: абстракция, инкапсуляция, наследование, полиморфизм 3

1. Абстракция 3

2. Инкапсуляция 4

3. Наследование 5

4. Полиморфизм 6

Свойства (поля/атрибуты) класса в Python 7

1. Статические поля (они же атрибуты или свойства класса) 7

2. Динамические поля (атрибуты или свойства экземпляра класса) 8

Методы (функции) класса в Python 9

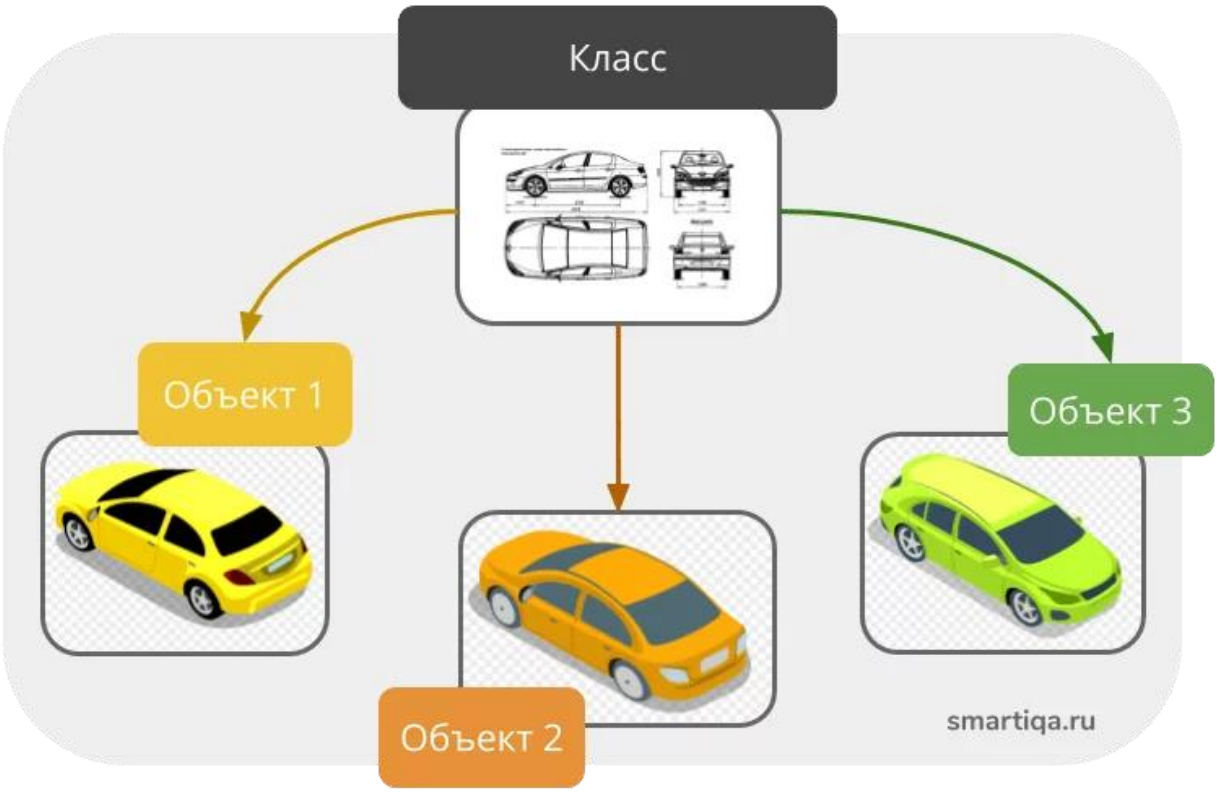
1. Методы экземпляра класса (Обычные методы) 10

2. Статические методы 11

3. Методы класса 12

Уровни доступа атрибутов в Python 13

Классы в Python



Класс — в объектно-ориентированном программировании, представляет собой шаблон для создания объектов, обеспечивающий начальные значения состояний: инициализация полей-переменных и реализация поведения функций или методов.

Объект — некоторая сущность в цифровом пространстве, обладающая определённым состоянием и поведением, имеющая определенные свойства (поля) и операции над ними (методы). Как правило, при рассмотрении объектов выделяется то, что объекты принадлежат одному или нескольким классам, которые определяют поведение (являются моделью) объекта. Термины «экземпляр класса» и «объект» взаимозаменяемы.

На что необходимо обратить внимание?

1. **Класс** описывает множество объектов, имеющих общую структуру и обладающих одинаковым поведением. Класс - это шаблон кода, по которому создаются объекты. Т. е. сам по себе класс ничего не делает, но с его помощью можно создать объект и уже его использовать в работе.
2. Данные внутри класса делятся на свойства и методы.
3. **Свойства класса** (они же поля/атрибуты) - это характеристики объекта класса.
4. **Методы класса** - это функции, с помощью которых можно оперировать данными класса.
5. **Объект** - это конкретный представитель класса.
6. Объект класса и **экземпляр класса** - это одно и то же.

Принципы ООП: абстракция, инкапсуляция, наследование, полиморфизм

1. Абстракция

Абстракция - принцип ООП, согласно которому объект характеризуется свойствами, которые отличают его от всех остальных объектов и при этом четко определяют его концептуальные границы.

Т. е. абстракция позволяет:

1. Выделить главные и наиболее значимые свойства предмета.
2. Отбросить второстепенные характеристики.

2. Инкапсуляция

Инкапсуляция - принцип ООП, согласно которому сложность реализации программного компонента должна быть спрятана за его интерфейсом.

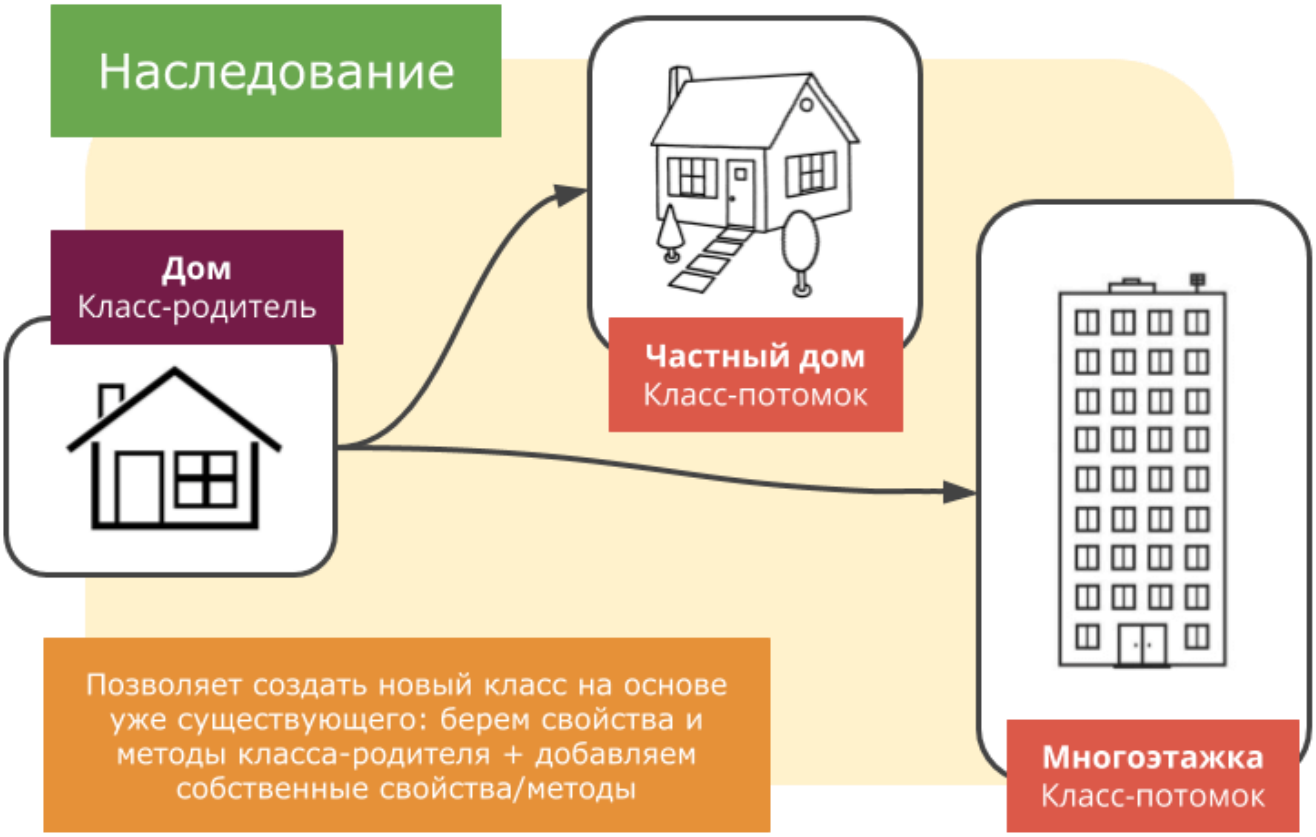
На что обратить внимание?

1. Отсутствует доступ к внутреннему устройству программного компонента.
2. Взаимодействие компонента с внешним миром осуществляется посредством интерфейса, который включает публичные методы и атрибуты (поля).

Для чего нужна инкапсуляция?

1. Инкапсуляция упрощает процесс разработки, т. к. позволяет нам не вникать в тонкости реализации того или иного объекта.
2. Повышается надежность программ за счет того, что при внесении изменений в один из компонентов, остальные части программы остаются неизменными.
3. Становится более легким обмен компонентами между программами.

3. Наследование



Наследование - способ создания нового класса на основе уже существующего, при котором класс-потомок заимствует свойства и методы родительского класса и также добавляет собственные.

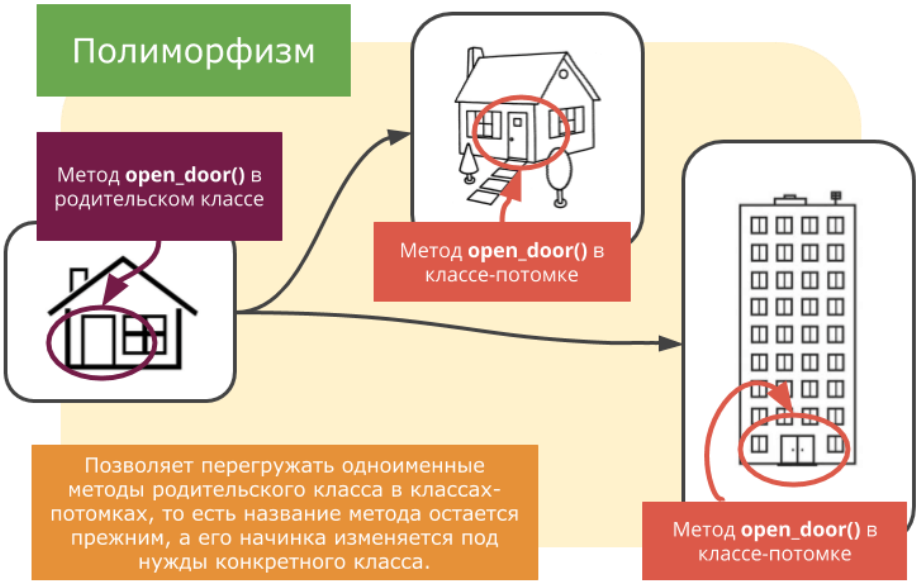
На что обратить внимание?

1. Класс-потомок = Свойства и методы родителя + Собственные свойства и методы.
2. Класс-потомок автоматически наследует от родительского класса все поля (атрибуты) и методы.
3. Класс-потомок может дополняться новыми свойствами.
4. Класс-потомок может дополняться новыми методами, а также заменять(переопределять) унаследованные методы. Переопределить родительский метод - это как? Это значит, внутри класса потомка есть метод, который совпадает по названию с методом родительского класса, но функционал у него новый - соответствующий потребностям класса-потомка.

Для чего нужно наследование?

1. Дает возможность использовать код повторно. Классы-потомки берут общий функционал у родительского класса.
2. Способствует быстрой разработке нового ПО на основе уже существующих открытых классов.
3. Наследование позволяет делать процесс написания кода более простым.

4. Полиморфизм



Полиморфизм - это поддержка нескольких реализаций на основе общего интерфейса.

Другими словами, полиморфизм позволяет перегружать одноименные методы родительского класса в классах-потомках.

Полиморфизм как один из ключевых элементов ООП существует независимо от наследования. Классы могут быть не родственными, но иметь одинаковые методы.

Полиморфизм дает возможность реализовывать так называемые единые интерфейсы для объектов различных классов.

Например, два разных класса содержат метод *total*, однако инструкции каждого предусматривают совершенно разные операции. Так в классе T1 – это арифметическое суммирование аргументов *a, b;* в T2 – подсчет длины строки, полученной конкатенацией (сложение строк) аргументов *a, b*. В зависимости от того, к объекту какого класса применяется метод *total*, выполняются те или иные инструкции.

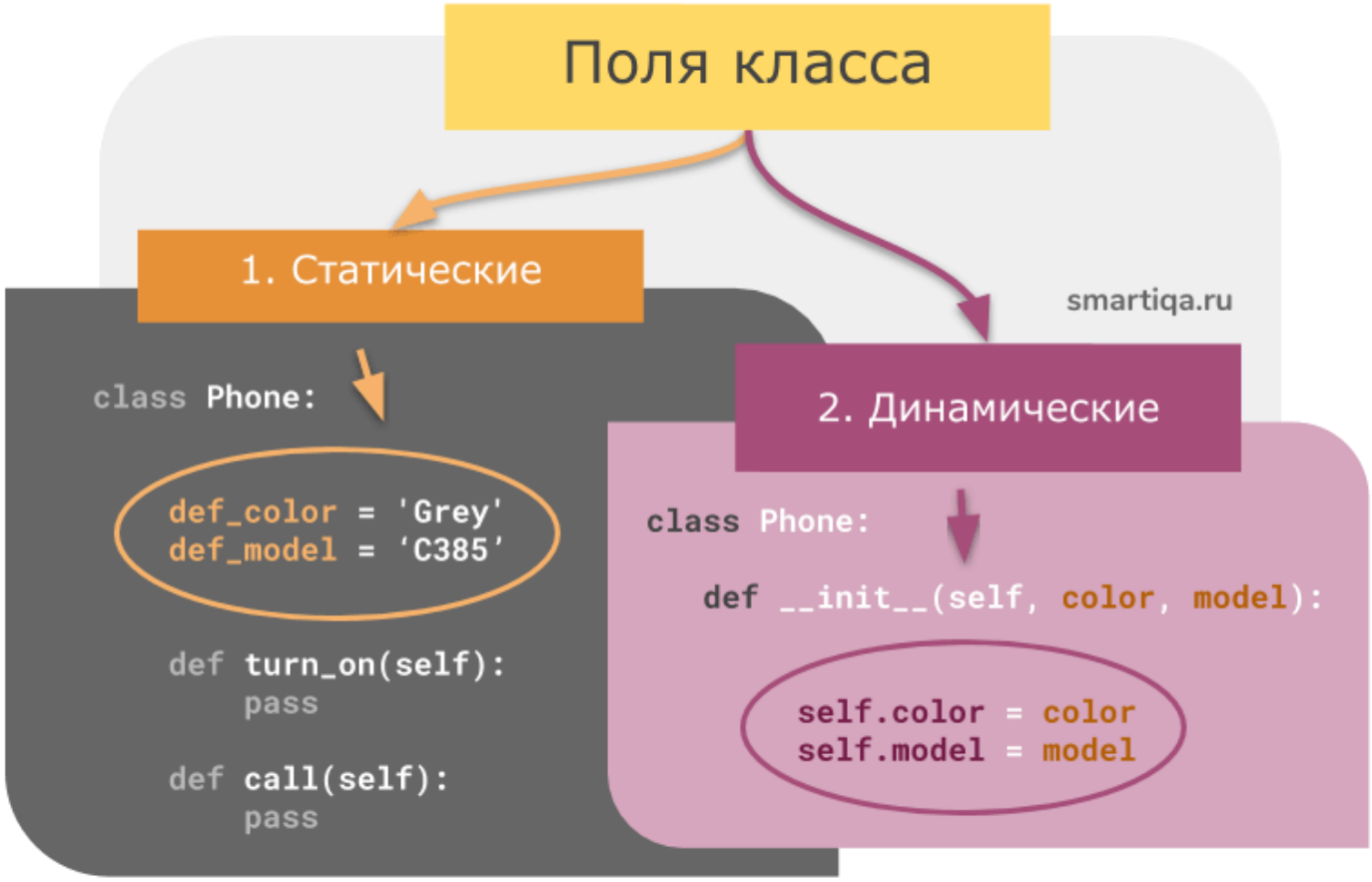
```
class T1:
    def __init__(self):
        self.string = 'Hi'
    def total(self, a, b):  #  Метод total считает сумму чисел a, b
        return int(a) + int(b)
class T2:
    def __init__(self):
        self.string = 'Hi'
    def total(self, a, b):  #  Метод total считает длину строки a + b
        return len(str(a) + str(b))
#  Создаем экземпляры классов и вызываем полиморфный метод total
t1 = T1()
t2 = T2()
print(t1.total(1, 2))   # Вывод: 3
print(t2.total(1, 2))   # Вывод: 2
```

Свойства (поля/атрибуты) класса в Python

Свойства (они же атрибуты или поля) – это переменные, определенные внутри класса.

Атрибуты можно (так же условно) разделить на две группы:

1. Статические – относящиеся к классу
2. Динамические – относящиеся к экземпляру класса



1. Статические поля (они же атрибуты или свойства класса)

Это переменные, которые объявляются внутри тела класса и создаются тогда, когда создается класс. Создали класса - создалась переменная:

```
class Phone:
    # Статические атрибуты (переменные класса)
    default_color = 'Grey'
    default_model = 'C385'

    def turn_on(self):
        pass
    def call(self):
        pass
```

2. Динамические поля (атрибуты или свойства экземпляра класса)
Это переменные, которые создаются на уровне экземпляра класса. Нет экземпляра - нет его переменных. Для создания динамического свойства необходимо обратиться к *self* внутри метода:

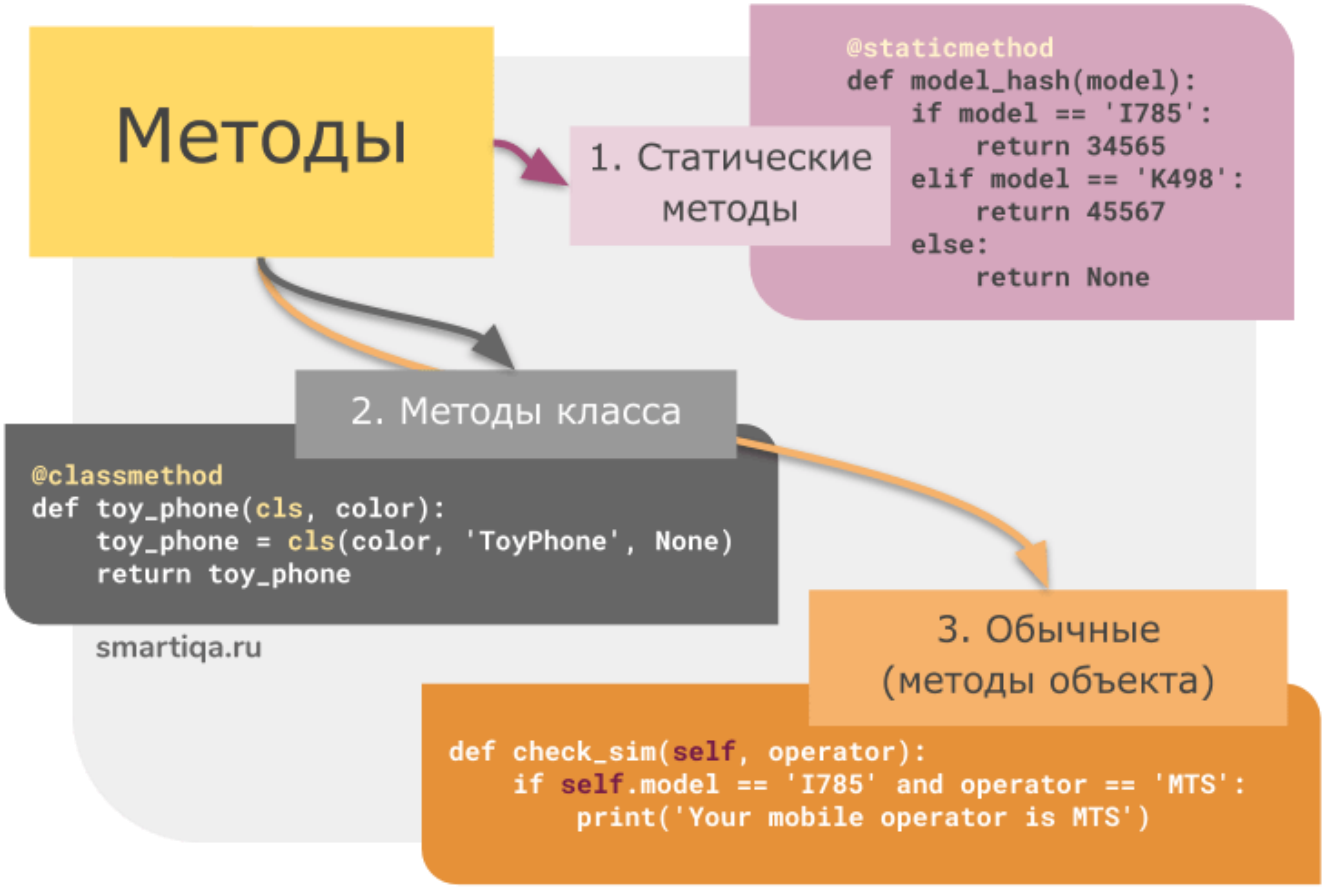
```
class Phone:

    # Статические атрибуты (переменные класса)
    default_color = 'Grey'
    default_model = 'C385'


    def __init__(self, color, model):
        # Динамические атрибуты (переменные экземпляра)
        self.color = color
        self.model = model


# Создаем экземпляр класса:
my_phone = Phone('red', 'I785')
# Прочитаем динамические поля объекта:
print(my_phone.color)  #  Выводит: red
print(my_phone.model)  #  Выводит: I785
```

Служебное слово ***self*** - это ссылка на текущий экземпляр класса. Как правило, эта ссылка передается в качестве первого параметра метода Python.

Стоит обратить внимание, что на самом деле слово *self* не является зарезервированным. Просто существует некоторое соглашение, по которому первый параметр метода именуется *self* и передает ссылку на текущий объект, для которого этот метода был вызван. Хотите назвать первый параметр метода по-другому - пожалуйста.

В других языках программирования(например, Java или C++) аналогом этого ключа является служебное слово *this*.

Методы (функции) класса в Python



Как вы уже знаете, функции внутри класса называются **методами**. Методы так же бывают разными, а именно - их можно разделить на 3 группы:

1. Методы экземпляра класса (они же обычные методы)
2. Статические методы
3. Методы класса

1. Методы экземпляра класса (Обычные методы)

Это группа методов, которые становятся доступны только после создания экземпляра класса, то есть чтобы вызвать такой метод, надо обратиться к экземпляру. Как следствие - первым параметром такого метода является слово *self*. И как мы уже обсудили выше, с помощью данного параметра в метод передается ссылка на объект класса, для которого он был вызван. Теперь пример:

```
class Phone:
    def __init__(self, color, model):
        self.color = color
        self.model = model
    # Обычный метод, первый параметр метода - self
    def check_sim(self, mobile_operator):
        if self.model == 'I785' and mobile_operator == 'MTS':
            print('Your mobile operator is MTS')


# Создаем экземпляр класса
my_phone = Phone('red', 'I785')
# Обращаемся к методу check_sim() через объект my_phone
my_phone.check_sim('MTS')  #  Выводит: Your mobile operator is MTS
```

2. Статические методы

Статические методы - это обычные функции, которые помещены в класс для удобства и тем самым располагаются в области видимости этого класса. Чаще всего это какой-то вспомогательный код.

Важная особенность заключается в том, что данные методы можно вызывать посредством обращения к имени класса, создавать объект класса при этом не обязательно. Именно поэтому в таких методах не используется *self* - этому методу не важна информация об объектах класса.

Чтобы создать статический метод в Python, необходимо воспользоваться специальным декоратором - *@staticmethod*. Выглядит это следующим образом:

```
class Phone:
    # Статический метод справочного характера
    # Возвращает хэш по номеру модели
    # self внутри метода отсутствует
    @staticmethod
    def model_hash(model):
        if model == 'I785':
            return 34565
        elif model == 'K498':
            return 45567
        else:
            return None
    # Обычный метод:
    def check_sim(self, mobile_operator):
        pass


# Вызываем статический метод model_hash, просто обращаясь к имени класса
# Объект класса Phone при этом создавать не надо
print(Phone.model_hash('I785'))  #  Выводит: 34565
```

3. Методы класса

Методы класса являются чем-то средним между обычными методами (привязаны к объекту) и статическими методами (привязаны только к области видимости). Как легко догадаться из названия, такие методы тесно связаны с классом, в котором они определены.

Обратите внимание, что такие методы могут менять состояние самого класса, что в свою очередь отражается на ВСЕХ экземплярах данного класса. Правда при этом менять конкретный объект класса они не могут (этим занимаются методы экземпляра класса).

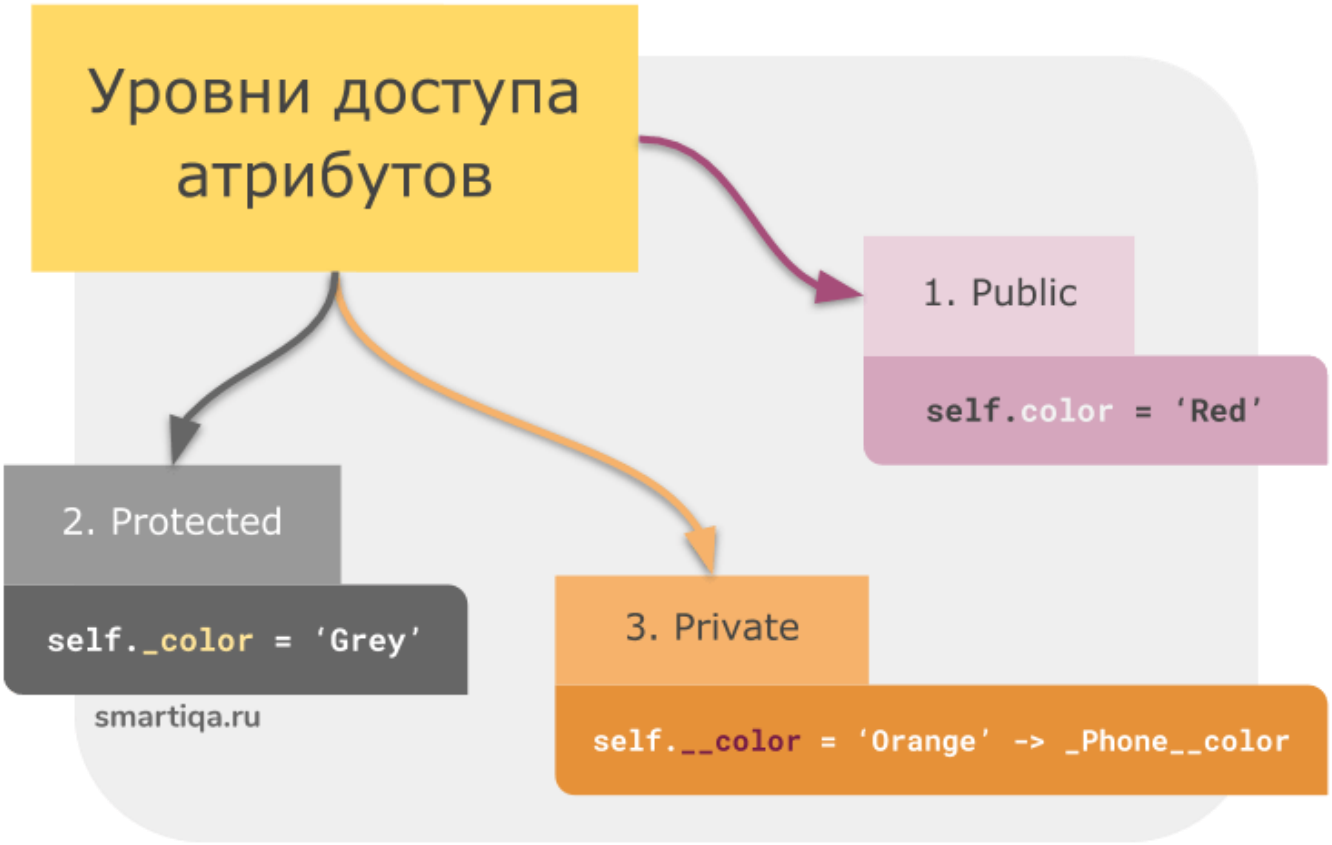
Чтобы создать метод класса, необходимо воспользоваться соответствующим декоратором - *@classmethod*. При этом в качестве первого параметра такого метода передается служебное слово *cls*, которое в отличие от *self* является ссылкой на сам класс (а не на объект). Рассмотрим пример:

```
class Phone:
    def __init__(self, color, model, os):
        self.color = color
        self.model = model
        self.os = os
    # Метод класса
    # Принимает 1) ссылку на класс Phone и 2) цвет в качестве параметров
    # Создает специфический объект класса Phone(игрушечный телефон)
    @classmethod
    def toy_phone(cls, color):
        toy_phone = cls(color, 'ToyPhone', None)
        return toy_phone
    # Статический метод
    @staticmethod
    def model_hash(model):
        pass
    # Обычный метод:
    def check_sim(self, mobile_operator):
        pass
# Создаем объект игрушечный телефон
# Обращаемся к методу класса toy_phone через имя класса и точку
my_toy_phone = Phone.toy_phone('Red')
print(my_toy_phone)  #  Выводит: <phone.Phone object at 0x101a236d0>
```

Как видно из примера, методы класса часто используются, когда

1. Необходимо создать специфичный объект текущего класса
2. Нужно реализовать фабричный паттерн - создаём объекты различных унаследованных классов прямо внутри метода

Уровни доступа атрибутов в Python



В классических языках программирования (таких как C++ и Java) доступ к ресурсам класса реализуется с помощью служебных слов *public*, *private* и *protected*:

1. ***Private***. Приватные члены класса недоступны извне - с ними можно работать только внутри класса.
2. ***Public***. Публичные методы наоборот - открыты для работы снаружи и, как правило, объявляются публичными сразу по-умолчанию.
3. ***Protected***. Доступ к защищенным ресурсам класса возможен только внутри этого класса и также внутри унаследованных от него классов (иными словами, внутри классов-потомков). Больше никто доступа к ним не имеет

С точки зрения разграничения доступа к атрибутам класса Python является особенным языком - в нем отсутствует механизм, который мог бы запретить доступ к переменной или методу внутри класса. Вместо этого создатели Python предложили соглашение, в соответствии с которым:

1. Если переменная/метод начинается с одного нижнего подчеркивания (*_protected_example*), то она/он считается защищенным (*protected*).
2. Если переменная/метод начинается с двух нижних подчеркиваний (*__private_example*), то она/он считается приватным (*private*).