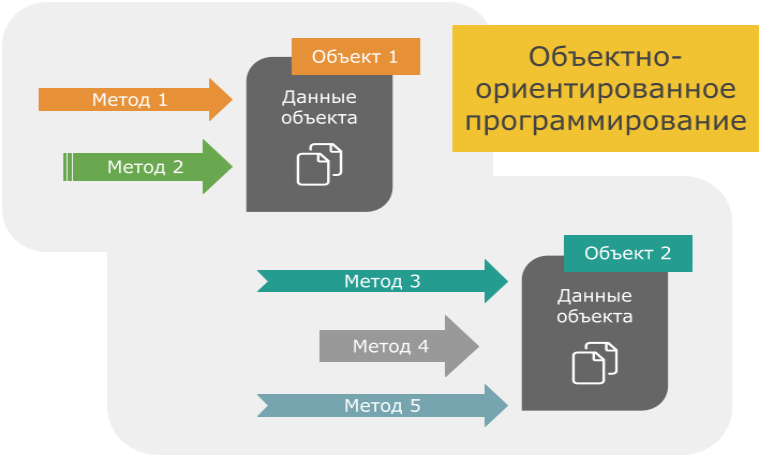
Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

В основе ООП лежит простая идея, в соответствии с которой главное в программе – это данные. Именно они определяют, какие методы будут использоваться для их обработки. Т. е. данные первичны, код для обработки этих данных - вторичен.



Итак, чем же хорош подход ООП?

1. Программа разбивается на объекты. Каждый объект отвечает за собственные данные и их обработку. Как результат - код становится проще и читабельней.

2. Уменьшается дупликация кода. Нужен новый объект, содержимое которого на 90% повторяет уже существующий? Давайте создадим новый класс и унаследуем эти 90% функционала от родительского класса!

3. Упрощается и ускоряется процесс написания программ. Можно сначала создать высокоуровневую структуру классов и базовый функционал, а уже потом перейти к их подробной реализации.

Объекты и классы в ООП

Мир, в котором мы живем, состоит из объектов. Это деревья, солнце, горы, дома, машины, бытовая техника. Каждый из этих объектов имеет свой набор характеристик и предназначение. Несложно догадаться, что именно объектная картина реального мира легла в основу ООП. Разберем несколько ключевых понятий, основываясь на Википедии:

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 (атрибуты) 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. **Свойства класса** (они же поля или атрибуты) - это характеристики объекта класса.
3. **Методы класса** - это функции, с помощью которых можно оперировать данными класса.
4. **Объект** - это конкретный представитель класса.
5. Объект класса и **экземпляр класса** - это одно и то же.

Класс = Свойства + Методы

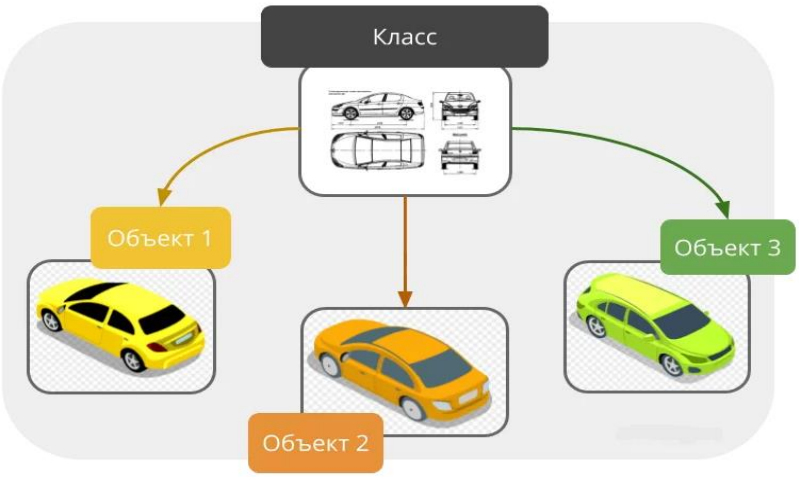
Рассмотрим простой пример. Перед нами класс "Автомобиль". Если мыслить абстрактно, то он представляет собой набор чертежей и инструкций, с помощью которых можно собрать машину. При этом каждая машина, которую мы будем собирать, должна обладать рядом характеристик, которые соответствуют нашему классу. Как мы уже выяснили, данные характеристики - называются **свойствами класса** и в нашем примере могут быть следующими:

1. Цвет
2. Объем двигателя
3. Мощность
4. Тип коробки передач

Так же наш автомобиль может выполнять какие-то действия, характерные для всего класса. Эти действия, как мы теперь знаем, есть **методы класса**, и выглядеть они могут вот так:

1. Ехать
2. Остановиться
3. Заправиться
4. Поставить на сигнализацию
5. Включить дворники

Что общего будет в объектах? Все объекты создаются по одному шаблону, то есть на выходе обязательно будут машины, никаких велосипедов и мотоциклов. Они будут выкрашены в какой-то цвет, ехать они будут за счет наличия в них двигателя, скорость будет регулироваться с помощью коробки передач. Также объекты данного класса будут обладать одинаковыми методами - все машины этого класса будут ездить, периодически им будет нужна заправка, а от угона они будут защищены установкой сигнализации.



Но в чем разница? Значения свойств будут различаться. Одна машина красная, другая - зеленая. У одной объем двигателя 1968 см3 и коробка-робот, а у другой - 1395 см3 и ездить владельцу придется на механике.

Вывод: Объекты класса на выходе похожие и одновременно разные. Различаются, как правило, свойства. Методы остаются одинаковыми.

Пример созданного объекта "Автомобиль Volkswagen Tiguan":

Свойства: Цвет="Белый", Объем двигателя="1984 см3", Мощность="180 л.с.", Тип коробки передач="Робот"

Методы: Ехать, Остановиться, Заправиться, Поставить на сигнализацию, Включить дворники

Принципы ООП: абстракция, инкапсуляция, наследование, полиморфизм

Как мы уже сказали, на текущий момент ООП является самой востребованной и распространенной парадигмой программирования. Концепция ООП строится на основе 4 принципов, которые мы предлагаем вам кратко рассмотреть.

Принцип 1. Абстракция

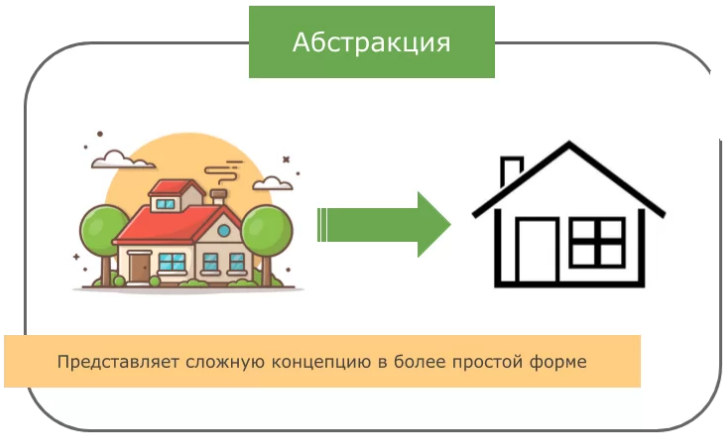
Абстракция - принцип ООП, согласно которому объект характеризуется свойствами, которые отличают его от всех остальных объектов и при этом четко определяют его концептуальные границы.

Т. е. абстракция позволяет:

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

Когда мы имеем дело с составным объектом - мы прибегаем к абстракции. Например, мы должны понимать, что перед нами абстракция, если мы рассматриваем объект как "дом", а не совокупность кирпича, стекла и бетона. А если уже представить множество домов как "город", то мы снова приходим к абстракции, но уже на уровень выше.

Зачем нужна абстракция? Если мыслить масштабно – то она позволяет бороться со сложностью реального мира. Мы отбрасываем все лишнее, чтобы оно нам не мешало, и концентрируемся только на важных чертах объекта.



Принцип 2. Инкапсуляция

Абстракция утверждает следующее: "Объект может быть рассмотрен с общей точки зрения". А инкапсуляция от себя добавляет: "И это единственная точка зрения, с которой вы вообще можете рассмотреть этот объект.". А если вы внимательно посмотрите

на название, то увидите в нем слово "капсула". В этой самой "капсуле" спрятаны данные, которые мы хотим защитить от изменений извне.

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 поля.

А теперь опять пример с домом. Как в данном случае будет работать инкапсуляция? Она будет позволять нам смотреть на дом, но при этом не даст подойти слишком близко. Например, мы будем знать, что в доме есть дверь, что она коричневого цвета, что она открыта или закрыта. Но каким способом и из какого материала она сделана, инкапсуляция нам узнать не позволит.

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.

Принцип 3. Наследование

Наследование используется в случае, если одни объекты аналогичны другим за исключением нескольких различий

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 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.

Снова перед нами объект **Дом**. Дом можно построить, отремонтировать, заселить или снести. В нем есть фундамент, крыша, окна и двери. В виде списка это может выглядеть следующим образом:

СВОЙСТВА
1) Тип фундамента
2) Материал крыши
3) Количество окон
4) Количество дверей

МЕТОДЫ
1) Построить
2) Отремонтировать
3) Заселить
4) Снести

А если мы захотим создать объект **Частный дом**? Данный объект по-прежнему будет являться домом, а значит будет обладать свойствами и методами класса **Дом**. Например, в нем так же есть окна и двери, и такой дом по-прежнему можно построить или отремонтировать. Однако при этом у него также будут собственные свойства и методы, ведь именно они отличают новый класс от его родителя. Новый класс **Частный дом** может выглядеть следующим образом:

СВОЙСТВА
1) Тип фундамента (УНАСЛЕДОВАНО)
2) Материал крыши (УНАСЛЕДОВАНО)
3) Количество окон (УНАСЛЕДОВАНО)
4) Количество дверей (УНАСЛЕДОВАНО)
5) Количество комнат
6) Тип отопления
7) Наличие огорода

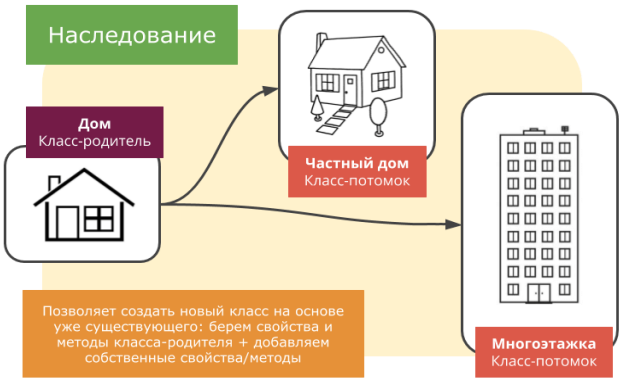
МЕТОДЫ
1) Построить (УНАСЛЕДОВАНО)
2) Отремонтировать (УНАСЛЕДОВАНО)
3) Заселить (УНАСЛЕДОВАНО)
4) Снести (УНАСЛЕДОВАНО)
5) Изменить фасад
6) Утеплить
7) Сделать пристройку

С такой же легкостью мы можем создать еще один класс-потомок - **Многоэтажный дом**. Его свойства и методы могут выглядеть так:

СВОЙСТВА
1) Тип фундамента (УНАСЛЕДОВАНО)
2) Материал крыши (УНАСЛЕДОВАНО)
3) Количество окон (УНАСЛЕДОВАНО)
4) Количество дверей (УНАСЛЕДОВАНО)
5) Количество квартир
6) Количество подъездов
7) Наличие коммерческой недвижимости

МЕТОДЫ
1) Построить (УНАСЛЕДОВАНО)
2) Отремонтировать (УНАСЛЕДОВАНО)
3) Заселить (УНАСЛЕДОВАНО)
4) Снести (УНАСЛЕДОВАНО)
5) Выбрать управляющую компанию
6) Организовать собрание жильцов
7) Нанять дворника

Т. е. наследование позволяет нам использовать функционал уже существующих классов для создания новых.



Принцип 4. Полиморфизм

В данном случае глубоко вдаваться в подробности не будем(так как по-хорошему это тема для целой статьи), но суть данного принципа постараемся передать.

Полиморфизм - это поддержка нескольких реализаций на основе общего интерфейса.

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

Также для понимания работы этого принципа важным является понятие абстрактного метода:

Абстрактный метод (он же виртуальный метод) - это метод класса, реализация для которого отсутствует.

А теперь попробуем собрать все воедино. Для этого снова обратимся к классам **Дом**, **Частный дом** и **Многоквартирный дом**. Предположим, что на этапе написания кода мы еще знаем, какой из домов(частный или многоэтажный) нам предстоит создать, но вот то, что какой-то из них придется строить, мы знаем наверняка. В такой ситуации поступают следующим образом:

1. В родительском классе(в нашем случае - класс **Дом**) создают пустой метод(например, метод **Построить()**) и делают его абстрактным.
2. В классах-потомках создают одноименные методы, но уже с соответствующей реализацией. И это логично, ведь например, процесс постройки **Частного** и **Многоквартирного дома** отличается кардинально. К примеру, для строительства **Многоквартирного дома** необходимо задействовать башенный кран, а **Частный дом** можно построить и собственными силами. При этом данный процесс все равно остается процессом строительства.
3. В итоге получаем метод с одним и тем же именем, который встречается во всех классах. В родительском - имеем только интерфейс, реализация отсутствует. В классах-потомках - имеем и интерфейс и реализацию. Причем в отличие от родительского класса реализация в потомках уже становится обязательной.
4. Теперь мы можем увидеть полиморфизм во всей его красе. Даже не зная, с объектом какого из классов-потомков мы работаем, нам достаточно просто вызвать метод **Построить()**. А уже в момент исполнения программы, когда класс объекта станет известен, будет вызвана необходимая реализация метода **Построить()**.

Как итог - за одинаковым названием могут скрываться методы с совершенно разным функционалом, который в каждом конкретном случае соответствует нуждам класса, к которому он относится.

Классы и объекты в Python

Теперь давайте посмотрим, как реализуется ООП в рамках языка программирования Python. Синтаксис для создания класса выглядит следующим образом:

Синтаксис

```
class <название_класса>:
   <тело_класса>
```

Пример объявления класса с минимально возможным функционалом:

Python

```
class Car:
   pass
```

Для задания класса используется инструкция **class**, далее следует имя класса (обязательно с большой буквы PEP8) **Car** и двоеточие. После них идет тело класса, которое в нашем случае представлено оператором **pass**. Данный оператор сам по себе ничего не делает - фактически это просто заглушка.

Чтобы создать объект класса, нужно воспользоваться следующим синтаксисом:
Синтаксис
<имя_объекта> = <имя_класса>()

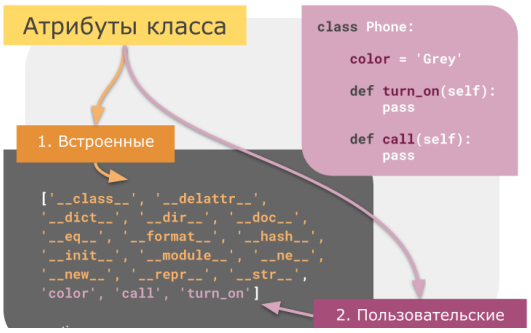
Примера создания объекта класса **Car**:
Python
```
car_object = Car()
```

Атрибуты класса в Python

Давайте договоримся, что атрибутом класса/объекта мы будем называть любой элемент класса/объекта (переменную, метод, подкласс), на который мы можем сослаться через символ точки. Т. е. вот так: **MyClass.<атрибут>** или **my_object.<атрибут>**.

Все атрибуты можно разделить на 2 группы:

1. Встроенные (служебные) атрибуты
2. Пользовательские атрибуты



1. Встроенные атрибуты

Называть данную группу атрибутов встроенными - это своего рода условность, и вот почему. Суть в том, что на самом деле все классы в Python (начиная с 3-й версии) имеют один общий родительский класс - **object**. Это значит, что когда вы создаете новый класс, вы неявно наследуете его от класса **object**, и потому свежесозданный класс наследует его атрибуты. Именно их мы и называем встроенными (служебными). Вот некоторые из них (заметьте, что в списке есть как поля, так и методы):

Атрибут	Назначение	Тип
__new__(cls[, ...])	Конструктор. Создает экземпляр (объект) класса. Сам класс передается в качестве аргумента.	Функция
__init__(self[, ...])	Инициализатор. Принимает свежесозданный объект класса из конструктора.	Функция
__del__(self)	Деструктор. Вызывается при удалении объекта сборщиком мусора	Функция
__str__(self)	Возвращает строковое представление объекта.	Функция
__hash__(self)	Возвращает хэш-сумму объекта.	Функция
__setattr__(self, attr, val)	Создает новый атрибут для объекта класса с именем attr и значением val	Функция
__doc__	Документация класса.	Строка (тип str)
__dict__	Словарь, в котором хранится пространство имен класса	Словарь (тип dict)

В теории ООП **конструктор** класса - это специальный блок инструкций, который вызывается при создании объекта. При работе с питоном может возникнуть мнение, что метод **__init__(self)** - это и есть конструктор, но это не совсем так. На самом деле, при создании объекта в Python вызывается метод **__new__(cls, *args, **kwargs)** и именно он является конструктором класса.

Также обратите внимание, что **__new__()** - это метод класса, поэтому его первый параметр **cls** - ссылка на текущий класс. В свою очередь, метод **__init__()** является так называемым **инициализатором** класса. Именно этот метод первый принимает созданный конструктором объект. Метод **__init__()** часто переопределяется внутри класса самим программистом. Это позволяет со всем удобством задавать параметры будущего объекта при его создании.

2. Пользовательские атрибуты

Это атрибуты, которые непосредственно составляют основной функционал класса. Если служебные атрибуты наследуются от базового класса **object**, то пользовательские - пишутся программистом во время реализации начинки класса и дальнейшей работы с ним.

Список атрибутов класса / объекта можно получить с помощью команды **dir()**. Если взять самый простой класс:

Python

```
class Phone:
    pass
```

То мы получим вот такой список атрибутов:

Python

```
dir(Phone)
```

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']

Как видим, в нем есть только встроенные атрибуты, которые наш класс по-умолчанию унаследовал от базового класса **object**. А теперь добавим ему функционала:

Синтаксис

```
class Phone:

    color = 'Grey'

    def turn_on(self):
        pass

    def call(self):
        pass
```

И теперь посмотрим, как изменился список атрибутов класса:

Python

```
dir(Phone)
```

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',

'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'color', 'call', 'turn_on']

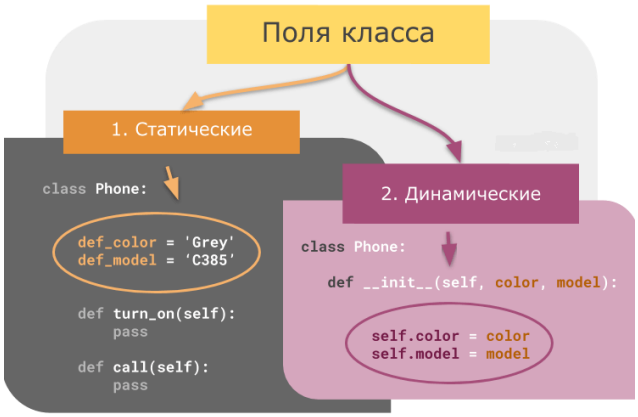
Несложно заметить, что в конце списка добавились три новых пользовательских атрибута: переменная **color** и методы **turn_on()** и **call()**.

Поля (свойства) класса в Python

Поля (они же свойства или переменные) можно (так же условно) разделить на две группы:

1. Статические поля
2. Динамические поля

В чем же разница?



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

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

Python

```
class Phone:

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

    def turn_on(self):
        pass

    def call(self):
        pass
```

Вот так выглядит список атрибутов класса после его создания:

Python

```
dir(Phone)
```

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'call', 'default_color', 'default_model', 'turn_on']

Кроме того, у нас есть возможность получить или изменить такое свойство, просто обратившись к самому классу по его имени (экземпляр класса при этом создавать не нужно).

Python - Интерактивный режим

```
>>> Phone.default_color
'Grey'

# Изменяем цвет телефона по умолчанию с серого на черный
>>> Phone.default_color = 'Black'
>>> Phone.default_color
'Black'
```

2. Динамические поля (переменные 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:

Python

```
class Phone:

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

    def __init__(self, color, model):
        # Динамические поля (переменные объекта)
        self.color = color
        self.model = model
```

Python - Интерактивный режим

```
# Создадим экземпляр класса Phone - телефон красного цвета модели 'I495'
>>> my_phone_red = Phone('Red', 'I495')

# Полный список атрибутов созданного экземпляра:
>>> dir(my_phone_red)
'__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'color', 'default_color', 'default_model',
'model']

# Прочитаем статические поля объекта
>>> my_phone_red.default_color
'Grey'
>>> my_phone_red.default_model
'C385'

# Прочитаем динамические поля объекта
>>> my_phone_red.color
'Red'
>>> my_phone_red.model
'I495'
```

 # Создадим еще один экземпляр класса Phone - такой же телефон, но другого цвета

```
>>> my_phone_blue = Phone('Blue', 'I495')
```

 # Прочитаем динамические поля объекта

```
>>> my_phone_blue.color
'Blue'
>>> my_phone_blue.model
'I495'
```

Что такое self в Python?

Аргументу **self** следует уделить особое внимание. В него передается тот объект, который вызвал этот метод. Поэтому self еще часто называют контекстным объектом. Рассмотрим чуть подробнее. Когда программа вызывает метод объекта, Python передает ему в первом аргументе экземпляр вызывающего объекта, который всегда связан с параметром self. Иными словами, greet.hello_world() преобразуется в вызов Greeter.hello_world(greet). Этот факт объясняет особую важность параметра self и то, почему он должен быть первым в любом методе объекта, который вы пишите. Вызывая метод, вы не должны передавать значение для self явно – интерпретатор сделает это за вас.

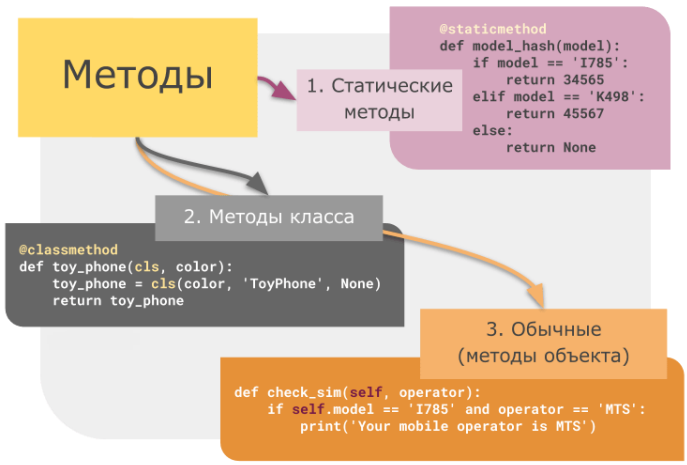
Вообще говоря, self – обычная переменная, которая может называться по другому. Но так категорически не рекомендуется делать: соглашение об имени контекстного объекта – самое строгое из всех соглашений в Python. Его выполняют 99,9% программистов. Если нарушить это соглашение, другие программисты просто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

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

А в чем отличие между ними, давайте разбираться.



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:

Python

```
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')
```

Python - Интерактивный режим

```
# Создаем экземпляр класса
>>> my_phone = Phone('red', 'I785')

# Обращаемся к методу check_sim() через объект my_phone
>>> my_phone.check_sim('MTS')
Your mobile operator is MTS
```

Стоит заметить, что, как правило, данная группа методов является самой многочисленной и часто используемой в сравнении со статическими методами и методами класса.

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:

Python

```
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
```

Python - Интерактивный режим

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

Python

```
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(особенность объекта в том, что это игрушечный телефон)
    # При этом вызывается инициализатор класса Phone
    # которому в качестве аргументов мы передаем цвет и модель,
    # соответствующую созданию игрушечного телефона
    @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
```

Python - Интерактивный режим

```
# Создаем объект игрушечный телефон
# Обращаемся к методу класса toy_phone через имя класса и точку
>>> my_toy_phone = Phone.toy_phone('Red')
>>> my_toy_phone
<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

Пример

Создадим класс Greeter добавим в него несколько методов

```
1  # -*- coding: utf-8 -*-
2  class Greeter:
3      pass
4
5      def hello_world(self):
6          print("Доброе утро!")
7      def greeting(self,name):
8          #Поприветствовать человека с именем name
9          print("Доброе утро,{}!".format(name))
10
```

Создадим экземпляр класса

```
10
11 greet=Greeter()
12
```

```
In [30]: greet.hello_world()
Доброе утро!
```

```
In [36]: greet.greeting('Лариса Васильевна')
Доброе утро,Лариса Васильевна!
```

Создадим еще один метод

```
10      def start_talking(self,name,weather_is_good):
11          #Поприветствовать и начать разговор о погоде
12          print("Доброе утро,{}!".format(name))
13          if weather_is_good:
14              print('Хорошая погода, не так ли?')
15          else:
16              print('Отвратительная погода, не так ли?')
17
```

```
10
11 greet=Greeter()
12
```

```
In [41]: greet.start_talking('Мария Александровна','weather_is_good')
Доброе утро,Мария Александровна!
Хорошая погода, не так ли?
```

Задание 1

Постановка задачи: Написать класс Car для описания поездки на машине цветом …..в город …..

Создать **методы**, которые задают **цвет машины**, **работу двигателя**, **пункт назначения.**

Метод решения: Создаем класс Car с 3-мя методами **color**, **start_engine**, **drive_to**. Пользователь вводит состояние двигателя, цвет машины и пункт назначения. Если состояние двигателя –не включен, “машина никуда не едет”.

Задание 2

Постановка задачи: Написать класс Balance для описания весов с двумя чашами. Требуется определить положение чаш. Создать методы для веса на левой чаше, правой и для вывода результатов сравнения.

Метод решения: Создаем класс Balance для описания весов с двумя чашами. На левую и правую чашу объекта будут добавляться грузы с различным весом, требуется определить положение чаш. Для этого создаем метод add_right принимает целое число — вес, положенный на правую чашу весов, add_left — на левую чашу. Метод result должен возвращать сообщение "Вес на правой и левой чашах одинаков", если вес на чашах одинаковый, "Вес на правой чаше больше" — если перевесила правая, "Вес на левой чаше больше" — если перевесила левая.

Задание 3

Постановка задачи: Написать класс FindWords, который должен анализировать текст и находить в нём слова наименьшей и наибольшей длины. Если одно из самых коротких слов встретилось в исходных предложениях несколько раз, оно должно столько же раз повториться в списке самых коротких слов. Самые длинные слова наоборот должны входить в список без повторов.

Метод решения: Создаем класс FindWords. Вводимый пользователем текст, состоящий из предложений, добавляется в обработку методом add_sentence. Метод shortest_words возвращает список самых коротких на данный момент слов, метод longest_words — самых длинных. Слова, возвращаемые методами shortest_words и longest_words, должны быть отсортированы по алфавиту.

Полезные ссылки: https://proproprogs.ru/python_oop