# Парадигма ООП, базовые приемы работы с классом

[Смотреть материал на видео](https://www.youtube.com/watch?v=Yt08fz52Cj0&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/paradigma-oop>

Сразу следует отметить, что ООП – это совершенно другая парадигма программирования, нежели структурное программирование, основу которого в основном образует функциональное программирование. По сути, мы с вами рассматривали структурное программирование на начальных занятиях по Python. Здесь же перейдем на следующий уровень написания программ в соответствии с ООП. Но что это такое и чем оно кардинально отличается от структурного подхода? Общий принцип я, обычно, объясняю на примере из реальной жизни – кофемолки. Представьте, что некий инженер решил создать электрическую кофемолку. До этого все кофемолки были ручные. Что же он сделал? Он взял механическую часть ручной кофемолки, поставил электрический мотор, добавил кнопку включения на корпусе и получил новый прибор. То есть, для создания нового прибора он использовал уже существующие наработки, а не делал все с нуля. Ровно это лежит в концепции **наследования** ООП. Мы можем брать ранее созданные классы, выполняющие определенные задачи и, затем, видоизменять их под текущий проект. В этом случае создается новый класс на основе существующего и дополняется необходимым функционалом. Это первый столп (наследование), на котором базируется ООП.

Далее, чтобы новая кофемолка представляла собой единый объект, а не два разрозненных (старая + электрический мотор), в новом созданном классе можно вызывать функции наследуемого класса так, словно они определены в новом. Это называется **полиморфизмом** в ООП. Благодаря полиморфизму программист будет работать с новым классом как с единым объектом, используя всю необходимую функциональность, несмотря на то, что он является лишь производным от другого класса. А про базовый класс он может даже и не знать.  Это второй столп ООП – полиморфизм.

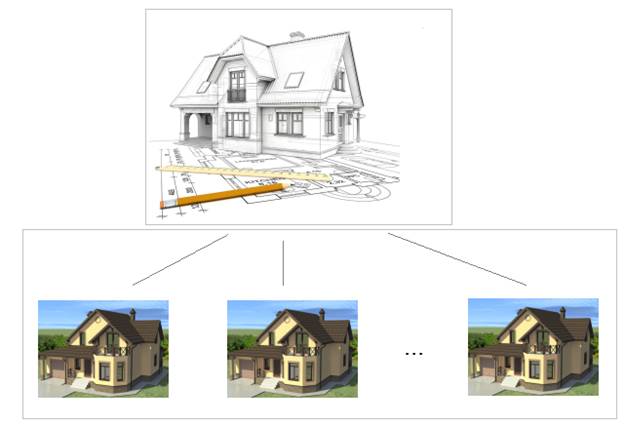
Наконец, третье и последнее. Хозяин новой электрической кофемолки имеет возможность только засыпать в нее зерна, нажимать на кнопку «мотор» и высыпать уже перемолотый кофе. Вся начинка: мотор, лопасти, электрическая схема – скрыты от пользователя. Для достижения результата ему эта начинка не нужна. Достаточно лишь знать последовательность действий и больше ничего. Ровно это делает инструмент **инкапсуляция** в ООП. Инкапсуляция – это возможность закрывать данные и методы класса от внешнего вмешательства. И вся работа с классом возможна только через разрешенные интерфейсные методы и реже непосредственно через данные.

Вот так в целом можно представить себе три основные концепции, лежащие в основе парадигмы ООП:

* инкапсуляция;
* наследование;
* полиморфизм.

На последующих занятиях мы с вами подробно разберем работу этих концепций на конкретных примерах. Но прежде чем начать, следует разобраться в базовых моментах, а именно: что же из себя представляет класс в Python? Когда мы говорили о наследовании, то затронули это понятие и сейчас пора его конкретизировать.

Класс можно воспринимать как некую схему, чертеж, по которому конструируются его экземпляры. Представьте, что нам нужно построить дом. С чего начинается строительство? Да, с плана дома, его чертежей и схем. А уже потом по этому плану строится здание. И таких однотипных зданий можно построить множество. Вот также следует воспринимать класс – это схема для построения однотипных **экземпляров** данного класса.



Чтобы в Python определить класс, записывается ключевое слово class, и через пробел указывается его имя:

class Point:

    x = 1

    y = 1

Обратите внимание, в соответствии со стандартом PEP8 имя класса принято записывать с заглавной буквы. И, конечно же, называть так, чтобы имя отражало суть этого класса. В дальнейшем мы будем придерживаться этого правила.

Сразу же здесь отмечу, что по тому же стандарту PEP8 первая строка в классе автоматически считается его описанием, например, так:

class Point:

    "Класс для представления координат точек на плоскости"

    x = 1

    y = 1

И мы впоследствии можем вывести это описание с помощью предопределенного свойства \_\_doc\_\_:

print(Point.\_\_doc\_\_)

Вообще существует множество предопределенных свойств, например:

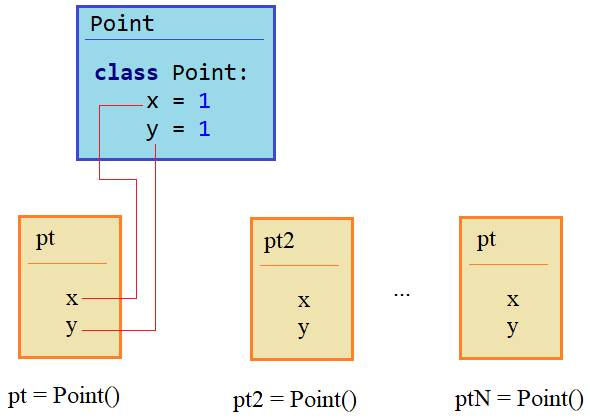
print(Point.\_\_name\_\_)

возвращает имя класса. Мы с ними будем постепенно знакомиться по мере необходимости. Полный набор данных класса или экземпляра можно увидеть с помощью функции dir:

print( dir(pt) )

Итак, мы определили класс с именем Point для представления координат точек на плоскости. Это проект, чертеж, и чтобы по этому проекту создать экземпляр данного класса, следует записать так:

pt = Point()



И вот здесь появляется первый важный момент. Нам нужно четко понимать: что из себя представляет переменная pt, а точнее, на что она ссылается? В действительности, мы ее можем рассматривать как **пространство имен** с именем pt. В этом пространстве есть две переменные x и y, которые берутся непосредственно из класса Point. И мы в этом можем легко убедиться, если изменить переменную x в самом классе:

Point.x = 100

то это приведет к изменению значения и в пространстве имен pt:

print( pt.x, pt.y )

Но при этом, pt и Point – это совершенно разные объекты, их id:

print( id(pt), id(Point), sep="\n" )

различаются. То есть, операция «pt = Point()» действительно создает новый объект (по существу, новое пространство имен) с набором данных, которые были прописаны в классе Point. И если мы будем создавать другие экземпляры этого же класса:

pt2 = Point()

и так далее, то все они будут ссылаться на независимые объекты, но переменные x, y в них, будут также браться из класса Point. Для тех кто знает ООП на С++ или Java, это похоже на поведение статических переменных, объявленных внутри класса. Кстати, в Python статические переменные именно так и объявляют, но о них мы будем говорить отдельно позже.

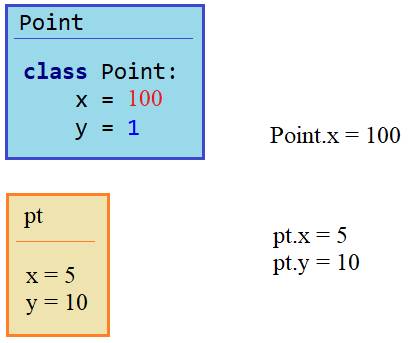
Далее, смотрите, если после создания экземпляра написать вот такие строчки:

pt.x = 5

pt.y = 10

то в пространстве имен pt будут созданы свои переменные x и y, ссылающиеся на свои числовые значения, никак не связанные с классом Point:

print( pt.x, Point.x )



И в каждом экземпляре имеется ссылка на словарь \_\_dict\_\_, который содержит список всех его локальных переменных

print( pt.\_\_dict\_\_ )

pt.x = 5

pt.y = 10

print( pt.\_\_dict\_\_ )

Из этого примера хорошо видно, что до определения x, y внутри пространства pt словарь был пуст, а после, появились две переменные. В терминологии ООП языка Python переменные x, y внутри класса Point или его экземпляров называются **атрибутами** (или **свойствами**). Именно так я в дальнейшем их буду называть.

С атрибутами экземпляров класса можно работать через такие функции:

* getattr(obj, name [, default]) — возвращает значение атрибута объекта;
* hasattr(obj, name) — проверяет на наличие атрибута name в obj;
* setattr(obj, name, value) — задает значение атрибута (если атрибут не существует, то он создается);
* delattr(obj, name) — удаляет атрибут с именем name.

Использовать их можно так:

print( getattr(pt, "x") )

print( getattr(pt, "z") )

print( getattr(pt, "z", False) )

print( hasattr(pt, "z") )

setattr(pt, "z", 7)

print( pt.\_\_dict\_\_ )

delattr(pt, "z")

print( pt.\_\_dict\_\_ )

Эти же операции можно делать и с самим классом:

setattr(Point, "z", 7)

Спрашивается, зачем нужны эти функции, если с атрибутами мы можем работать непосредственно через имена объектов:

Point.z = 100

pt.msg = "hello"

и так далее. Дело в том, что в ряде случаев они дают большую гибкость. Например, при обращении к несуществующему атрибуту:

pt.sss

получим ошибку, но через функцию:

res = getattr(pt, "sss", False)

ошибки не будет и переменная res будет равна False. Кроме того, удалить атрибут:

delattr(pt, "x")

или проверить его существование:

print( hasattr(Point, "t") )

можно только через эти функции. Правда, удалить атрибут можно также с помощью оператора del:

del pt.x

print( pt.\_\_dict\_\_ )

В заключение этого занятия рассмотрим весьма полезную функцию isinstance(), которая позволяет определить принадлежность экземпляра к тому или иному классу. Например, вот такая запись:

print(isinstance(pt, Point))

отобразит в консоли значение True, так как pt – экземпляр класса Point. Если для примера, объявить еще один класс:

class Point3D:

    pass

и выполнить проверку:

print(isinstance(pt, Point3D))

то получим значение False, т.к. pt – не экземпляр класса Point3D.

Итак, на этом первом занятии мы с вами разобрали принципы, лежащие в основе ООП:

* инкапсуляция;
* наследование;
* полиморфизм.

Понятие класса, экземпляра класса и как они создаются. Понятие атрибутов, затронули некоторые встроенные переменные:

* \_\_doc\_\_ –  содержит строку с описанием класса;
* \_\_name\_\_ –  содержит строку с именем класса;
* \_\_dict\_\_ –  содержит набор атрибутов экземпляра класса.

Набор функций для работы с атрибутами и функцию isinstance.

# Методы класса, конструктор и деструктор

[Смотреть материал на видео](https://www.youtube.com/watch?v=7kk2gRf8Uws&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/rezhimy-dostupa-gettery-i-settery>

На предыдущем занятии мы с вами разобрались с понятием класса в Python, его атрибутами и посмотрели как создаются экземпляры класса. Теперь ответим на вопрос: что же может содержать класс? По сути, только две вещи: атрибуты (свойства) и методы. Атрибуты – это переменные класса, а методы – это функции, реализующие определенную логику (то есть, функциональность) класса. Далее, мы будем использовать термин **метод**, подразумевая под ним функцию класса. Имена методов, обычно, являются глаголами, т.к. выполняют определенные действия. В то время, как имена атрибутов – это существительные, т.к. лишь хранят определенные данные.

Давайте, для примера объявим метод setCoords в классе Point:

class Point:

    x = 1; y = 1

    def setCoords(self):

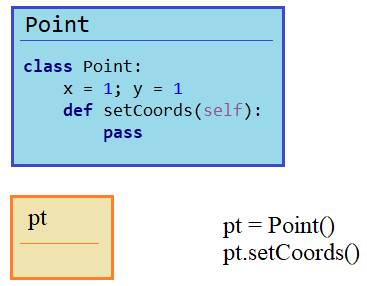
        pass

Здесь сразу бросается в глаза вот этот параметр self. Зачем он здесь, если мы пока ничего не собираемся передавать этому методу? Так требуется делать по синтаксису самого Питона: когда мы объявляем какой-либо метод внутри класса, то обязательно должны указывать такой первый параметр. Теперь нужно разобраться: что он означает? При вызове этого метода, например, через экземпляр pt:

pt = Point()

pt.setCoords()

параметр self будет ссылаться на этот объект-экземпляр, причем, Python сам автоматически подставит нужное значение. Поэтому в скобках при вызове метода ничего писать не нужно.



Давайте в этом убедимся и выведем список локальных атрибутов экземпляра, на который ссылается self:

    def setCoords(self):

        print( self.\_\_dict\_\_ )

И при запуске видим пустой список. Пропишем координаты в экземпляре pt:

pt.x = 5

pt.y = 10

Теперь, при запуске видим две записи для x и y. То есть, self действительно ссылается на тот же объект, что и pt.

Раз это так, то мы с помощью метода setCoords можем переопределять координаты нашей точки:

    def setCoords(self, x, y):

        self.x = x

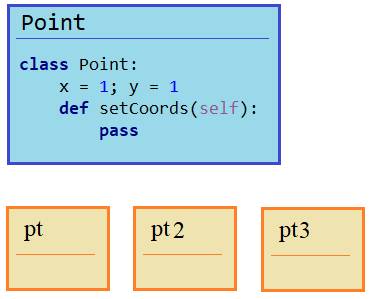
        self.y = y

И, далее, вызвать этот метод:

pt.setCoords(5, 10)

print( pt.\_\_dict\_\_ )

Смотрите, мы здесь автоматически (через self) добавляем локальные атрибуты для экземпляра pt. То есть, указанные координаты будут принадлежать только объекту pt и никак не повлияют на значения других экземпляров. Что, в общем то, мы и ожидаем от использования объекта pt.



И здесь, опять же обратите внимание, что метод setCoords – общий для всех возможных экземпляров класса Point. Когда мы с вами на предыдущем занятии рассматривали переменные внутри класса, то об этом подробно говорили. Поведение методов абсолютно такое же: объявленные внутри класса они становятся общими для всех экземпляров, т.е. это будет одна и та же функция, а не копии внутри экземпляров. И только благодаря первому параметру self мы можем «знать» из какого конкретно экземпляра был вызван данный метод: self всегда ссылается на этот экземпляр. А вот, если вызвать этот метод непосредственно из класса:

Point.setCoords(5, 10)

то возникнет ошибка, т.к. в этом случае нужно явно указать первый аргумент:

Point.setCoords(pt, 5, 10)

## Конструктор класса

Теперь предположим, что мы хотели бы в момент создания экземпляра класса сразу указывать нужные координаты и прописывать их как локальные атрибуты. Это можно сделать с помощью специального метода:

\_\_init\_\_()

который вызывается при создании объекта:

    def \_\_init\_\_(self):

        print("создание экземпляра класса Point")

И, при выполнении:

pt = Point()

будет вызван данный метод. В ООП он имеет специальное название – **конструктор**. Итак, укажем в этом конструкторе еще два аргумента и создадим в нем локальные атрибуты:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

Теперь создать объект можно, только явно указав два аргумента:

pt = Point(5, 10)

print( pt.\_\_dict\_\_ )

и, как видите, у нас получается полноценный экземпляр класса с двумя локальными координатами, связанными только с этим объектом.

Для большей гибкости, укажем в конструкторе аргументы x, y по умолчанию:

def \_\_init\_\_(self, x = 0, y = 0):

и тогда можно вызывать конструктор следующими способами:

pt = Point()

pt2 = Point(5)

pt3 = Point(5, 10)

print( pt.\_\_dict\_\_, pt2.\_\_dict\_\_, pt3.\_\_dict\_\_, sep="**\n**" )

## Деструктор класса

Противоположный конструктору метод:

\_\_del\_\_()

автоматически вызывается при уничтожении экземпляра класса. Он называется **деструктор**.

Давайте запишем вот такой деструктор в наш класс Point:

    def \_\_del\_\_(self):

        print("Удаление экземпляра: "+self.\_\_str\_\_())

и, запуская программу, мы видим, что все наши три объекта были уничтожены при завершении программы.

Но, в какой момент вообще происходит удаление объекта в Python? Алгоритм прост: пока на какой-либо объект имеется хотя бы одна внешняя ссылка, он продолжает существовать. Если же на объект нет внешних ссылок, он автоматически уничтожается «**сборщиком мусора**». Например, если создать экземпляр:

pt = Point()

то  на него будет ссылаться переменная pt. Но, при ее переопределении, например, так:

pt = 0

экземпляр класса Point в памяти остается без внешних ссылок и вскоре будет удален сборщиком мусора. При удалении, как раз и будет вызван деструктор класса \_\_del\_\_().

# Режимы доступа, геттеры и сеттеры

[Смотреть материал на видео](https://www.youtube.com/watch?v=WP2sqI2BkeY&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/rezhimy-dostupa-gettery-i-settery>

Теперь, когда мы научились создавать экземпляры классов и объявлять в них атрибуты и методы пришла пора рассмотреть важную тему ограничения доступа к данным и методам класса извне. Это основа механизма **инкапсуляции**. Также здесь посмотрим как происходит обращение к закрытым (приватным) свойствам через специальные методы: **геттеры** и **сеттеры**. И, затем, уже на следующих занятиях разовьем этот подход и поговорим о создании **объектов-свойств** (property) класса и **дескрипторах**. Фактически, это одна большая и единая тема. Для простоты восприятия я разобью ее на несколько занятий. Здесь мы разберемся с основами работы с закрытыми переменными класса, а следующее занятие будет естественным продолжением этой темы.

Итак, начнем с реализации принципа **инкапсуляции** в Python. Давайте вернемся к нашему классу представления точки на плоскости:

class Point:

    def \_\_init\_\_(self, x = 0, y = 0):

        self.x = x; self.y = y

Смотрите, когда мы создаем экземпляр этого класса:

pt = Point(1, 2)

то мы имеем полный доступ ко всем его локальным переменным:

print( pt.x, pt.y )

а, значит, можем изменить их вне этого класса:

pt.x = 100

pt.y = "abc"

и присвоить любые значения, в том числе и недопустимые (строку). Чтобы программист не мог произвольным образом задавать атрибуты их следует «закрывать» от вмешательства извне. В Python возможны следующие варианты доступа к данным:

* <имя атрибута> (без одного или двух подчеркиваний вначале) – публичное свойство (**public**);
* \_<имя атрибута> (с одним подчеркиванием) – режим доступа **protected** (можно обращаться только внутри класса и во всех его дочерних классах)
* \_\_<имя атрибута> (с двумя подчеркиваниями) – режим доступа **private** (можно обращаться только внутри класса).

Как видите, ничего сложного нет. Пока мы будем использовать или публичные свойства (public) или закрытые (private). О режиме protected поговорим, когда будем рассматривать наследование классов.

Итак, создадим в конструкторе два приватных атрибута:

self.\_\_x = x; self.\_\_y = y

и теперь обратиться напрямую к этим переменным уже не получится, они стали закрытыми (приватными):

print( pt.\_\_x )

менять и использовать их можно только внутри экземпляра класса.

Но как мы тогда сможем переопределять такие свойства? Для этого в классах объявляют специальные методы, которые, обычно, начинаются с префикса set (что означает задать, установить) и далее, какое-либо имя, например, так:

def setCoords(self, x, y):

    self.\_\_x = x; self.\_\_y = y

Теперь, мы можем вызвать этот метод и указать нужные координаты для приватных атрибутов:

pt = Point()

pt.setCoords(10, 20)

Чтобы увидеть изменение данных, давайте добавим еще один метод, у которого будет префикс get (получить):

def getCoords(self):

    return self.\_\_x, self.\_\_y

И, далее, вызовем его:

print( pt.getCoords() )

Такие методы в ООП называются **сеттерами**и **геттерами**. Их назначение не только передавать значения между приватными атрибутами класса, но и проверять их корректность. Например, в нашем случае координаты должны быть числами. Поэтому, прежде чем обновлять значения переменных, следует проверить их тип данных. Для этого можно воспользоваться функцией isinstance и записать сеттер следующим образом:

def setCoords(self, x, y):

    if (isinstance(x, int) or isinstance(x, float)) and \

        (isinstance(y, int) or isinstance(y, float)) :

        self.\_\_x = x

        self.\_\_y = y    else:

        print("Координаты должны быть числами")

Обратите внимание вот на этот слеш. Он используется когда конструкцию в языке Python нужно записать в несколько строчек. Он, говорит интерпретатору языка, что следующая строчка – это продолжение первой. Итак, здесь мы проверяем, что обе переданные переменные x и y должны иметь тип int или float и только после этого приватным атрибутам экземпляра класса будут присвоены эти значения. Иначе, выводится сообщение, что координаты должны быть числами. Давайте проверим как это будет работать:

pt.setCoords(10, 20)

print( pt.getCoords() )

pt.setCoords("10", 20)

print( pt.getCoords() )

Как видите, все работает именно так, как мы и хотели.

Продолжим совершенствовать наш класс Point. Вот такие проверки корректности данных, часто реализуют в виде вспомогательных методов, которые доступны только внутри класса, то есть, их делают приватными (закрытыми). Приватный метод объявляется также как и приватная переменная – двумя подчеркиваниями, например, так:

def \_\_checkValue(x):

    if isinstance(x, int) or isinstance(x, float):

        return True

   return False

Мы здесь объявили метод без параметра self, т.к. ему не нужен доступ к экземпляру класса и вызываться он будет непосредственно через класс:

if Point.\_\_checkValue(x) and Point.\_\_checkValue(y) :

Запускаем программу и видим, что все работает также. Но, при этом, доступа к этому методу извне нет, он приватный.

Конечно, при большом желании мы можем обратиться и к таким свойствам и методам класса извне. В Python они шифруются по шаблону:

\_Имя класса\_\_имя переменной  
\_Имя класса\_\_имя метода

Например, мы можем изменить приватную переменную \_\_x, так:

pt.\_Point\_\_x = 100

и теперь она равна 100. Или, вызвать скрытый метод:

Point.\_Point\_\_checkValue(4)

Однако, без крайней необходимости так делать не нужно и двойное подчеркивание должно сигнализировать программисту, что работать с таким атрибутом нужно только через сеттеры и геттеры. Иначе, возможны непредвиденные ошибки.

При необходимости мы можем осуществлять дополнительный контроль изменения атрибутов, путем перегрузки следующих методов:

* \_\_setattr(self, key, value)\_\_ – автоматически вызывается при изменении свойства key класса;
* \_\_getattribute\_\_(self, item) – автоматически вызывается при получении свойства класса с именем item;
* \_\_getattr\_\_(self, item) – автоматически вызывается при получении несуществующего свойства item класса;
* \_\_delattr\_\_(self, item) – автоматически вызывается при удалении свойства item (не важно: существует оно или нет).

Например, переопределив метод:

def \_\_getattribute\_\_(self, item):

    if item == "\_Point\_\_x":

        raise ValueError("Private attribute")

    else:

        return object.\_\_getattribute\_\_(self, item)

мы не сможем извне узнать значение приватной переменной \_\_x:

print( pt.\_Point\_\_x )

выдаст исключение.  Обратите внимание, что для всех остальных свойств мы должны вызывать этот метод базового класса, в данном случае – это класс object. Иначе, мы не сможем работать с атрибутами класса. Что такое базовый класс мы еще будем говорить в теме «Наследование». Пока просто запомните этот момент.

Далее, переопределяя метод:

def \_\_setattr\_\_(self, key, value):

    if key == "WIDTH":

        raise AttributeError

    else:

        self.\_\_dict\_\_[key] = value

мы запрещаем изменять свойство WIDTH нашего класса. При попытке это сделать:

pt.WIDTH = 5

сработает исключение AttributeError. А иначе, мы в экземпляре класса меняем значение свойства с именем key на значение value. Обратите внимание, что внутри \_\_setattr\_\_ менять свойства следует только через словарь \_\_dict\_\_. Если записать здесь что-то вроде:

self.\_\_x = value

то у нас метод \_\_setattr\_\_ начнет рекуррентно вызываться, пока не переполнится стек вызова функций.

Наконец, методы \_\_delattr\_\_ и \_\_getattr\_\_ можно перегрузить так:

def \_\_getattr\_\_(self, item):

    print("\_\_getattr\_\_: "+item)

def \_\_delattr\_\_(self, item):

    print("\_\_delattr\_\_: "+item)

Еще один дополнительный контроль за локальными свойствами экземпляров классов можно сделать с помощью коллекции:

\_\_slots\_\_ = ["\_\_x", "\_\_y"]

и при попытке создать какие-либо дополнительные локальные атрибуты:

pt.z = 1

возникнет ошибка. И, обратите внимание, в slots можно указывать имена только локальных свойств экземпляров, записать туда имена переменных самого класса нельзя. Вот такая запись приведет к ошибке:

\_\_slots\_\_ = ["\_\_x", "\_\_y", "WIDTH"]

На этом мы завершим первую часть темы работы с атрибутами класса. На следующем занятии продолжим и поговорим о создании объектов-свойств (property) и довольно интересном и гибком механизме – дескрипторах.

# Создание объектов-свойств

[Смотреть материал на видео](https://www.youtube.com/watch?v=zwel95I7O88&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/sozdanie-obektov-svoystv>

Это занятие является продолжением предыдущего, где мы рассматривали приватные атрибуты, сеттеры и геттеры, а также контроль за их изменением при помощи перегрузки некоторых базовых методов. Однако, пользоваться на практике напрямую геттерами и сеттерами бывает не всегда удобно. Большего изящества кода можно добиться, используя так называемые объекты-свойства (property). Например, мы хотим создать в классе Point свойство

coordX

которому можно было бы присваивать и считывать значения подобно обычной переменной:

pt.coordX = 100 *# запись значения*

x = pt.coordX   *# чтение значения*

но с дополнительной проверкой корректности записи и считывания данных. Для этого создадим внутри класса геттер и сеттер, на основе которых будет работать это свойство. Пусть они будут очень простыми:

def \_\_getCoordX(self):

    print("вызов \_\_getCoordX")

    return self.\_\_x

def \_\_setCoordX(self, x):

    print("вызов \_\_setCoordX")

    self.\_\_x = x

То есть, при вызове геттера мы возвращаем значение приватного свойства\_\_x, а при вызове сеттера – заносим новое значение в этот атрибут. На основе этих приватных методов создаем свойство через специальный класс property:

coordX = property(\_\_getCoordX, \_\_setCoordX)

Все, наше свойство готово и далее можем работать с ним, используя синтаксис обычной переменной:

pt = Point()

print( pt.coordX )

pt.coordX = 100

print( pt.coordX )

Видите, это намного удобнее, чем вызывать отдельно сеттеры и геттеры.

Теперь уберем из геттера и сеттера функцию print и добавим проверку на корректность передаваемых данных:

def \_\_getCoordX(self):

    return self.\_\_x

def \_\_setCoordX(self, x):

    if Point.\_\_checkValue(x):

        self.\_\_x = x

    else:

        raise ValueError("Неверный формат данных")

Теперь, если попытаться передать не числовое значение:

pt.coordX = "100"

то возникнет исключение ValueError.

У свойства может быть еще один метод, вызываемый при его удалении:

def \_\_delCoordX(self):

    print("Удаление свойства")

    del self.\_\_x

Он указывается третьим параметром при вызове класса property, а четвертым можно указать описание свойства:

coordX = property(\_\_getCoordX, \_\_setCoordX, \_\_delCoordX, "Работа с X")

Если теперь выполнить удаление свойства:

del pt.coordX

то увидим сообщение «Удаление свойства» и дальнейшая попытка к его обращению:

pt.coordX

приведет к ошибке, т.к. приватного свойства \_\_x уже не существует. Вот так работают и создаются свойства в Python. И давайте здесь же я вам покажу еще один способ объявления свойств через декораторы (если вы не знаете что такое декоратор, то смотрите урок по этой теме – ссылка под этим видео).

Перед геттером мы укажем декоратор

@property

И название метода должно совпадать с названием свойства:

def coordX(self):

Далее, у сеттера указываем декоратор с то же имя метода:

@coordX.setter

def coordX(self, x):

То есть, пишем имя нашего свойства и через точку зарезервированное имя setter. Ну а перед делитером декоратор

@coordX.deleter

def coordX(self):

Все, теперь мы абсолютно также можем работать со свойством coordX:

pt = Point()

print( pt.coordX )

pt.coordX = 100

print( pt.coordX )

del pt.coordX

Хорошо, свойство coordX у нас есть. Но нам нужно создать еще одно – coordY. Как вы понимаете, это, фактически, приведет к дублированию кода для coordX, что не есть хорошо, так как нарушает принцип программирования

DRY (Don’t Repeat Yourself) – не повторяйся!

И здесь нам на помощь приходит другой механизм Python – **дескрипторы**, о котором речь пойдет на следующем занятии.

# Создание дескрипторов классов

[Смотреть материал на видео](https://www.youtube.com/watch?v=zwel95I7O88&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/sozdanie-deskriptorov-klassov>

На предыдущем занятии мы с вами рассмотрели создание объектов-свойств и видели, что для определения нескольких однотипных таких объектов приходится дублировать программный код, что не очень хорошо. Чтобы этого избежать можно воспользоваться механизмом дескрипторов и сейчас вы узнаете что это такое.

В самом простом случае **дескриптор** – это класс, в котором определены следующие специальные методы:

class CoordValue:

    def \_\_get\_\_(self, instance, owner):

        return self.\_\_value

    def \_\_set\_\_(self, instance, value):

        self.\_\_value = value

    def \_\_delete\_\_(self, obj):

        del self.\_\_value

Как вы уже догадались, get – это геттер, set – сеттер, а delete – вызывается при удалении свойства дескриптора. И далее, используя этот класс, создадим два дескриптора в классе Point:

class Point:

    coordX = CoordValue()

    coordY = CoordValue()

    def \_\_init\_\_(self, x = 0, y = 0):

        self.coordX = x

        self.coordY = y

Сразу скажу, что это не совсем верная реализация дескрипторов, но я намеренно допустил ошибку, чтобы вы лучше поняли материал. Если визуально представить эти классы и экземпляр объекта:

pt = Point(1, 2)

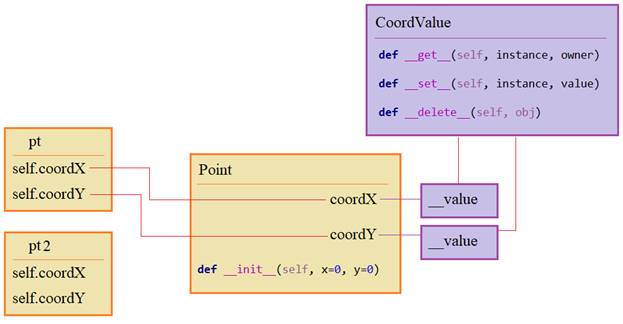
то это будет выглядеть так. Смотрите, здесь свойства экземпляра coordX и coordY берутся непосредственно из класса Point. Мы в этом можем убедиться, если выведем их id:

print( id(pt.coordX), id(Point.coordX) )

И даже операция в конструкторе класса:

self.coordX = x

не создает новое локальное свойство, так как в дескрипторах оператор присваивания перегружен и идет просто вызов метода \_\_set\_\_ класса CoordValue. При этом вызове параметр self ссылается на объект coordX, в котором приватному значению \_\_value присваивается значение value. То же самое и при обращении к coordY. Только здесь self будет ссылаться уже на второй объект и информация записывается в другую переменную \_\_value.



И если мы будем работать с дескрипторами подобно свойствам объекта, то вроде бы у нас все работает как надо:

pt = Point(1, 2)

print( pt.coordX, pt.coordY )

pt.coordX = 100

print( pt.coordX, pt.coordY )

Но давайте теперь создадим еще один экземпляр:

pt = Point(1, 2)

pt2 = Point(10, 20)

Смотрите, при выполнении мы видим, что свойства обоих объектов оказываются идентичными:

print( pt.coordX, pt.coordY )

print( pt2.coordX, pt2.coordY )

И вы уже догадались почему: coordX, coordY второго экземпляра также берутся напрямую из класса Point, то есть, это все те же самые дескрипторы.

Давайте теперь поправим программу и сделаем так, чтобы координаты точки хранились непосредственно в экземплярах классов. Перепишем класс дескриптора вот в таком виде:

class CoordValue:

    def \_\_init\_\_(self, name):

        self.\_\_name = name

    def \_\_get\_\_(self, instance, owner):

        return instance.\_\_dict\_\_[self.\_\_name]

    def \_\_set\_\_(self, instance, value):

        instance.\_\_dict\_\_[self.\_\_name] = value

Здесь параметр instance ссылается на экземпляр класса Point, для которого был вызван метод. И через него, используя коллекцию \_\_dict\_\_ мы делаем обработку локальных свойств: возвращаем их или устанавливаем. А, чтобы знать имя локального свойства, здесь был добавлен конструктор с параметром name – имя свойства. И мы его сохраняем в экземпляре объекта-дескриптора в приватной переменной \_\_name.

Далее, в самом классе Point сделаем такие вызовы:

coordX = CoordValue("coordX")

coordY = CoordValue("coordY")

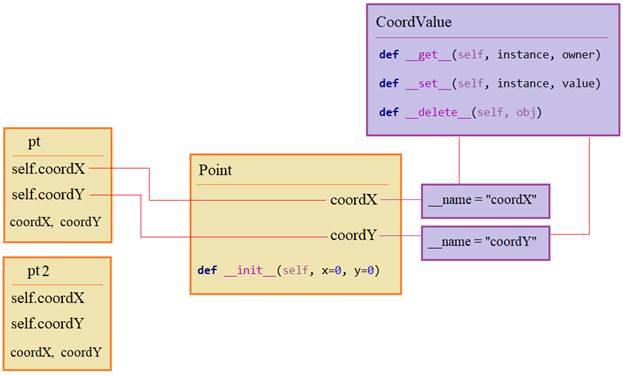
И, теперь, при запуске программы, мы видим разные значения координат для разных экземпляров классов:

pt = Point(1, 2)

pt2 = Point(10, 20)

print( pt.coordX, pt.coordY )

print( pt2.coordX, pt2.coordY )



Но эту программу можно еще упростить и улучшить. Начиная с версии Python 3.6 и выше, у дескрипторов появился новый весьма полезный метод:

def \_\_set\_name\_\_(self, owner, name):

    print(name)

    self.\_\_name = name

Он вызывается автоматически при создании экземпляра дескриптора и в параметре name хранится имя экземпляра класса. Мы вполне можем его использовать для создания соответствующих локальных атрибутов в экземплярах классов Point. И конструктор нам больше не нужно переопределять, уберем его. В самом классе Point создадим дескрипторы без указания имен:

coordX = CoordValue()

coordY = CoordValue()

и, запуская программу, видим, имена coordX и coordY и полную работоспособность наших дескрипторов.

И у вас здесь может возникнуть вопрос: у нас в экземплярах класса есть локальные атрибуты coordX, coordY и доступ к дескрипторам класса Point с теми же именами? Как Python «понимает», что нужно обращаться именно к дескрипторам, а не к локальным свойствам объекта pt? Все дело в приоритетах. Если в самом классе (или базовом классе) имеется дескриптор с тем же именем, что и свойство в коллекции \_\_dict\_\_, то приоритет отдается дескриптору и именно он берется для записи или считывания значений. Поэтому наша программа корректно работала.

Но здесь тоже есть одна тонкость: дескрипторы в Python бывают двух видов: для данных и для не данных (non-data descriptor). Последний отличается тем, что реализует только один метод \_\_get\_\_:

class NoDataDescr:

    def \_\_set\_name\_\_(self, owner, name):

        self.\_\_name = name

    def \_\_get\_\_(self, instance, owner):

        return "NoDataDescr \_\_get\_\_"

И, если его создать в классе Point:

noData = NoDataDescr()

то при запуске мы увидим, строчку из этого «дескриптора не данных». Но, если выполним присвоение свойству с тем же именем:

pt.noData = "hello"

то наш дескриптор noData пропадет из локального окружения экземпляра pt. Что, в общем то не удивительно, т.к. в этом дескрипторе нет метода set для изменения соответствующего локального свойства.

Но, чаще всего на практике используют дескрипторы для данных, то есть, и для записи и для считывания значений. Поэтому этот последний момент просто нужно знать, чтобы избежать лишних ошибок.

И в заключение отмечу такую вещь. Дескрипторы – это безусловно классный, красивый и гибкий механизм, позволяющий создавать красивый программный код. Однако, прежде чем использовать какой-либо инструмент, задайте себе вопрос: действительно ли он здесь нужен, не проще ли все реализовать двумя методами: геттером и сеттером и, как говорится, не городить огород? Неоправданное усложнение программы это также плохо как и отказ от существующих инструментов языка программирования. При проектировании программ следует придерживаться грани, когда с одной стороны простые способы реализации, а с другой – возможные проблемы при масштабировании проекта. Но, это все приходит с опытом и, так как дорогу осилит только идущий, то вперед на покорение дескрипторов и свойств классов. Для начала попробуйте реализовать выполнить следующие задания на Python.

# Статические свойства и методы классов

[Смотреть материал на видео](https://www.youtube.com/watch?v=77kW3F9ZBHI&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/staticheskie-svoystva-i-metody-klassov>

На предыдущих занятиях мы с вами часто отмечали такой факт, что публичное свойство, объявленное внутри какого-либо класса напрямую доступно из его экземпляров. С нашим уже традиционным классом Point это выглядело так:

class Point:

    count = 0

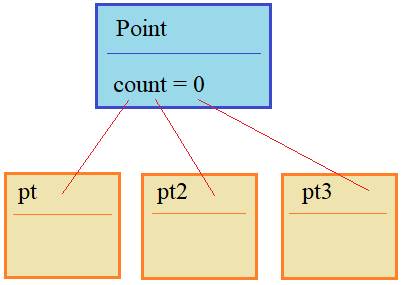
И, создавая экземпляры этого класса:

pt = Point()

pt2 = Point()

pt3 = Point()

все они непосредственно обращаются к этому свойству:



В этом легко убедиться, изменив значение count непосредственно в классе Point:

Point.count = 10

все три экземпляра выведут в консоль именно это значение:

print( pt.count, pt2.count, pt3.count )

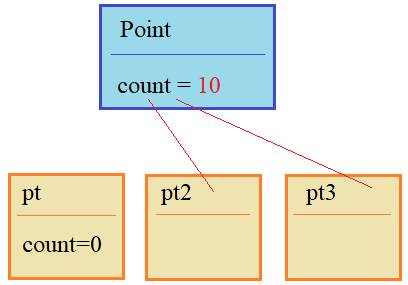
Свойства классов с таким поведением подобны **статическим переменным**. В противовес нестатическим, которые создаются как локальные непосредственно в экземплярах классов. Например, если записать вот такую строчку:

pt.count = 0

то это не значит, что мы меняем статическую переменную в классе Point, мы создаем здесь локальное свойство в экземпляре класса pt. И при выводе значений в консоль:

print( pt.count, pt2.count, pt3.count )

увидим, что первое значение 0, а остальные по 10.



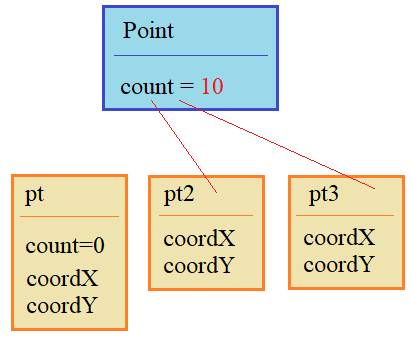
То же самое происходит и при вызове конструктора:

def \_\_init\_\_(self, x = 0, y = 0):

    self.coordX = x

    self.coordY = y

Здесь в каждом экземпляре класса создаются локальные свойства coordX и coordY:



Помимо так называемых статических свойств в Python можно создавать статические методы. По идее, любой метод, объявленный в классе, ведет себя как статический. Давайте сделаем наш счетчик приватным и в конструкторе будем его увеличивать на 1, т.е. будем подсчитывать число экземпляров класса Point:

class Point:

    \_\_count = 0

    def \_\_init\_\_(self, x = 0, y = 0):

        Point.\_\_count += 1

        self.coordX = x

        self.coordY = y

И, далее, пропишем геттер для получения этого значения:

def getCount(self):

    return Point.\_\_count

Все это можно использовать так:

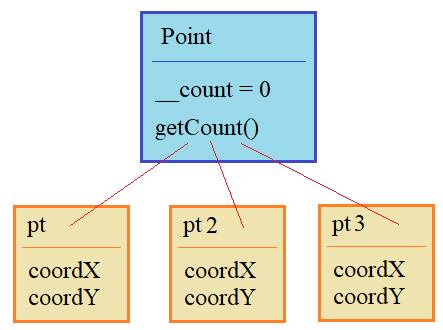
pt = Point()

pt2 = Point()

pt3 = Point()

print( pt.getCount() )

Увидим значение 3. Причем, обращение к методу также происходит напрямую через класс Point:



И, так как он возвращает значение статического свойства, объявленного внутри Point, то параметр self у этого метода нам не нужен. Мало того, из-за этого параметры мы не можем вызвать этот метод через Point вот так:

Point.getCount()

здесь обязательно нужно передать первый параметр:

Point.getCount(pt)

или, даже так:

Point.getCount(1)

потому что, нам абсолютно все равно, чему равен self, мы с ним не работаем. Давайте попробуем убрать его из метода:

def getCount():

Смотрите, среда PyCharm нам подсказывает, что так делать нельзя, параметр необходим. Но, мы то знаем, что будем использовать этот метод как статический и это явно нужно указать интерпретатору Python. Для этого воспользуемся специальным декоратором функций:

@staticmethod

def getCount():

    return Point.\_\_count

и теперь можем обращаться к этому методу без указания каких-либо параметров:

print( pt.getCount(), Point.getCount() )

причем, это также работает и с экземплярами классов. То есть, мы создали чисто статический метод, который вызывается непосредственно из класса Point  и может работать только со свойствами и методами этого класса.

Конечно, мы всегда в экземплярах класса можем перегрузить метод getCount, записать, например, так:

def getCount():

    return 5

pt.getCount = getCount

Тогда в pt будет создано новое свойство getCount, которое будет ссылаться на объект-функцию и, соответственно, ее вызывать:

print( pt.getCount(), Point.getCount() )

Но это уже все на совести программиста. Если он так написал, значит, знает, что делает. Вот что из себя представляют статические методы и свойства.

## Класс-синглетон (singleton)

В заключение приведу один довольно популярный прием в программировании – создание класса-синглетона, то есть, класс, для которого можно создать только один экземпляр. Для этого добавим в наш класс Point приватную переменную

\_\_instance = None

и выполним перегрузку метода

def \_\_new\_\_(cls, \*args, \*\*kwargs):

    if not isinstance(cls.\_\_instance, cls):

        cls.\_\_instance = super().\_\_new\_\_(cls)

    else:

        print("Экземпляр класса Point уже создан!")

    return cls.\_\_instance

который вызывается в момент создания экземпляра класса. Здесь мы проверяем: если статическое свойство \_\_instance не является создаваемым классом, значит, мы еще ни разу не создавали экземпляр. В этом случае обращаемся к базовому классу и через него создаем экземпляр текущего класса Point. Результат (ссылку на него) сохраняем в нашем статическом свойстве \_\_instance и в конце возвращаем его. При повторной попытке создать экземпляр, условие будет ложным, мы увидим строку «Экземпляр класса Point уже создан» и вернем ссылку на ранее созданный объект.

Протестируем эту программу, создадим несколько экземпляров и выведем их id:

pt = Point()

pt2 = Point()

pt3 = Point()

print( id(pt), id(pt2), id(pt3), sep="**\n**" )

Как видите, все три переменные ссылаются на один и тот же объект, чего мы и хотели добиться.

Это одна из самых простых реализаций singleton (синглетона), но имеет ряд недостатков. Один из них, возможность обойти перегрузку метода \_\_new\_\_ через создание дочерних экземпляров классов, которые бы наследовались от класса Point. Но это уже детали. Я вам хотел лишь в целом показать как используются статические свойства классов на практике.

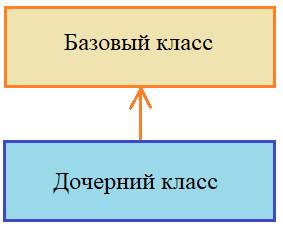
# Простое наследование классов

[Смотреть материал на видео](https://www.youtube.com/watch?v=VQ6vUzqAInU&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/prostoe-nasledovanie-klassov>

На этом занятии мы начнем рассмотрение очень важной темы в ООП – наследование классов. На первом уроке данного курса я приводил пример с кофемолкой и говорил, что для создания электрической кофемолки можно взять уже готовые наработки механической кофемолки, добавить туда мотор, схему и получим новый прибор. Этот пример как раз описывает общий принцип наследования в ООП: мы можем взять некий класс и на его основе создать новый, дочерний, изменив и расширив функционал базового класса.





Причем, в Python 3 любой создаваемый класс, например:

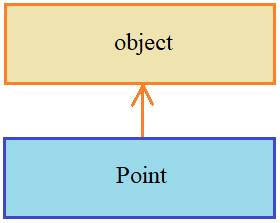
class Point:

    def \_\_init\_\_(self, x = 0, y = 0):

        self.\_\_x = x

        self.\_\_y = y

автоматически является дочерним по отношению к базовому object:



И в этом легко убедиться, вызвав специальную функцию:

print( issubclass(Point, object) )

которая возвращает True, если класс является дочерним для класса, указанного вторым параметром.

Теперь давайте посмотрим как в Python реализуется механизм наследования. Для начала создадим вспомогательный класс Point для хранения координат на плоскости:

class Point:

    def \_\_init\_\_(self, x = 0, y = 0):

        self.x = x

        self.y = y

И после него объявим класс для работы с графическим примитивом линией:

class Line:

    def \_\_init\_\_(self, sp:Point, ep:Point, color:str = "red", width:int = 1):

        self.\_sp = sp

        self.\_ep = ep

        self.\_color = color

        self.\_width = width

    def drawLine(self):

        print(f"Рисование линии: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_width}")

Обратите внимание вот на эту нотацию (:Point, :str, :int). Так можно в тексте программы указывать: какого типа параметры мы здесь ожидаем получать. Конечно, это не значит, что другие типы здесь не могут быть переданы (сюда по прежнему можно передать что угодно), это лишь подсказка программисту с чем мы тут собираемся иметь дело – с объектами класса Point.

Создадим экземпляр этого класса:

l = Line( Point(1,2), Point(10,20) )

l.drawLine()

И в консоли увидим не совсем читаемое сообщение. Дело в том, что при вызове self.\_sp возвращаются не координаты точки, а стандартное сообщение класса Point. Давайте его переопределим, чтобы выводились координаты:

def \_\_str\_\_(self):

    return f"({self.\_\_x}, {self.\_\_y})"

Теперь при запуске видим то, что нам было нужно.

Далее, мы подумали и решили добавить класс еще одного примитива – прямоугольника. И мы пишем следующее:

class Rect:

    def \_\_init\_\_(self, sp:Point, ep:Point, color:str = "red", width:int = 1):

        self.\_sp = sp

        self.\_ep = ep

        self.\_color = color

        self.\_width = width

    def drawRect(self):

        print(f"Рисование прямоугольника: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_width}")

И, затем, вызов:

r = Rect( Point(30,40), Point(70,80) )

r.drawRect()

Смотрите, мы здесь фактически, продублировали текст из класса Line. Отличие только вот этот метод drawRect, остальное абсолютно такое же. Как только что-то подобное встречается в тексте программы, это значит, что программист нарушает общеизвестный принцип DRY (Don’t Repeat Yourself) – не повторяйся! И как раз это можно поправить с помощью механизма наследования.

Вот этот дублирующий текст вынесем в отдельный класс:

class Prop:

    def \_\_init\_\_(self, sp:Point, ep:Point, color:str = "red", width:int = 1):

        self.\_sp = sp

        self.\_ep = ep

        self.\_color = color

        self.\_width = width

а, классы примитивов будем наследовать от него:

class Line(Prop):

    def drawLine(self):

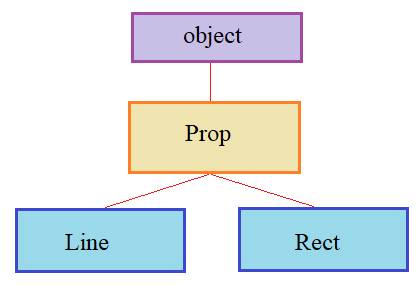
        print(f"Рисование линии: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_width}")

class Rect(Prop):

    def drawRect(self):

        print(f"Рисование прямоугольника: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_width}")

То есть, мы в круглых скобках после имени класса указываем базовый класс. В результате, получаем такую иерархию:

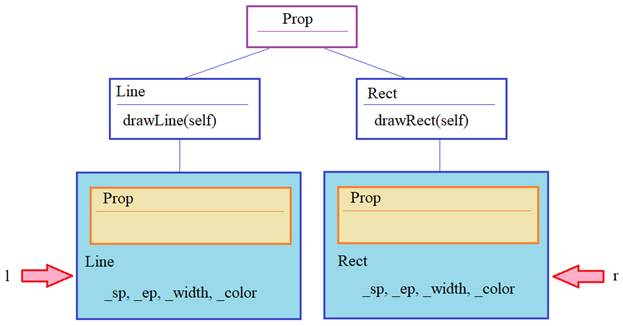


И когда мы создаем объекты этих примитивов:

l = Line( Point(1,2), Point(10,20) )

r = Rect( Point(30,40), Point(70,80) )

то вот этот self ссылается на общий класс Line, поэтому все публичные свойства и методы создаются для дочерних классов Line и Rect.



Вот это нижнее подчеркивание сигнализирует программисту о режиме доступа **protected** к этим атрибутам. То есть, мы оговариваем, что они должны использоваться только внутри объекта Prop и во всех его дочерних классах. И здесь ключевое слово **оговариваем**. Python при этом никак не ограничивает область их использования. Фактически, это такие же публичные атрибуты, только с одним нижним подчеркиванием. Если выполнить вот такую операцию:

print( l.\_width )

то это свойство будем выведено в консоль без каких-либо ошибок. Но тогда зачем нам писать это нижнее подчеркивание, если оно не играет никакой роли. Одна роль у этого префикса все-таки есть: нижнее подчеркивание должно предостерегать программиста от использования этого свойства вне класса. Впоследствии это может стать причиной непредвиденных ошибок. Например, изменится версия модуля и такое свойство может перестать существовать, т.к. никто не предполагал доступа к нему извне. А вот сеттеры и геттеры будут по-прежнему работать и давать тот же результат. Так что, к таким атрибутам лучше не обращаться напрямую – это внутренние, служебные переменные.

Далее, создавая экземпляры дочерних классов, мы использовали конструктор базового класса Prop. В действительности, сначала вызывается конструктор в дочерних классах, но так как мы его явно там не прописывали, то использовался конструктор по умолчанию, который вызывал конструктор базового класса.

Давайте переопределим конструктор в дочернем классе Line и убедимся в этом:

def \_\_init\_\_(self, \*args):

    print("Переопределенный конструктор Line")

Если теперь запустить программу, то мы увидим сообщение «Переопределенный конструктор» и, затем ошибку, связанную с отсутствием выводимых свойств, так как конструктор базового класса не был вызван и локальные свойства не были созданы.

Конструктор суперкласса можно вызвать, просто обратившись к нему:

Prop.\_\_init\_\_(self, \*args)

И в конструкторе Prop добавим:

print("Конструктор базового класса Prop")

Но делать так – порочная практика. В случае множественного наследования – это источник потенциальных ошибок. Почему? Мы об этом позже еще поговорим. А сейчас отметим, что вместо явного указания имени базового класса, следует вызвать специальную функцию super, которая в правильном порядке будет перебирать вышестоящие классы:

super().\_\_init\_\_(\*args)

О ней мы подробнее еще поговорим. Теперь, наша программа вновь обрела работоспособность и мы видим порядок вызовов конструкторов: от дочернего к базовому классу.

Вот так в самом простом варианте работает механизм наследования. Давайте, теперь немного усложним программу и посмотрим как наследуются приватные (частные) свойства. Укажем в суперклассе Prop свойство \_width как приватное:

self.\_\_width = width

добавим в него метод

def getWidth(self):

    return self.\_\_width

При запуске видим ожидаемую ошибку, т.к. частное свойство \_\_width базового класса Prop стало недоступно в дочернем классе Line. Но почему? Ведь self в базовом классе Prop ссылается на Line. Так и есть, но создавая приватную переменную в каком-либо определенном классе, она создается с префиксом этого класса и среди локальных свойств Line мы увидим свойство

\_Prop\_\_width

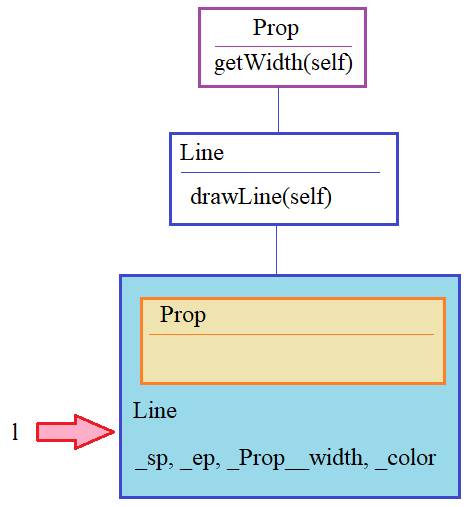
Выведем их в косоль:

print(l.\_\_dict\_\_)

и вот в списке локальных атрибутов объекта Line появилось \_Prop\_\_width со значением 1. Причем, обратиться к нему мы можем только из метода getWidth класса Prop:

def drawLine(self):

    print(f"Рисование линии: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.getWidth()}")



Вызывая метод getWidth базового класса, мы сначала ищем его в классе Line, не находим и переходим в базовый класс Prop. Там он есть, вызываем его и, обращаясь к частному свойству уже из этого класса, к нему добавляется префикс \_Prop и мы успешно читаем этот локальный атрибут из экземпляра класса Line.

Если в конструкторе класса Line после вызова конструктора базового класса создать такое же частное свойство:

self.\_\_width = 5

то в экземпляре класса Line появится запись:

\_Line\_\_width

к которому мы можем обратиться напрямую из этого класса:

def drawLine(self):

    print(f"Рисование линии: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_\_width}")

Вот так создаются приватные атрибуты классов. Ко всем остальным можно напрямую обращаться из любых экземпляров классов.

## Задания для самоподготовки

1. Создайте суперкласс «Персональные компьютеры» и на его основе подклассы: «Настольные ПК» и «Ноутбуки». В базовом классе определите общие свойства: размер памяти, диска, модель, CPU. А в производных классах уникальные свойства:

* для настольных: монитор, клавиатура, мышь, их габариты; и метод для вывода этой информации в консоль;
* для ноутбуков: габариты, диагональ экрана; и метод для вывода этой информации в консоль.

2. Повторите это задания для суперкласса «Человек» и подклассов «Мужчина» и «Женщина». Подумайте, какие общие характеристики можно выделить в суперкласс и какие частные свойства указать в подклассах.

# Полиморфизм - что это такое?

[Смотреть материал на видео](https://www.youtube.com/watch?v=9qr3neFX0Rs&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/polimorfizm-chto-eto-takoe>

Приветствую всех на еще одном внеочередном занятии по ООП. И здесь я хочу затронуть еще один вопрос, который волнует моих подписчиков: что такое полиморфизм в ООП на Python? Если попытаться ответить в двух словах на этот вопрос, то можно было бы сказать так:

*Это возможность работы с совершенно разными объектами языка Python единым образом.*

Кажется, не особо стало понятнее? Поэтому давайте, как всегда, постигнем суть этого подхода на конкретном примере.

Вначале я продемонстрирую пример, где мы увидим один недостаток, который как раз исправляется с помощью полиморфизма. Предположим, у нас есть два класса Rectangle и Square:

class Rectangle:

    def \_\_init\_\_(self, w, h):

        self.w = w

        self.h = h

    def getPerRect(self):

        return 2\*(self.w+self.h)

class Square:

    def \_\_init\_\_(self, a):

        self.a = a

    def getPerSq(self):

        return 4\*self.a

И в них объявлены геттеры getPerRect и getPerSq для получения периметра соответствующих фигур: прямоугольника и квадрата. Далее, мы можем создать экземпляры этих классов и вывести в консоль значения периметров:

r1 = Rectangle(1, 2)

r2 = Rectangle(3, 4)

print(r1.getPerRect(), r2.getPerRect())

s1 = Square(10)

s2 = Square(20)

print(s1.getPerSq(), s2.getPerSq())

Все отлично, все работает. Но, теперь предположим, что все эти объекты помещаются в коллекцию:

geom = [r1, r2, s1, s2]

которую можно легко перебрать с помощью цикла for и где бы мы хотели получить значение периметра для каждой фигуры:

for g in geom:

    print( g.getPerRect() )

Как вы понимаете, когда в цикле очередь дойдет до объекта s1, возникнет ошибка, т.к. в классе Square отсутствует метод getPerRect. Конечно, зная, что в коллекции находятся объекты Rectangle и Square, можно было бы в цикле записать проверку:

for g in geom:

    if isinstance(g, Rectangle):

        print( g.getPerRect() )

    else:

        print(g.getPerSq())

и все заработает. Но у такого кода мало гибкости и, например, при добавлении еще одного класса:

class Triangle:

    def \_\_init\_\_(self, a, b, c):

        self.a = a

        self.b = b

        self.c = c

    def getPerTr(self):

        return self.a + self.b + self.c

Получим снова ошибку:

t1 = Triangle(1,2,3)

t2 = Triangle(4,5,6)

geom = [r1, r2, s1, s2, t1, t2]

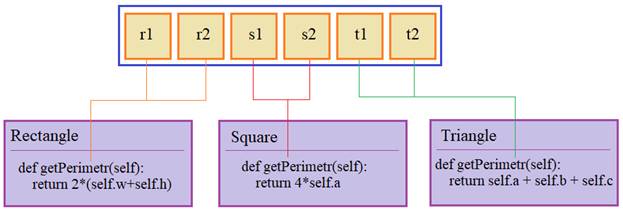
Конечно, в цикле for можно дополнительно проверить на соответствие классам Square и Triangle, но красоты и гибкости нашей программе это не придаст. Вот как раз здесь очень хорошо применим подход, который и называется полиморфизмом. Мы договоримся в каждом классе создавать методы с одинаковыми именами, например,

getPerimetr

Тогда в цикле будем просто обращаться к этому методу и получать периметры соответствующих фигур:

for g in geom:

    print( g.getPerimetr() )



То есть, Python «понимает» объект какого класса хранится в коллекции и соответственно из этого класса берет метод getPerimetr. Это и есть полиморфизм в действии, когда к разным объектам происходит обращение по одному и тому же имени метода и на выходе получаем разное поведение этой функции.

# Переопределение и перегрузка методов, абстрактные методы

[Смотреть материал на видео](https://www.youtube.com/watch?v=LVv8V94eZ9Y&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/pereopredelenie-i-peregruzka-metodov-abstraktnye-metody>

Продолжаем тему наследования классов в Python и поговорим о возможности переопределения публичных методов в дочерних классах. Давайте вернемся к нашему примеру из предыдущего занятия:

class Point:

    def \_\_init\_\_(self, x = 0, y = 0):

        self.\_\_x = x

        self.\_\_y = y

    def \_\_str\_\_(self):

        return f"({self.\_\_x}, {self.\_\_y})"

class Prop:

    def \_\_init\_\_(self, sp:Point, ep:Point, color:str = "red", width:int = 1):

        self.\_sp = sp

        self.\_ep = ep

        self.\_color = color

        self.\_width = width

class Line(Prop):

    def drawLine(self):

        print(f"Рисование линии: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_width}")

Для изменения координат графических примитивов добавим в класс Prop метод:

def setCoords(self, sp:Point, ep:Point):

    if sp.isDigit() and ep.isDigit():

        self.\_sp = sp

        self.\_ep = ep

    else:

        print("Координаты должны быть числами")

Который вначале проверяет: являются ли переданные значения числами или нет. Соответственно, в классе Point мы определим метод:

def isDigit(self):

    if (isinstance(self.\_\_x, int) or isinstance(self.\_\_x, float)) and \

       (isinstance(self.\_\_y, int) or isinstance(self.\_\_y, float)):

        return True

    return False

Теперь, мы можем создать экземпляр класса Line и менять его координаты:

l = Line( Point(1,2), Point(10,20) )

l.drawLine()

l.setCoords( Point(10.1,10), Point(30,30) )

l.drawLine()

Далее, нам приходит указание свыше устанавливать координаты для объекта Line только целочисленные. А все остальные примитивы могут принимать как целочисленные, так и вещественные координаты. Как нам это реализовать? Очевидно, нужно переопределить метод setCoords в дочернем классе Line. Сделаем это, запишем его в виде:

def setCoords(self, sp:Point, ep:Point):

    if sp.isInt() and ep.isInt():

        self.\_sp = sp

        self.\_ep = ep

    else:

        print("Координаты должны быть целочисленными")

А в класс Point добавим метод:

def isInt(self):

    if isinstance(self.\_\_x, int) and isinstance(self.\_\_y, int):

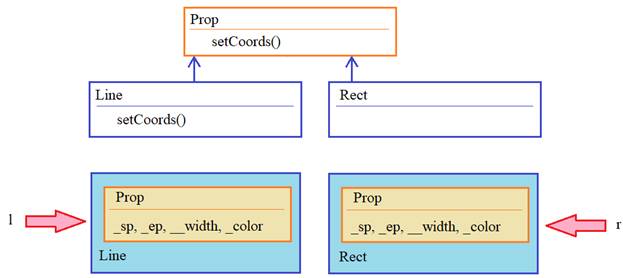
        return True

    return False

Теперь, при запуске программы, мы увидим сообщение, что координаты не целочисленные. То есть, это сработал переопределенный метод класса Line. А что же метод базового класса? Существует ли он? Да, и в данном случае, чтобы не дублировать вот этот код, мы можем его вызвать напрямую из этого класса:

Prop.setCoords(self, sp, ep)

В результате, у нас получается такая картина использования методов:



То есть, метод сначала ищется в дочернем классе и если не находится, то поиск продолжается в базовых. И, так как мы создали аналогичный в дочернем, то он и был взят, но при необходимости, мы всегда можем обратиться и к методу базового класса. Здесь всегда следует помнить, что методы в Python ведут себя как статические и обращаясь к ним из экземпляров, мы берем их непосредственно из классов. И только благодаря параметру self можем «понимать» с каким экземпляром класса работаем.

Но так переопределять можно только публичные методы, с частными (private) это не работает: они просто будут определяться независимо в своих классах. Я думаю это вполне очевидно и понятно.

Далее, в Python методы можно перегружать, то есть, выполнять разный функционал в зависимости от переданных данных. Например, можно перегрузить метод setCoords так, что при передаче только одного аргумента данные будут записываться в свойство \_sp, а при двух он бы работал так как и сейчас. В Python это делается следующим образом. Второй аргумент устанавливается по умолчанию в None:

def setCoords(self, sp:Point, ep:Point = None):

    if ep is None:

        if sp.isInt():

            self.\_sp = sp

        else:

            print("Координата должна быть целочисленной")

    else:

        if sp.isInt() and ep.isInt():

            Prop.setCoords(self, sp, ep)

        else:

            print("Координаты должны быть целочисленными")

И, теперь можно вызывать этот метод с одним или двумя аргументами:

l.setCoords( Point(10,10), Point(30,30) )

l.drawLine()

l.setCoords( Point(-10,-10) )

l.drawLine()

Метод setCoords в данном случае можно улучшить и реализовать через два вспомогательных метода:

def \_\_setOneCoord(self, sp):

    if sp.isInt():

        self.\_sp = sp

    else:

        print("Координата должна быть целочисленной")

def \_\_setTwoCoords(self, sp, ep):

    if sp.isInt() and ep.isInt():

        Prop.setCoords(self, sp, ep)

    else:

        print("Координаты должны быть целочисленными")

def setCoords(self, sp:Point, ep:Point = None):

    if ep is None:

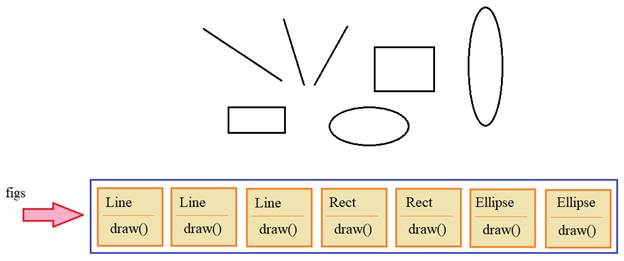
        self.\_\_setOneCoord(sp)

    else:

        self.\_\_setTwoCoords(sp, ep)

Так программа выглядит более читабельной.

Одним из преимуществ языка Python является возможность перебирать коллекции разнородных объектов и вызывать у них какой-нибудь единый метод. Например, если мы создаем графический редактор и пользователь нарисовал множество линий, прямоугольников, эллипсов и т.п., то мы можем их все сохранить в упорядоченной коллекции и отрисовывать, вызывая единый метод Draw, который в них реализуем.



Например, так:

class Line(Prop):

    def draw(self):

        print(f"Рисование линии: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_width}")

class Rect(Prop):

    def draw(self):

        print(f"Рисование прямоугольника: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_width}")

class Ellipse(Prop):

    def draw(self):

        print(f"Рисование эллипса: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_width}")

figs = []

figs.append( Line(Point(0,0), Point(10,10)) )

figs.append( Line(Point(10,10), Point(20,10)) )

figs.append( Rect(Point(50,50), Point(100,100)) )

figs.append( Ellipse(Point(-10,-10), Point(10,10)) )

for f in figs:

    f.draw()

В других языках программирования, таких как С++ или Java это реализуется через виртуальные методы. Здесь же весь этот механизм скрыт от нас и представляется естественным образом. Дополнительно здесь можно поставить «защиту» на случай, если метод draw не будет определен в дочернем классе. Для этого в базовом можно прописать такой же метод, но генерирующий специальное исключение:

def draw(self):

    raise NotImplementedError("В дочернем классе должен быть определен метод draw()")

Теперь, если убрать в каком-либо дочернем классе метод draw, то он будет вызван из базового класса и возникнет это исключение. Такие методы в Python иногда называют **абстрактными**, т.к. для корректной работы они требуют своего обязательного переопределения в дочерних классах. Хотя, на мой взгляд, это не совсем точное название, здесь лишь имитация абстракции в реализации метода.

## Задания для самоподготовки

1. Создайте базовый класс «Стол» и дочерние: «Прямоугольные столы» и «Круглые столы». Через конструктор базового класса передавайте размер поверхности стола: для прямоугольного – ширина и длина, для круглого – радиус. В дочерних классах реализуйте метод вычисления площади поверхности стола.

2. Создайте класс Animal (животное) и разные производные от него подклассы: Fox, Bird, Cat, Dog и т.п. Реализуйте у них общий метод say(), который бы возвращал звук, издаваемый этим животным. Создайте кортеж из нескольких этих экземпляров классов, переберите их в цикле и выведите в консоль их звуки (вызовите метод say()).

# Множественное наследование, функция super

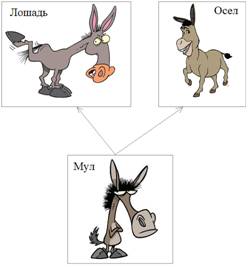
[Смотреть материал на видео](https://www.youtube.com/watch?v=4N-Q74IJd9U&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/mnozhestvennoe-nasledovanie-funkciya-super>

Продолжаем тему наследования классов в Python и поговорим о механизме множественного наследования.

*Множественное наследование – это возможность построения дочернего класса на основе нескольких базовых.*

Вот хорошая иллюстрация этого принципа: скрещивая лошадь с ослом, получаем мула.



В качестве примера множественного наследования построим класс графического примитива Line на основе двух классов: Styles и Pos. Если мы все запишем вот так:

class Point:

    def \_\_init\_\_(self, x = 0, y = 0):

        self.\_\_x = x

        self.\_\_y = y

    def \_\_str\_\_(self):

        return f"({self.\_\_x}, {self.\_\_y})"

class Styles:

    def \_\_init\_\_(self, color = "red", width = 1):

        print("Конструктор Styles")

        self.\_color = color

        self.\_width = width

class Pos:

    def \_\_init\_\_(self, sp:Point, ep:Point):

        print("Конструктор Pos")

        self.\_sp = sp

        self.\_ep = ep

class Line(Pos, Styles):

    def draw(self):

        print(f"Рисование линии: {self.\_sp}, {self.\_ep}, {self.\_color}, {self.\_width}")

l = Line( Point(10, 10), Point(100, 100), "green", 5 )

То при запуске программы увидим ошибку, т.к. первый вызываемый конструктор класса Pos принимает всего два аргумента, а мы передаем четыре. Давайте поправим этот момент и запишем его в таком виде:

class Pos:

    def \_\_init\_\_(self, sp:Point, ep:Point, \*args):

        print("Конструктор Pos")

        self.\_sp = sp

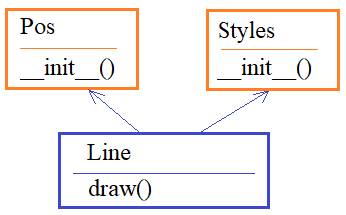
        self.\_ep = ep

        Styles.\_\_init\_\_(self, \*args)

Мы здесь добавили остаточные аргументы \*args и далее, вызвали конструктор второго базового класса Styles из класса Pos с этими остаточными аргументами. В результате у нас получилась полная инициализация данных как мы этого и хотели. Теперь, при вызове метода:

l.draw()

мы видим отображение всех указанных локальных свойств. И иерархия наших классов стала следующей:



Но при таком построении программы есть один большой недостаток. Если, например, мы в классе Line поменяем порядок наследования базовых классов:

class Line(Styles, Pos):

то при выполнении у нас возникнет ошибка, т.к. сначала будет вызван конструктор класса Styles, а в нем по прежнему всего два параметра. Ну, хорошо, давайте добавим и ему остаточные аргументы:

class Styles:

    def \_\_init\_\_(self, color, width, \*args):

        print("Конструктор Styles")

        self.\_color = color

        self.\_width = width

        Pos.\_\_init\_\_(self, \*args)

И, далее, мы должны, получается, еще вызвать конструктор класса Pos. Но, тогда у нас получится рекурсия: из Styles вызываем Pos, а из Pos – Styles! И такая путаница возникла всего из двух базовых классов! А представьте, если их будет десяток? Для выхода из такой ситуации как раз и была предложена функция super(). Она позволяет обойти всех предков в определенном порядке и только один раз. Давайте поставим вызов этой функции вместо конкретных классов:

super().\_\_init\_\_(\*args)

И, теперь, при запуске у нас нет никаких ошибок, кроме изменения порядка записи значений в атрибуты. Но, это логично, т.к. мы сначала вызвали Styles и записали координаты, а затем, Pos и записали через него свойства. Если поменять порядок наследования:

class Line(Pos, Styles):

то все будет в правильном порядке.

Но такой вариант множественного наследования тоже не очень хорош. Вызывать конструкторы с аргументами в этом случае порочная практика, т.к. даже незначительное изменение иерархии классов может привести к серьезным ошибкам. Поэтому, чаще всего, конструкторы при множественном наследовании записывают без аргументов. В нашем случае программу можно очень легко изменить, записав ее так. В класс Line добавим конструктор с аргументами:

def \_\_init\_\_(self, sp:Point, ep:Point, color = "red", width =1):

    self.\_sp = sp

    self.\_ep = ep

    self.\_color = color

    self.\_width = width

    super().\_\_init\_\_()

И, далее, просто вызываем super() и указываем конструкторы вышестоящих классов без аргументов:

class Styles:

    def \_\_init\_\_(self):

        print("Конструктор Styles")

        super().\_\_init\_\_()

class Pos:

    def \_\_init\_\_(self):

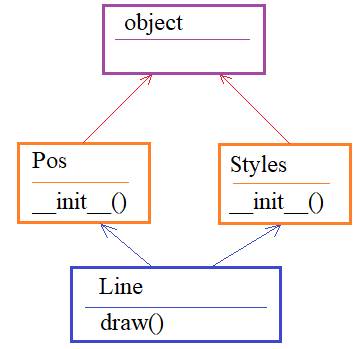
        print("Конструктор Pos")

        super().\_\_init\_\_()

Запускаем программу, и видим, что все работает абсолютно также. Но, зато теперь, мы можем наследовать классы в любом порядке:

class Line(Styles, Pos):

И это не приведет к какой-либо ошибке. Кстати, в действительности, на вершине иерархии наших классов стоит класс object, который наследуется по умолчанию любым классом в Python:



Так вот, функция super() перебирает предков в соответствии с алгоритмом поиска C3:

*C3 – поиск в дереве наследования классов Python 3 (MRO)*

Очень упрощенно его можно представить так:

* формируется список родителей в порядке их наследования
* в конец этого списка добавляются родители этих родителей (в том же порядке (и т.д.)
* если класс встречается второй раз, то он не заносится в список.

Более подробно об этом алгоритме можно почитать здесь:

[https://ru.wikipedia.org/wiki/C3-линеаризация](https://ru.wikipedia.org/wiki/C3-%D0%BB%D0%B8%D0%BD%D0%B5%D0%B0%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F)

В соответствии с алгоритмом MRO обход нашего дерева наследования будет происходить так:

Line → Pos → Styles → object

Кстати, для любого класса мы можем посмотреть этот список:

print( Line.\_\_mro\_\_ )

И получаем последовательность в соответствии с описанным алгоритмом.

## Задание для самоподготовки

Создайте дочерний класс Motherboard (материнская плата), которая наследуется от классов: CPU (процессор), GPU (графич. сопроцессор), Memory (память). В свою очередь CPU наследуется от классов: AMD и Intell, GPU от классов NVidia, GeForce.

Создайте экземпляр класса Motherboard и наполните ее конкретным содержимым (локальным свойствам этого объекта присвойте определенные значения). Определите вспомогательные методы в базовых классах и выведите итоговую информацию в консоль с помощью метода showInfo() класса Motherboard.

# Перегрузка операторов

[Смотреть материал на видео](https://www.youtube.com/watch?v=eHnkiDQO3lc&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/peregruzka-operatorov>

На этом занятии рассмотрим способы перегрузки операторов в классах. Что это такое? Давайте для примера создадим такой простой класс Clock (часы). В каждом экземпляре этого класса будем хранить время в виде секунд, отсчитывая от 00:00 часов ночи. Соответственно, число секунд не должно превышать значения:

24 \* 60 \* 60 = 86400 (число секунд в одном дне)

Поэтому, в конструкторе класса перед присвоением выполним операцию вычисления остатка от деления на это число:

self.\_\_secs = secs % self.\_\_DAY

Затем, реализуем метод getFormatTime(), который возвращает указанное время в формате:

01:32:02

Получаем такую реализацию класса:

class Clock:

    \_\_DAY = 86400   *# число секунд в дне*

    def \_\_init\_\_(self, secs:int):

        if not isinstance(secs, int):

            raise ValueError("Секунды должны быть целым числом")

        self.\_\_secs = secs % self.\_\_DAY

    def getFormatTime(self):

        s = self.\_\_secs % 60            *# секунды*

        m = (self.\_\_secs // 60) % 60    *# минуты*

        h = (self.\_\_secs // 3600) % 24  *# часы*

        return f"{Clock.\_\_getForm(h)}:{Clock.\_\_getForm(m)}:{Clock.\_\_getForm(s)}"

    @staticmethod

    def \_\_getForm(x):

        return str(x) if x > 9 else "0"+str(x)

И, далее, мы можем его вот так создавать и выводить время в консоль:

c1 = Clock(100)

print( c1.getFormatTime() )

Но теперь предположим, что мы хотели бы складывать времена вот так:

c2 = Clock(200)

c3 = c1 + c2

Если сейчас запустить программу, то увидим ожидаемую ошибку, что два таких класса складывать нельзя. И вот для таких случаев в Python предусмотрена возможность перегрузки операторов, чтобы они стали делать нужную нам операцию. В данном случае нужно перегрузить оператор сложения («+»). В действительности, когда он выполняется с классами, то в классе вызывается специальный метод

\_\_add\_\_(self, other)

вот его нам и нужно переопределить. Запишем в классе Clock следующие строки:

def \_\_add\_\_(self, other):

    if not isinstance(other, Clock):

        raise ArithmeticError("Правый операнд должен быть типом Clock")

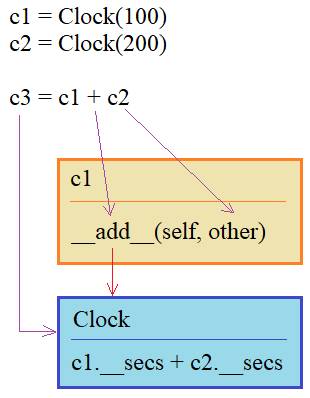
    return Clock(self.\_\_secs + other.getSeconds())

и добавим геттер getSeconds:

def getSeconds(self):

    return self.\_\_secs

Все, теперь при выполнении оператора сложения будет формироваться новый экземпляр класса Clock с суммой секунд от правого и левого операндов. Запускаем программу и видим, что все работает как мы и хотели.



Мало того, мы можем вычислять и вот такие операции:

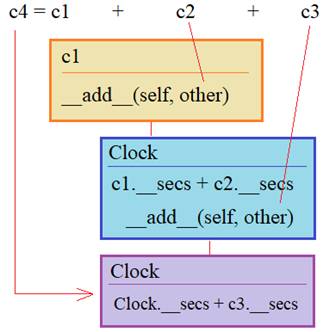
c1 = Clock(100)

c2 = Clock(200)

c3 = Clock(300)

c4 = c1 + c2 + c3

print( c4.getFormatTime() )



То есть, мы получили универсальный оператор сложения для часов.

Теперь посмотрим как можно перегрузить операторы типа

+=

для этого нужно переопределить метод

\_\_iadd\_\_(self, other)

Сделаем это следующим образом:

def \_\_iadd\_\_(self, other):

    if not isinstance(other, Clock):

        raise ArithmeticError("Правый операнд должен быть типом Clock")

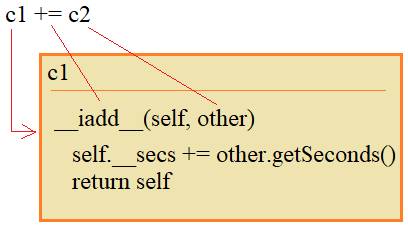
    self.\_\_secs += other.getSeconds()

    return self

Смотрите, мы здесь меняем значение секунд в самом экземпляре класса self и возвращаем этот же экземпляр, не создавая нового. В итоге операцию:

c1 += c2

можно представить так:



Или же все наши перегруженные операторы записать и так:

c1 = Clock(100)

c2 = Clock(200)

c3 = Clock(300)

c1 += c2+c3

Как это будет в деталях работать, я думаю, вы уже догадались. По аналогии могут быть перегружены и другие подобные операторы. Вот их краткий список:

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Метод оператора | Оператор | Метод оператора |
| x+y | \_\_add\_\_(self, other) | x += y | \_\_iadd\_\_(self, other) |
| x-y | \_\_sub\_\_(self, other) | x -= y | \_\_isub\_\_(self, other) |
| x\*y | \_\_mul\_\_(self, other) | x \*= y | \_\_imul\_\_(self, other) |
| x/y | \_\_truediv\_\_(self, other) | x /= y | \_\_itruediv\_\_(self, other) |
| x//y | \_\_floordiv\_\_(self, other) | x //= y | \_\_ifloordiv\_\_(self, other) |
| x % y | \_\_mod\_\_(self, other) | x %= y | \_\_imod\_\_(self, other) |

По аналогии можно перегружать операторы сравнения. Например, мы хотим сравнить время для двух часов:

if c1 == c2:

    print("Времена равны")

Для этого в классе Clock нужно переопределить метод:

def \_\_eq\_\_(self, other):

    if self.\_\_secs == other.getSeconds():

        return True

    return False

Тогда при одинаковом числе секунд:

c1 = Clock(100)

c2 = Clock(100)

оператор == возвратит True:

if c1 == c2:

    print("Времена равны")

Или же, метод eq можно записать в более краткой форме:

def \_\_eq\_\_(self, other):

    return self.\_\_secs == other.getSeconds()

Это будет одно и то же (по результату).

Далее, сравнение на неравенство можно сделать так:

def \_\_nq\_\_(self, other):

    return not self.\_\_eq\_\_(other)

И ниже записать:

c3 = Clock(300)

if c1 != c3:

    print("Времена не равны")

Как видите, это все работает довольно просто и очевидно. Основные операторы сравнения, следующие:

|  |  |
| --- | --- |
| Оператор | Метод оператора |
| x == y | \_\_eq\_\_(self, other) |
| x != y | \_\_nq\_\_(self, other) |
| x < y | \_\_lt\_\_(self, other) |
| x <= y | \_\_le\_\_(self, other) |
| x > y | \_\_gt\_\_(self, other) |
| x >= y | \_\_ge\_\_(self, other) |

Еще один интересный вариант перегрузки представляют методы доступа к объекту по ключу:

\_\_getitem\_\_(self, item)  
\_\_setitem\_\_(self, key, value)

Например, мы хотим получать значения часа, минут и секунд, обращаясь к экземпляру класса вот так:

print( c1["hour"], c1["min"], c1["sec"] )

Для этого переопределим метод \_\_getitem\_\_:

def \_\_getitem\_\_(self, item):

    if not isinstance(item, str):

        raise ValueError("Ключ должен быть строкой")

    if item == "hour":

        return (self.\_\_secs // 3600) % 24

    elif item == "min":

        return (self.\_\_secs // 60) % 60

    elif item == "sec":

        return self.\_\_secs % 60

    return "Неверный ключ"

И теперь, при запуске программы, видим желаемый результат.

c1 = Clock(80000)

У нас здесь идет некоторое дублирование кода, что нехорошо, но я сейчас не буду на это отвлекаться, вы уже сами самостоятельно можете улучшить эту программу.

Далее, используя переопределение метода \_\_setitem\_\_, можно реализовать функционал определения времени через ключи:

c1["hour"] = 10

Сам метод выглядит так:

def \_\_setitem\_\_(self, key, value):

    if not isinstance(key, str):

        raise ValueError("Ключ должен быть строкой")

    if not isinstance(value, int):

        raise ValueError("Значение должно быть целым числом")

    s = self.\_\_secs % 60  *# секунды*

    m = (self.\_\_secs // 60) % 60  *# минуты*

    h = (self.\_\_secs // 3600) % 24  *# часы*

    if key == "hour":

        self.\_\_secs = s + 60 \* m + value \* 3600

    elif key == "min":

        self.\_\_secs = s + 60 \* value + h \* 3600

    elif key == "sec":

        self.\_\_secs = value + 60 \* m + h \* 3600

Вот мы с вами рассмотрели основные принципы перегрузки операторов класса. И, если у вас возникнет необходимость в реализации подобного функционала, то теперь знаете как это делается.

## Задания для самоподготовки

1. Напишите класс Point3D для хранения координат в трехмерном пространстве (x, y, z). Реализуйте перегрузку операторов сложения, вычитания, умножения и деления для этого класса. Также сделайте возможность сравнения координат между собой и запись/считывание значений через ключи: “x”, “y”, “z”.

2. Напишите класс Matrix для работы с матрицами. Реализуйте перегрузку операторов сложения и вычитания для матриц равных размеров. Перегрузите оператор умножения для матриц, которые могут быть перемножены. Также сделайте возможность сравнения матриц между собой (на равенство и неравенство).

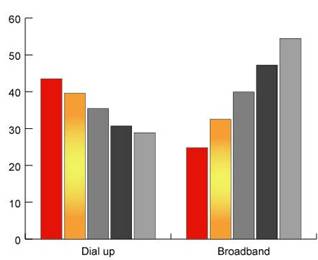
3. Напишите класс Complex для работы с комплексными числами. Реализуйте операторы сложения, вычитания и умножения. Также сделайте возможность присвоения действительных и мнимых значений через ключи «rel» и «img» и через свойства rel, img, реализованных с помощью дескрипторов.

# Собственные исключения и итерабельные объекты

[Смотреть материал на видео](https://www.youtube.com/watch?v=0zuBB1YArCQ&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/isklyucheniya-i-iterabelnye-obekty>

На этом занятии я решил привести пример класса для хранения клип-арт изображений, то есть, изображений составленных из графических примитивов. Их характерная особенность – однородный фон и небольшая цветовая гамма, например, деловые графики:



На этом примере мы посмотрим, как можно создавать свои собственные классы исключений и итераторы. Но для начала определимся как будем хранить пиксели изображения. В локальных свойствах мы сохраним ширину и высоту изображения (в пикселах):

self.\_\_width = width

self.\_\_height = height

цвет фона:

self.\_\_background = background

и две коллекции: для хранения уникальных цветов (палитра) и для хранения пикселей, чей цвет отличается от цвета фона:

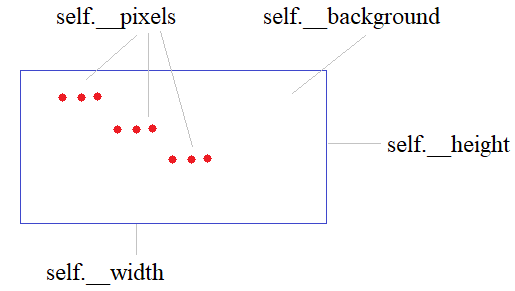
self.\_\_colors = { self.\_\_background}

self.\_\_pixels = {}

Причем, коллекция \_\_pixels – это словарь, в котором ключами будут кортежи (x,y) с координаты точек, а значениями – цвета:

self.\_\_pixels[(x,y)] = color

Палитру будем хранить в виде множества \_\_colors, которое гарантирует нам уникальность значений.



В результате, получаем такой конструктор класса Image:

class Image:

    def \_\_init\_\_(self, width, height, background="\_"):

        self.\_\_background = background

        self.\_\_pixels = {}

        self.\_\_width = width

        self.\_\_height = height

        self.\_\_colors = {self.\_\_background}

Далее, создадим два простых свойства в этом классе для получения или изменения высоты и ширины изображения:

@property

def width(self):

    return self.\_\_width

@width.setter

def width(self, width):

    self.\_\_width = width

@property

def height(self):

    return self.\_\_height

@height.setter

def height(self, height):

    self.\_\_height = height

И, затем, переопределим методы \_\_setitem\_\_ и \_\_getitem\_\_, чтобы мы могли обращаться к отдельным пикселам по синтаксису:

img[1,1] = "\*"color = img[5,5]

Метод \_\_setitem\_\_ запишем так:

def \_\_setitem\_\_(self, coord, color):

    self.\_\_checkCoord(coord)

    if color == self.\_\_background:

        self.\_\_pixels.pop(coord, None)

    else:

        self.\_\_pixels[coord] = color

        self.\_\_colors.add(color)

Мы здесь сначала проверяем: являются ли наши координаты кортежем с двумя значениями. Для этого реализуем вспомогательный метод

def \_\_checkCoord(self, coord):

    if not isinstance(coord, tuple) or len(coord) != 2:

        raise KeyError("Координаты точки должны быть двумерным кортежем")

    if not (0 <= coord[0] < self.\_\_width) or not (0 <= coord[1] < self.\_\_height):

        raise KeyError("Значение координаты выходит за пределы изображения")

который генерирует исключения, если координаты не удовлетворяют нашим условиям.

Если проверка проходит, то для пикселей с цветом, равным цвету фона удаляем их из нашего словаря \_\_pixels, т.к. там должны находиться только пиксели с отличающимися от фона цветами. Удаление делаем с помощью метода pop, в котором первым параметром указываем удаляемый ключ, а вторым – возвращаемое значение, если такой ключ не будет найден. При таком вызове метода pop он не будет генерировать исключение в случае отсутствия ключа, что нам и нужно.

Если же добавляется пиксел не с цветом фона, то мы его прописываем в словаре: если он там уже есть, то будет перезаписан цвет, если нет, то будет создан. И в коллекцию \_\_colors добавляет этот цвет. Так как коллекция представляет собой множество, то цвет будет добавлен только в том случае, если он там отсутствует. То есть, у нас там будут только уникальные цвета.

По аналогии пропишем метод

def \_\_getitem\_\_(self, coord):

    self.\_\_checkCoord(coord)

    return self.\_\_pixels.get(coord, self.\_\_background)

Он просто берет значение из словаря, если оно там есть. Иначе, будет возвращено значение фона.

Все, в первом приближении наш класс Image готов. Давайте посмотрим на его работу, запишем такие строчки:

img = Image(20, 4)

img[1,1] = "\*"; img[2,1] = "\*"; img[3,1] = "\*"

for y in range(img.height):

    for x in range(img.width):

        print(img[x,y], sep=" ", end="")

    print()

После запуска в консоли видим импровизированное изображение. Кстати, если в качестве координат мы укажем неверный формат, например, так:

img[1,1,1] = "\*"

то возникнет исключение KeyError. Давайте вместо этого исключения создадим свое, новое в виде класса CoorError. В Python это делается элементарно. Достаточно унаследовать этот класс от базового класса Exception и мы автоматически получим полный функционал для отслеживания нашего нового исключения:

class CoordError(Exception):

    pass

И, далее в методе \_\_checkCoord() пропишем его:

def \_\_checkCoord(self, coord):

    if not isinstance(coord, tuple) or len(coord) != 2:

        raise CoordError("Координаты точки должны быть двумерным кортежем")

    if not (0 <= coord[0] < self.\_\_width) or not (0 <= coord[1] < self.\_\_height):

        raise CoordError("Значение координаты выходит за пределы изображения")

Теперь при запуске программы мы видим свой класс исключения.

## Итерабельные классы

Продолжим улучшать функционал нашего класса Image и сделаем его итерабельным, то есть таким, чтобы можно было напрямую перебирать пиксели в циклах for вот так:

for row in img:

    for pixel in row:

        print(pixel, sep=" ", end="")

    print()

Давайте для начала посмотрим общий принцип создания итерабельных классов. Для этого у него нужно переопределить два обязательных метода:

\_\_iter\_\_(self)  
\_\_next\_\_(self)

Первый вызывается когда объект берется для итерации, а второй возвращает значение текущей итерации. Давайте вначале посмотрим как вообще можно объявить свой итерабельный класс. В самом простом варианте это можно сделать так:

class MyIter:

    def \_\_init\_\_(self, limit):

        self.\_\_num = 0

        self.\_\_limit = limit

    def \_\_iter\_\_(self):

        return self

    def \_\_next\_\_(self):

        if self.\_\_num >= self.\_\_limit:

            raise StopIteration

        self.\_\_num += 1

        return self.\_\_num

И, затем, перебирать с помощью цикла for:

it = MyIter(10)

for i in it:

    print(i)

То есть, цикл for вызывает метод \_\_next\_\_ пока не возникнет исключение StopIteration. Вот общий принцип построения собственных итераторов.

Давайте теперь применим этот принцип для перебора пикселей нашего изображения и, как мы уже отмечали, хотим это делать вот так:

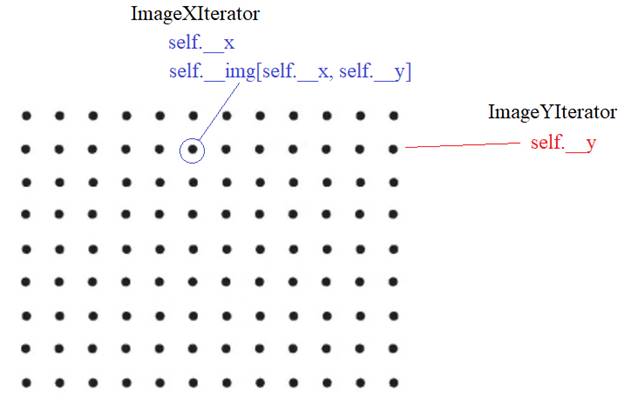
for row in img:

    for pixel in row:

        print(pixel, sep=" ", end="")

    print()

То есть, первый цикл перебирал бы строки, а второй – столбцы. Здесь нам понадобятся два итератора для перебора строк и столбцов соответственно.



Значит, когда мы обращаемся просто к объекту img в цикле for:

for row in img

он должен вернуть экземпляр итерабельного объекта для перехода по строкам, то есть, ImageYIterator. Поэтому, в самом классе Image переопределим метод \_\_iter\_\_:

def \_\_iter\_\_(self):

    return ImageYIterator(self)

В качестве аргумента передаем ему ссылку на экземпляр изображения, с которым он будет работать. А сам класс ImageYIterator будет выглядеть так:

class ImageYIterator:

    def \_\_init\_\_(self, img):

        self.\_\_limit = img.height

        self.\_\_img = img

        self.\_\_y = 0

    def \_\_iter\_\_(self):

        return self

    def \_\_next\_\_(self):

        if self.\_\_y >= self.\_\_limit:

            raise StopIteration

        self.\_\_y += 1

        return ImageXIterator(self.\_\_img, self.\_\_y-1)

Смотрите, мы здесь изменяем приватное свойство \_\_y, указывающее текущую строку, а переопределенный метод \_\_next\_\_ возвращает экземпляр объекта для итераций по столбцам изображения. То есть, переменная row в цикле for будет ссылаться на этот экземпляр класса ImageXIterator. Поэтому вложенный цикл может спокойно его перебирать:

for pixel in row

Класс для второго итерационного объекта выглядит так:

class ImageXIterator:

    def \_\_init\_\_(self, img, y:int):

        self.\_\_limit = img.width

        self.\_\_y = y

        self.\_\_img = img

        self.\_\_x = 0

    def \_\_iter\_\_(self):

        return self

    def \_\_next\_\_(self):

        if self.\_\_x >= self.\_\_limit:

            raise StopIteration

        self.\_\_x += 1

        return self.\_\_img[self.\_\_x-1, self.\_\_y]

Мы здесь уже переходим по координате x, в методе \_\_next\_\_ она увеличивается и возвращается текущее значение пиксела, используя ссылку на класс Image. Поэтому pixel уже принимает конкретное значение цвета точки изображения.

Здесь может возникнуть вопрос: почему бы нам не сделать сам класс Image в виде итерабельного объекта и не создавать дополнительно класс ImageYIterator? Однако, тут есть один нюанс: перебор данных в таких объектах происходит только один раз. И при таком подходе мы могли бы только один раз выводить изображение. При повторной попытке, сразу возникло бы исключение StopIteration и никакого вывода не было бы. Вот для этого и были созданы эти два дополнительных итератора.

## Задания для самоподготовки

1. Измените класс Image так, чтобы в нем появился метод resize(width, height). Если новая ширина или высота меньше текущего значения, все цвета, оказавшиеся за пределами новых границ изображения, должны удаляться. Если в качестве нового значения ширины или высоты передается None, соответствующее значение ширины или высоты должно оставаться без изменений.

2. Реализуйте класс ListInt для хранения списка целых чисел и сделайте его итерируемым так, чтобы значения возвращались с конца в начало.

3. Создайте класс Persons для хранения списка уникальных посетителей клуба. Сделайте возможность перебора гостей итератором(ми) следующим образом:

* с выводом только их имени;
* с выводом только их возраста;
* с выводом только их фамилии.

# Функторы

[Смотреть материал на видео](https://www.youtube.com/watch?v=-ZY0KHGOQoY&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/funktory>

Здравствуйте, дорогие друзья! Продолжим углубляться в ООП на Python и вначале этого занятия поговорим о возможности выполнения экземпляра класса как функции. То есть,

*Функторы – это объекты классов, которые можно выполнять как функции.*

Например, создадим класс счетчик:

class Counter:

    def \_\_init\_\_(self):

        self.\_\_counter = 0

И далее, положим, что мы хотим вызывать его экземпляр:

c1 = Counter()

как функцию, то есть:

c1()

Для этого нужно выполнить перегрузку оператора () с помощью реализации метода \_\_call\_\_. Пусть он у нас будет таким:

def \_\_call\_\_(self, \*args, \*\*kwargs):

    self.\_\_counter += 1

    print( self.\_\_counter )

    return self.\_\_counter

Теперь, при каждом вызове этого класса в виде функции, мы будем попадать в метод call и увеличивать счетчик на 1:

c1()

c1()

Также можно добавить еще один экземпляр класса, который будет работать совершенно независимо от первого:

c2 = Counter()

c2()

c2()

Этот довольно простой пример показывает общий принцип реализации функторов. Давайте немного усложним задачу и построим функтор, который бы убирал в начале и конце строки заданные символы. Для этого объявим вот такой класс:

class StripChars:

    def \_\_init\_\_(self, chars):

        self.\_\_chars = chars

    def \_\_call\_\_(self, \*args, \*\*kwargs):

        if not isinstance(args[0], str):

            raise ValueError("Аргумент должен быть строкой")

        return args[0].strip(self.\_\_chars)

И вызовем его как функцию:

s1 = StripChars("?:!.; ")

print( s1(" Hello World! ") )

На выходе получаем строку с удаленными символами, перечисленные в конструкторе. Но это опять же лишь пример функтора в виде класса. Если такая (подобная) задача встречается на практике, то ее, как правило, реализуют через замыкания, то есть, создают функцию с определением внутри нее еще одной функции:

def StripChars(chars):

    def stringStrip(string):

        if not isinstance(string, str):

            raise ValueError("Аргумент должен быть строкой")

        return string.strip(chars)

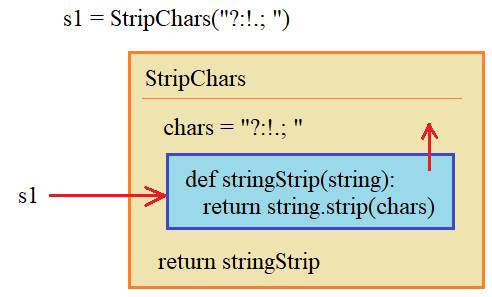
    return stringStrip

И, далее, следует аналогичный вызов:

s1 = StripChars("?:!.; ")

print( s1(" Hello World! ") )

Только теперь переменная s1 ссылается не на экземпляр класса, а на функцию stringStrip, которая возвращается функцией StripChars. Причем, в момент ее вызова создается контекст выполнения данной функции, фактически новый объект, и мы ссылаемся на функцию внутри этого объекта:



Сама же внутренняя функция stringStrip имеет ссылку на внешний контекст, то есть, на объект, созданный StipChars. Поэтому сборщик мусора не уничтожает его.

Из этой иллюстрации хорошо видно, что если создать еще один такой же объект:

s2 = StripChars("?:!.; ")

print( s2(" Hello? ") )

то они будут работать совершенно независимо, т.к. функция stringStrip также динамически создается внутри контекста выполнения функции StipChars. Если, например, вывести их id:

print( id(s1), id(s2), sep="**\n**")

то они будут совершенно разными.

Итак, из всего этого можно заключить, что если нам нужно выполнить какие-то простые операции, то лучше использовать замыкания. Но, если требуется реализовать более сложный механизм, предполагающий элементы наследования и полиморфизма, то следует задуматься об использовании функторов.

# Декораторы @classmethod и @staticmethod

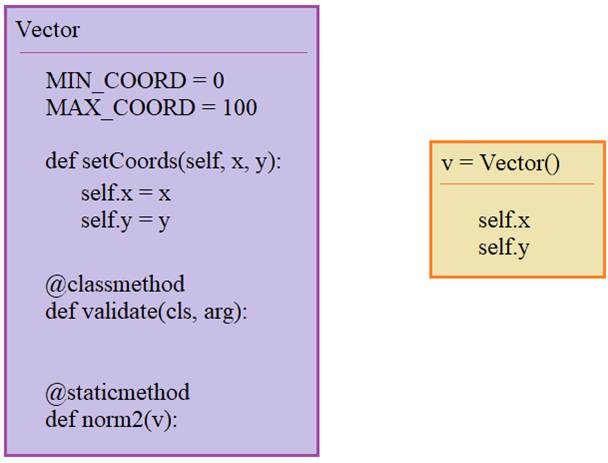
[Смотреть материал на видео](https://www.youtube.com/watch?v=rZY9CJn1y2E&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/dekoratory-classmethod-i-staticmethod>

Приветствую всех на внеочередном занятии по ООП. Меня часто спрашивают: чем отличаются обычные методы класса от методов, объявленных декораторами:

@classmethod и @staticmethod

Давайте разбираться. На уровне теории это выглядит так. Предположим, у нас есть класс Vector, который содержит два атрибута: MIN\_COORD = 0, MAX\_COORD = 100; обычный метод setCoords; метод уровня класса: validate и статический метод – norm2:



Так вот, все обычные методы в качестве первого параметра обязательно должны получать ссылку на экземпляр этого класса. И благодаря ей, они могут менять состояние как объекта класса, так и получать доступ к свойствами и методам самого класса Vector.

Когда метод класса объявляется через декоратор @classmethod, то первым параметром ему нужно передать сам класс, а не его экземпляр. В результате, такой метод может обращаться ко всем свойствам и методам текущего класса Vector, но не к экземпляру (из, которого он может быть вызван).

Наконец, третий тип методов, объявленные через декоратор @staticmethod, являются, своего рода, совершенно изолированными и от свойств и методов класса и, уж тем более, от свойств и методов экземпляра. Их можно воспринимать как отдельные, самостоятельные функции, объявленные внутри класса. Обычно, это делают для удобства, т.к. их функционал так или иначе связан с тематикой класса.

Итак, давайте на практике посмотрим, что все это значит. Я объявлю класс Vector вот в таком виде:

class Vector:

MIN\_COORD = 0

MAX\_COORD = 100

def setCoords(self, x, y):

if Vector.validate(x) and Vector.validate(y):

self.x = x

self.y = y

@classmethod

def validate(cls, arg):

if arg >= cls.MIN\_COORD and arg <= cls.MAX\_COORD:

return True

return False

@staticmethod

def norm2(x, y):

return x\*x + y\*y

И для удобства открою консоль Python, где импортирую этот класс из модуля ex1:

from ex1 import Vector

Далее, смотрите, непосредственно через класс Vector можно вызвать метод класса:

Vector.validate(5)

или, статический метод:

Vector.norm2(1, 2)

Но нельзя вызвать обычный метод:

Vector.setCoords(1, 2)

Так как здесь первым параметром нужно указывать ссылку на экземпляр класса self, из которого подразумевается вызов этого метода. У нас же, нет никакого объекта, только сам класс, отсюда и возникает такая ошибка. Давайте его создадим:

v = Vector()

Тогда можно вызвать setCoords, либо через экземпляр класса:

v.setCoords(1, 2)

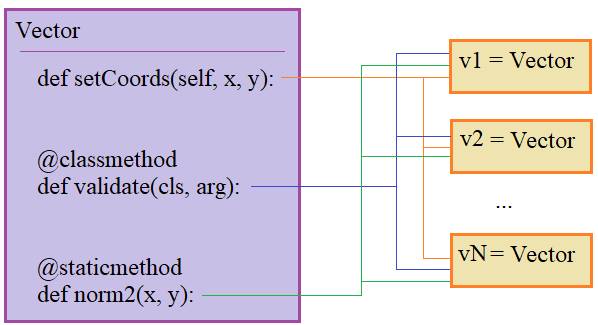
либо, через сам класс, но уже с явной передачей первого параметра – ссылки на ранее созданный объект:

Vector.setCoords(v, 10, 20)

Я думаю, этот момент понятен. И теперь самое главное: зачем это надо и когда какой метод объявлять? В некотором смысле ответ на этот вопрос дает теория, рассмотренная вначале.

1. Если нам нужен метод, который бы «работал» с атрибутами экземпляров классов, то это однозначно обычный метод класса с первым параметром self, который указывает на текущий объект.
2. Если нужен метод, который можно вызывать непосредственно из класса (или экземпляра) и, который бы имел доступ к свойствам и методам этого класса, то его следует объявить как метод класса через декоратор @classmethod.
3. Если нужен метод, который можно вызывать непосредственно из класса, но доступ к его атрибутам не предполагается, то достаточно его объявить как статический через декоратор @staticmethod.

И вот здесь есть один тонкий момент, особенно для тех, кто программирует на других языках, например, C++ или Java. Как бы мы ни объявляли методы внутри класса, экземпляры этих классов, при их вызове, будут обращаться напрямую к этому классу:



То есть, декораторы @classmethod и @staticmethod лишь как бы ограничивают область видимости внутри этих методов, но доступ к ним из вне происходит по одной схеме.

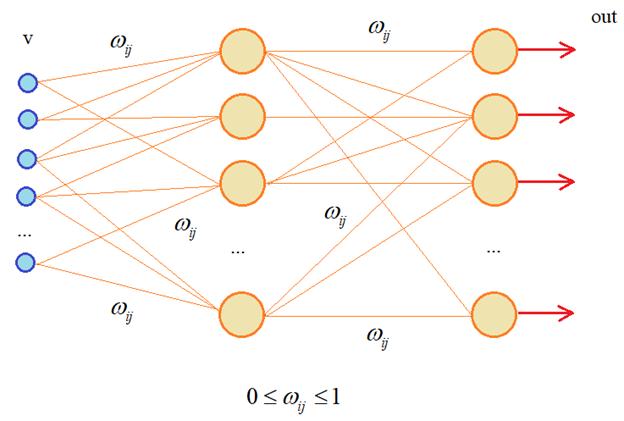
Вот что из себя представляют обычные методы, методы класса и статические методы.

# Пример использования ООП (нейронная сеть)

[Смотреть материал на видео](https://www.youtube.com/watch?v=KlW6O-5pcuU&list=PLA0M1Bcd0w8zo9ND-7yEFjoHBg_fzaQ-B" \t "_blank)

<https://proproprogs.ru/python_oop/neyronnaya-set>

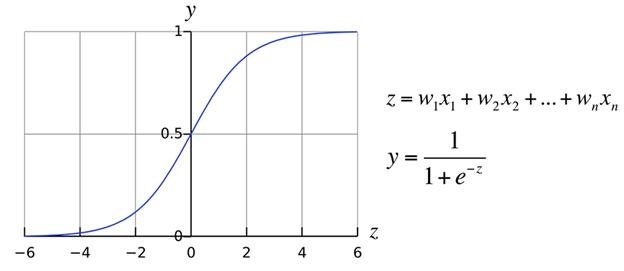
На этом занятии мы применим наши знания ООП в Python и реализуем простую нейронную сеть. Пару слов, о том что это такое. Нейронная сеть представляет собой вот такую распределенную структуру. Здесь у нас имеется входной слой, далее несколько скрытых слоев нейронов и затем, выходной слой.



Принцип ее работы такой. На вход подаются данные, представленные в виде вещественных чисел от 0 до 1. Далее, эти числа проходят по связям между нейронами с разной степенью затухания, определяемой множителем w\_ij. На входе каждого нейрона следующего слоя эти сигналы суммируются с учетом этих весов и на основе этой суммы формируется выходной сигнал соответствующего нейрона. Далее, эти сигналы подаются на нейроны следующего уровня, и так до последнего выходного слоя. По этим выходным числовым значениям делается вывод о свойствах входного сигнала. Разумеется, чтобы нейронная сеть работала, она должна быть обучена, то есть, веса ее связей должны быть должным образом настроены. Этот этап останется за рамками нашего занятия. Мы здесь лишь реализуем построение нейронной сети со случайными весами связей, а внутри каждого нейрона суммарный сигнал будет преобразовываться в выходной с помощью вот такой сигмоидальной функции:

Здесь z – это входной суммарный взвешенный сигнал, а y – выходное значение нейрона. Как видите, входной сигнал z из диапазона (-∞; +∞) масштабируется в диапазон (0;1). Поэтому эту функцию еще называют масштабирующей. Она гарантирует, что выходные значения всегда будут вещественными числами от 0 до 1.

Итак, реализуем такую нейронную сеть, схематично она представлена вот в таком виде. И начнем с описания класса нейронов.



Для этого создадим отдельный файл, в котором будет класс Neuro:

import math

class Neuro:

    def \_\_init\_\_(self, list\_in, list\_out):

        self.\_\_in = list\_in

        self.\_\_out = list\_out

        self.value = 0        *# выходной сигнал с нейрона*

    @property

    def list\_in(self):

        return self.\_\_in

    @list\_in.setter

    def list\_in(self, lst):

        self.\_\_in = lst

    @property

    def list\_out(self):

        return self.\_\_out

    @list\_out.setter

    def list\_out(self, lst):

        self.\_\_out = lst

    def act(self, x):

        return 1/(1+math.exp(-x))

Далее, создадим модуль для связей между нейронами:

class Link:

    def \_\_init\_\_(self, n\_in, n\_out, w = 0):

        self.\_\_in = n\_in

        self.\_\_out = n\_out

        self.\_\_w = w

    @property

    def n\_in(self):

        return self.\_\_in

    @property

    def n\_out(self):

        return self.\_\_out

    @property

    def w(self):

        return self.\_\_w

И, наконец, класс, описывающий создание и работы НС:

import random

from neuro import Neuro

from link import Link

class Network:

    def \_\_init\_\_(self, \*args):

        self.\_\_nlayers = len(args)  *# число слоев*

        self.\_\_neuros = args   *# число нейронов в кажом слое*

        self.\_\_layers = []

*#создаем нейроны в каждом слое*

        for i in range( self.\_\_nlayers ):

            self.\_\_layers.append( [Neuro([], []) for n in range(self.\_\_neuros[i])] )

*#создаем связи между нейронами*

        for i in range( self.\_\_nlayers ):

            for neuro in self.\_\_layers[i]:      *# перебираем нейроны i-го слоя*

                list\_in = 0 if i == 0 else [ Link(n\_in, neuro, random.random()) for n\_in in self.\_\_layers[i-1] ]

                list\_out = 0 if i == self.\_\_nlayers-1 else [Link(neuro, n\_out, random.random()) for n\_out in self.\_\_layers[i+1]]

                neuro.list\_in = list\_in

                neuro.list\_out = list\_out

    def run(self, v):

*# подаем на вход нейронов сигнал v*

        for neuro, inp in zip(self.\_\_layers[0], v):

            neuro.value = neuro.list\_in = inp

*# проводим сигнал по нейронной сети*

        for i in range(1, self.\_\_nlayers):

            for neuro in self.\_\_layers[i]:    *# перебираем нейроны i-го слоя*

                v = [ (link.n\_in.value\*link.w) for link in neuro.list\_in ]

                neuro.value = neuro.act( sum(v) )

    def output(self):

        return [ neuro.value for neuro in self.\_\_layers[-1] ]

Все это мы можем использовать, например, так:

from network import Network

net = Network(10, 5, 5)

net.run([1, 0.5, 0.1, 0.2, 0.7, 0.9, 1, 0.6, 0.3, 0.1])

out = net.output()

print( out )

Все, нейронная сеть готова!