**Общая методичка по прохождению курса ООП в Python**

**(Автор: Нерезов Алексей)  
(Год: 2022)**

**Урок №0: Концепция ООП простыми словами:**

ООП – объектно-ориентированное программирование.

**Классы и объекты классов**

Почти всегда в программах мы оперируем объектами данных, например, создаётся программа для учёта животных, в частности котов и кошек, значит, здесь мы имеем дело с объектами – коты. И было бы хорошо все данные связанные с котами представить в программе как единое целое, как раз это мы можем сделать с помощью класса.



В данном случае я представил класс в виде синего прямоугольника, имя этого класса – Cats. Этот класс мы можем воспринимать как некий шаблон, по которому в дальнейшем будут формироваться данные о котах. В этом шаблоне есть 3 свойства – порода, имя и возраст. Программист сам решает, сколько и какие свойства будут в его классе.



А вот конкретные коты – это объекты этого класса. Все эти объекты созданы по образу и подобию класса Cats, но в сущности это три совершенно разных объекта. Далее в программе мы можем работать с этими объектами котами как с единым целым.

Вообще класс может содержать не только свойства (атрибуты) но и методы (функции) определенных для работы с классом и его объектами.

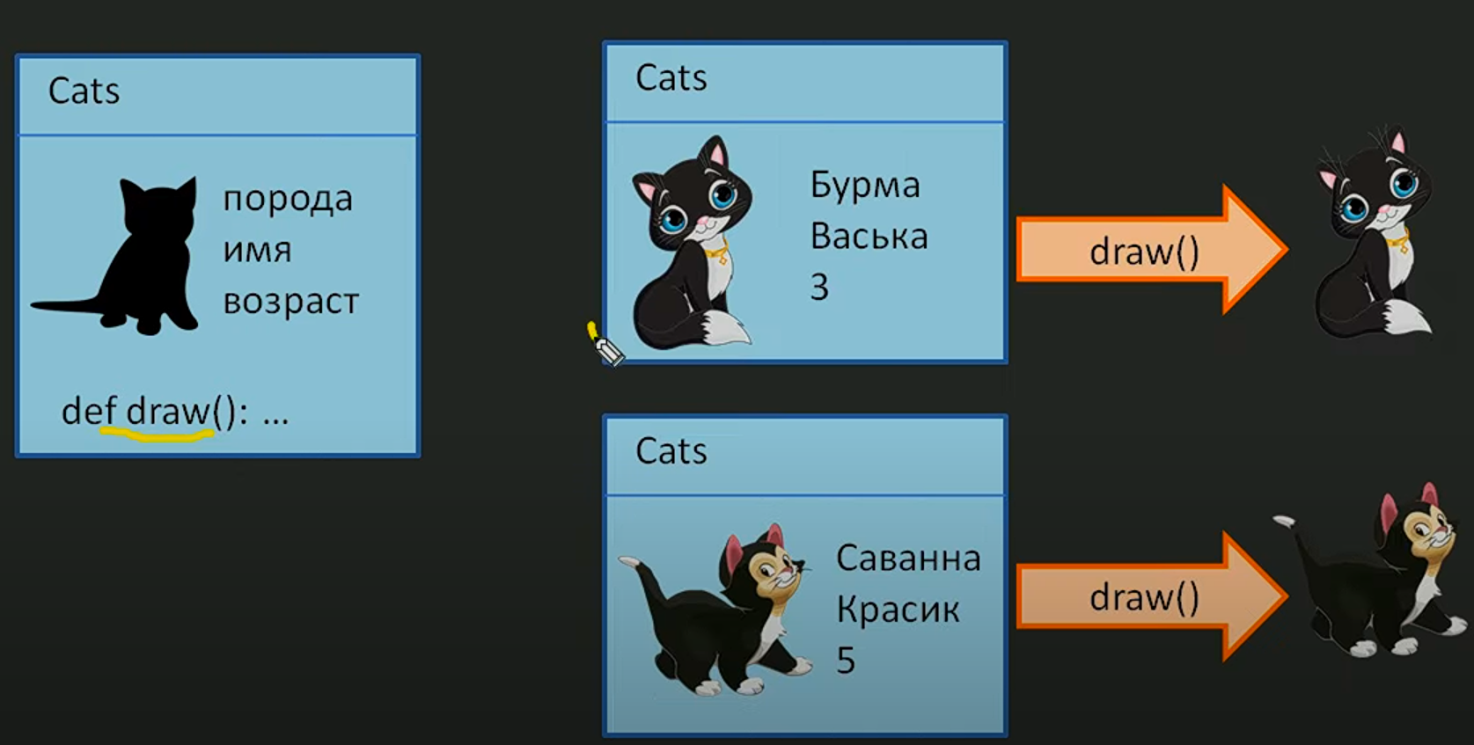
**Примечание 1:**

**Класс** – в объектно-ориентированном программировании, модель для создания объектов определённого типа, описывающая их структуру (набор полей и их начальное состояние) и определяющая алгоритмы (функции или методы) для работы с объектами этого класса.

**Примечание 2:**

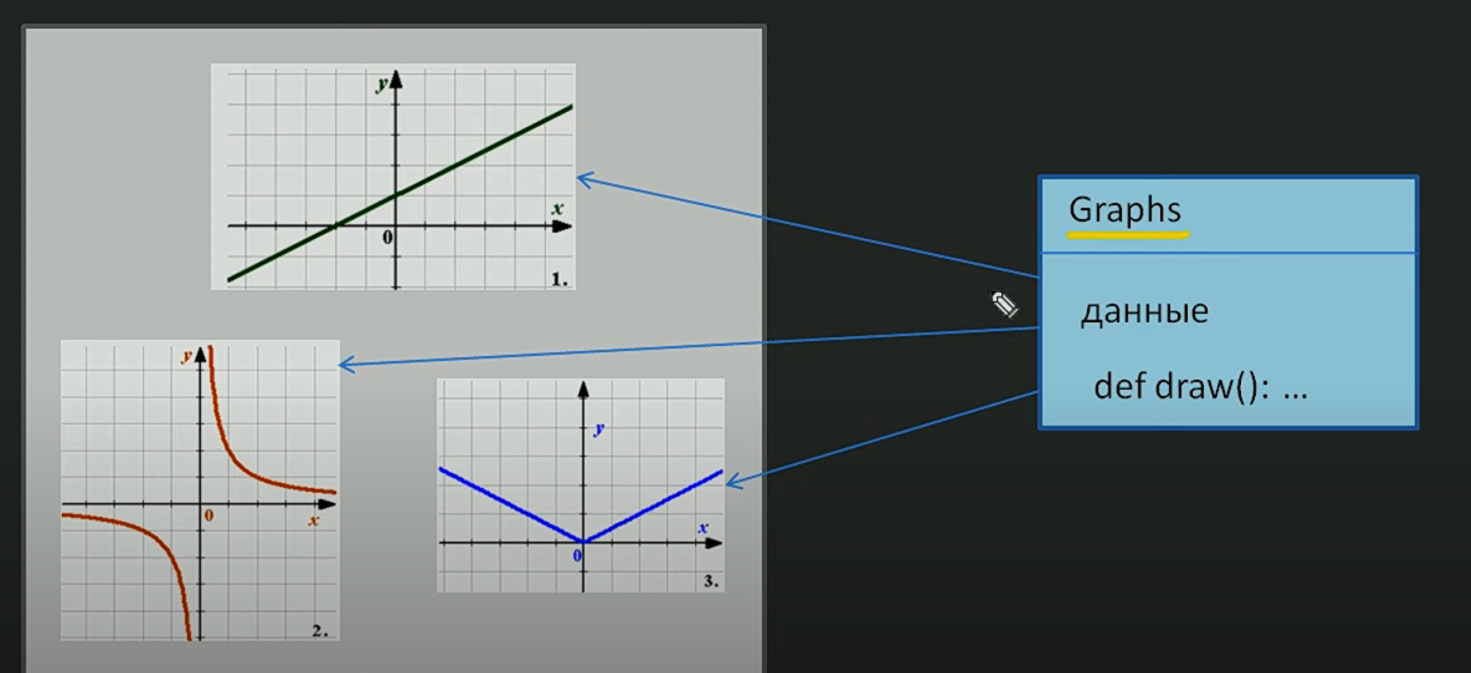
**Объект класса** – некоторая сущность в цифровом пространстве, обладающая определённым состоянием и поведением, имеющая определённые свойства (**атрибуты**) и операции над ними (**методы**). Как правило, при рассмотрении объектов выделяется то, что объекты принадлежат одному или нескольким классам, которые определяют поведение (являются моделью) объекта.

То есть класс может описывать некий алгоритм, фрагмент программы присущий именно этому объекту.



То есть видите, как удобно можно оперировать объектами на уровне классов.

**Пример использования ООП:**



**Примечание 3:**

Класс должен восприниматься как единая целостная конструкция. И все внутренние манипуляции с его данными должны быть сокрыты в этом классе, и в идеале недоступны извне.

**Инкапсуляция**



Это как с реальными вещами, когда мы покупаем автомобиль, то нам важно как им управлять, какие у него характеристики (разрешенные свойства и методы), но нам совершенно неважно как работает двигатель, почему крутятся колёса, из чего сделаны стёкла и т.д. (скрытые свойства и методы). То есть машина в целом некая капсула, которая внутри себя скрывает огромное количество разных узлов взаимодействующих между собой и в идеале нам совершенно не важен их внутренний алгоритм функционирования. Важен только конечный результат – завели и поехали.

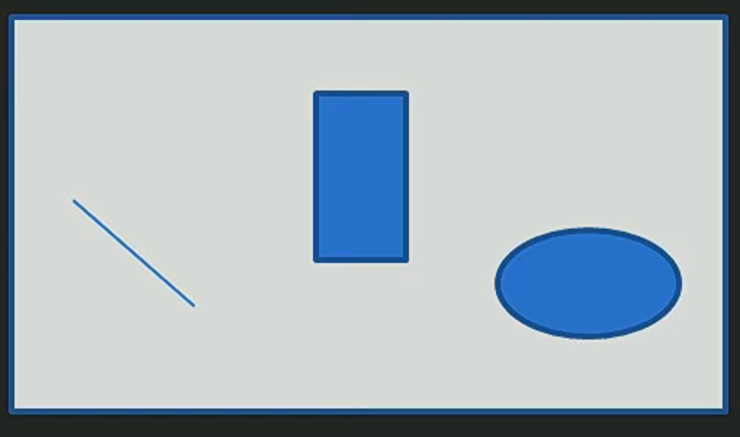
Та же идея заложена и в классах – они вроде капсул, которые содержат в себе данные (свойства) и методы (действия), и эти действия определяют внутренний алгоритм функционирования данного класса.

Так вот чтобы пользователь класса не мог случайно или намеренно вмешаться в его внутреннюю работу – данные и методы нужно скрывать. Такой механизм в ООП называется инкапсуляцией.

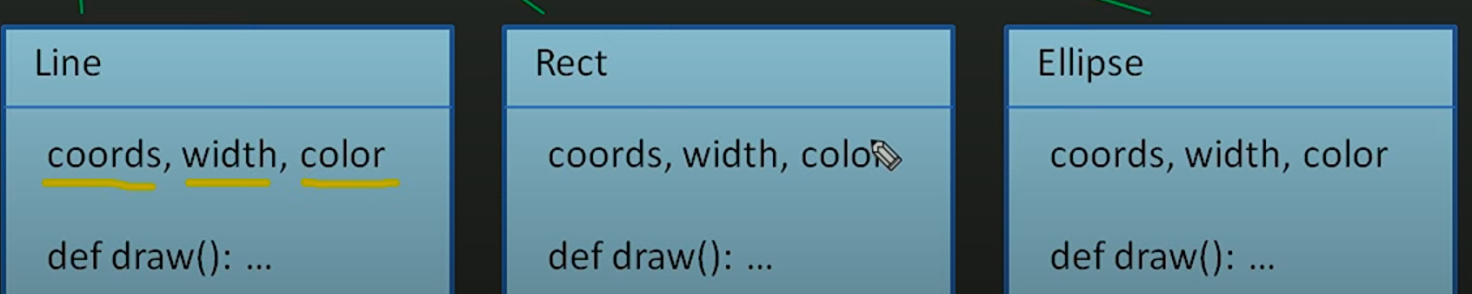
**Примечание 4:**

Благодаря инкапсуляции класс становится единым целым, работа с ним возможна только через разрешенные (публичные) свойства и методы, например в автомобили разрешенные методы: руль, коробка передач, педали газа и тормоза и т.д. А всё остальное скрыто где-то внутри.

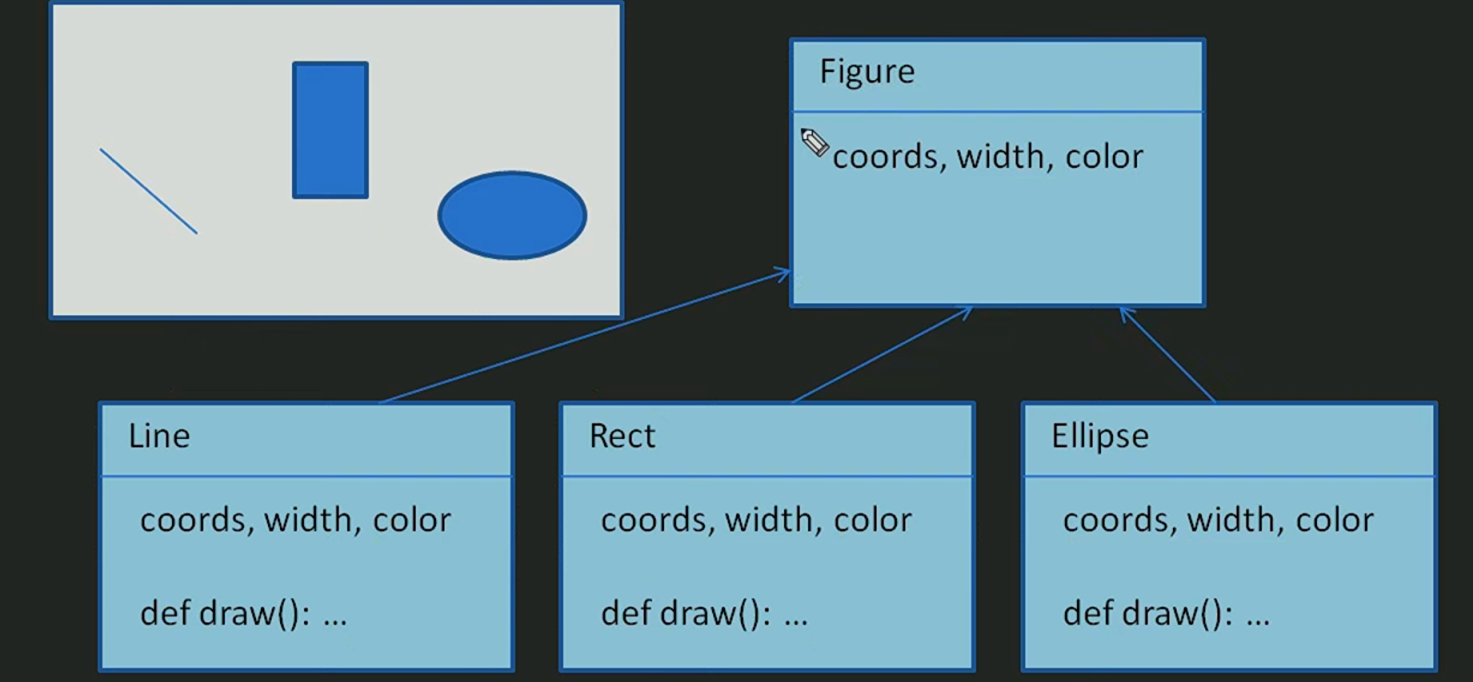
**Наследование**



Наследование позволяет, как бы вынести за скобки общее для разных классов.



Общие свойства – целесообразно вынести в общий для всех них базовый класс:



Чтобы эти атрибуты присутствовали в дочерних классах – мы должны унаследовать дочерние классы от базового класса.

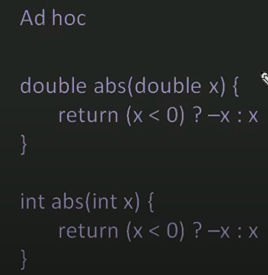
**Примечание 5:**

Благодаря механизму наследования классов – мы можем использовать раннее созданные классы и расширять их функциональность.

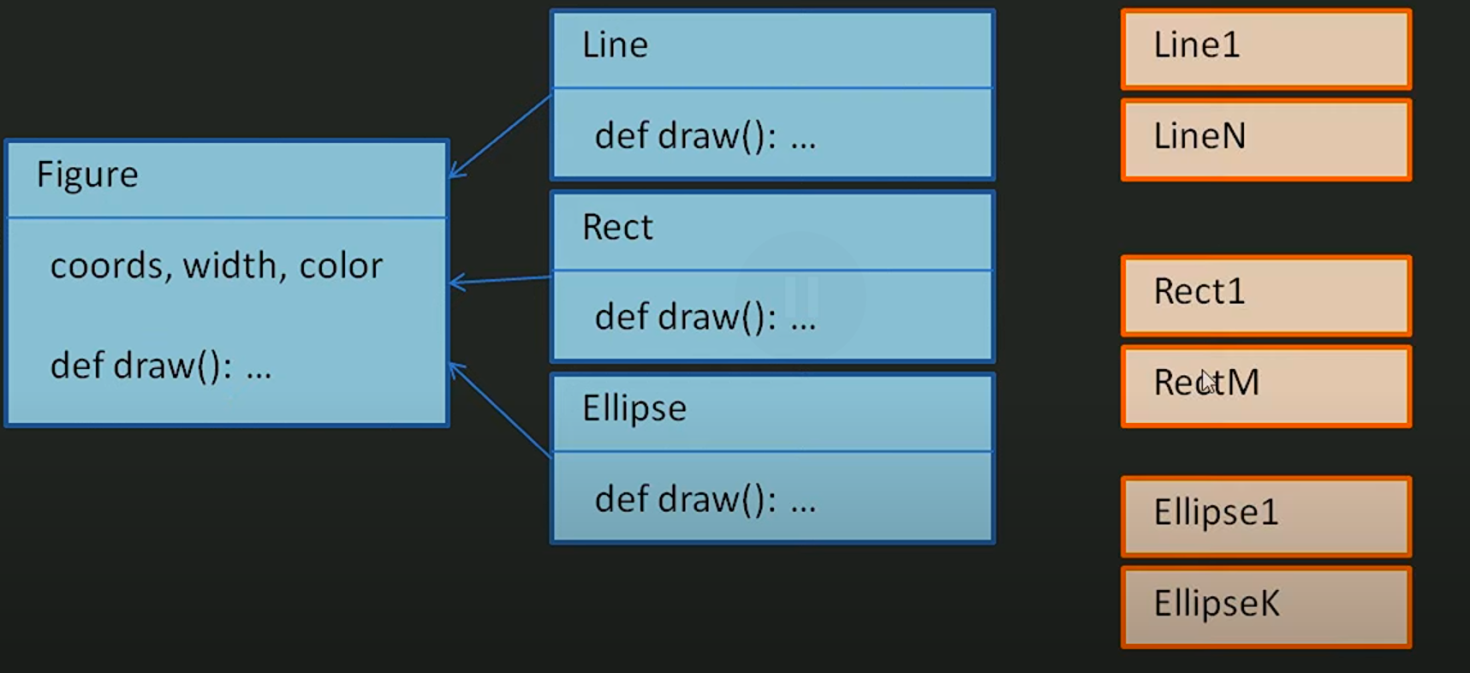
**Полиморфизм**

Возможность через единый интерфейс работать с объектами разных классов.

Полиморфизм разделяется на Ad hoc (не используется в Python) и Параметрический полиморфизм.



**Параметрический полиморфизм**



**Примечание 5:**

Благодаря параметрическому полиморфизму, мы можем оперировать разными типами объектов через их единый базовый класс. В итоге мы имеем общий интерфейс – класс Figure для управления самых разных графических примитивов. Причём в будущем в программу можно будет добавлять новые классы графических примитивов, просто наследовав их от базового класса.

**Примечание 6:**

Благодаря наследованию и полиморфизму мы можем на уровне объектно-ориентированного программирования описывать общую, абстрактную архитектуру работы программы в целом, а потом, создавая дочерние классы, наполнять эту программу конкретным содержимым.

**Концепция ООП**

* Инкапсуляция;
* Наследование;
* Полиморфизм.

**Урок №1: Классы и объекты. Атрибуты классов и объектов.**

Определить класс можно так:

class Name\_class:  
 attribute = value

Где class – ключевое слово, Name\_class – имя класса (по стандарту pep8, имя класса начинается с заглавной буквы, и также имя класса отображает суть его работы), attribute – свойство (атрибут) класса, value – значение свойства (атрибута).

class Point:  
 color = 'red'  
 circle = 2

Фактический класс Point образует пространство имен с именем Point и в этом пространстве имён находится два атрибута (две переменные – color и circle), к которым мы можем обращаться, используя синтаксис для пространства имён.

Прочитать какое либо свойство (атрибут) класса (если позволяет доступ):

Point.color => 'red'

Point.circle => 2

Также мы имеем возможность менять значение свойств (атрибутов) класса (если позволяет доступ):

Point.color = 'black'

Point.color => 'black'

Увидеть все свойства (атрибуты) класса можно с помощью специального встроенного свойства (атрибута) \_\_dict\_\_:

Point.\_\_dict\_\_ =>   
=> mappingproxy({'\_\_module\_\_': '\_\_main\_\_', 'color': 'black', 'circle': 2, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'Point' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'Point' objects>, '\_\_doc\_\_': None, '\_\_annotations\_\_': {}})

**Примечание 1:**

Атрибут \_\_dict\_\_. В Python у объектов есть встроенные специальные атрибуты. Мы их не определяем, но они есть. Одним из таких атрибутов объекта является свойство \_\_dict\_\_. Его значением является словарь, в котором ключи – это имена свойств экземпляра, а значения – текущие значения свойств.

Создать экземпляр (объект) класса можно так:

point1 = Point()  
point2 = Point()

Таким образом, мы создали переменные point1 и point2, которые являются экземплярами (объектами) класса Point. И через эти переменные доступны два свойства (атрибута) – color и circle. Объект point1 и point2 – это совершенно два разных независимых объекта.

Так как переменные point1 и point2 ссылаются на объекты класса Point, то мы это явно можем определить с помощью функции **type(arg)**.

type(point1) => <class '\_\_main\_\_.Point'>

Или:

type(point1) == Point => True

**Примечание 2:**

Функция **type(arg)** – возвращает тип объекта и является собственным метаклассом языка Python.

Также мы можем использовать функцию **isinstance(object, classinfo)**:

isinstance(point1, Point) => True

**Примечание 3:**

Функция **isinstance(object, classinfo)** вернет True , если проверяемый объект object является экземпляром указанного класса (классов) или его подкласса (прямого, косвенного или виртуального). Если объект object не является экземпляром данного типа, то функция всегда возвращает False.

**Примечание 4:**

Объекты point1 и point2 – образуют свои пространства имён экземпляров класса Point и не содержат никаких собственных атрибутов, свойства (атрибуты) – color и circle – берутся непосредственно из класса Point. То есть атрибуты класса – общие для всех его экземпляров.

Если мы изменим значение атрибута circle класса Point:

Point.circle = 1

То значение атрибутов экземпляров point1.circle и point2.circle тоже изменятся с 2 на 1.

Если мы посмотрим на коллекцию \_\_dict\_\_ от point1, то она будет пустой:

point1.\_\_dict\_\_ => {}

point2.\_\_dict\_\_ => {}

point1.color => 'black'

point2.circle => 1

Если же мы выполним такую команду:

point1.color = 'green' => объект point1 атрибут color принимает значение 'green', а объект point2 – остался без изменений.

Тем самым мы создали совершенно новый атрибут color в пространстве имён point1 и передали туда значение 'green'. А объект point2 по-прежнему будет ссылаться на атрибут color из класса Point.

point1.\_\_dict\_\_ => {'color': 'green'}

point2.\_\_dict\_\_ => {}

**Примечание 5:**

Point.color и Point.circle – это атрибуты класса, а point1.color – локальный атрибут экземпляра point1.

По аналогии мы можем динамически создавать новые атрибуты в классе Point:

Point.type\_pt = 'disc' => атрибут type\_pt появился как в классе Point, так и в экземплярах класса (ссылающихся на атрибут type\_pt).

Или мы можем использовать функцию **setattr(object, name, value)**

setattr(Point, 'prop', 1) => атрибут 'prop'\_pt появился как в классе Point, так и в экземплярах класса (ссылающихся на атрибут 'prop').

Если мы пропишем уже существующий атрибут:

setattr(Point, 'type\_pt', 'square') => То, просто поменяем значение 'disc' на 'square' и изменится он как в классе Point, так и в экземплярах класса (ссылающихся на атрибут type\_pt).

**Примечание 6:**

Функция **setattr(object, name, value)** устанавливает значение атрибута указанного объекта по его имени. Аргументами являются объект object, строка с именем name атрибута и произвольное значение value устанавливаемого атрибута.

При обращении к несуществующему атрибуту класса:

Point.a => Приведет к ошибке => AttributeError: type object 'Point' has no attribute 'a' - (Ошибка атрибута: объект типа 'Point' не имеет атрибута 'a')

Этой ошибки можно избежать, если воспользоваться специальной функцией **getattr(object, name, (Это значение будет возвращаться, если данного атрибута нет)):**

getattr(Point, 'a', False) => False

getattr(Point, 'color') => 'black'

**Примечание 7:**

Функция **getattr**(**object, name, (Это значение будет возвращаться если данного атрибута нет))**) возвращает значение атрибута указанного объекта object по его имени name . Имя атрибута name должно быть строкой. Если строка является именем name одного из атрибутов объекта object , результатом является значение этого атрибута.

Удалять любые атрибуты из классов или объектов класса:

del Point.prop => удалит атрибут prop как из класса Point, так и из экземпляров класса (ссылающихся на атрибут prop).

Прежде чем удалять атрибут, неплохо было бы проверить его существование с помощью функции **hasattr(object, name)**:

hasattr(Point, 'prop') => False – так как, мы уже удалили этот атрибут и его нет.

hasattr(Point, 'color') => True – так как, такой атрибут существует.

Или удалять атрибуты можно с помощью функции **delattr(object, name):**

delattr(Point, 'type\_pt') => Удалит атрибут 'type\_pt' из класса Point, так и из экземпляров класса (ссылающихся на атрибут type\_pt).

**Примечание 8:**

Функция **hasattr(object, name)** проверяет существование атрибута с именем name в объекте object . Возвращает True , если атрибут с именем name существует, иначе False .

**Примечание 9:**

Функция **delattr(object, name)** удаляет из объекта указанный атрибут, если объект позволяет это сделать. Строка с именем должна быть именем одного из атрибутов объекта.

**Примечание 10:**

hasattr(point1, circle') => True – Это True не говорит что данный атрибут находится непосредственно в этом пространстве имён point1, оно лишь говорит о том что мы можем в принципе получить через пространство имён point1 доступ к этому атрибуту circle.

point1.\_\_dict\_\_ => {'color': 'green'}

point1.color => 'green'

del point1.color

point1.color => 'black'

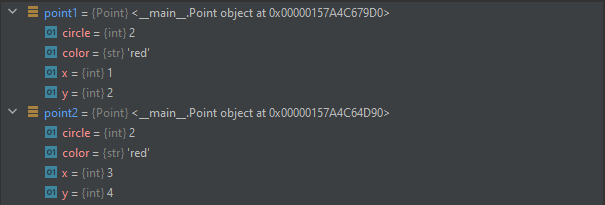
point1.\_\_dict\_\_ => {}

А вот удаление атрибутов происходит непосредственно в текущем пространстве имён.

**После удаления локального атрибута color из point1, этот атрибут стал браться непосредственно из класса Point.**

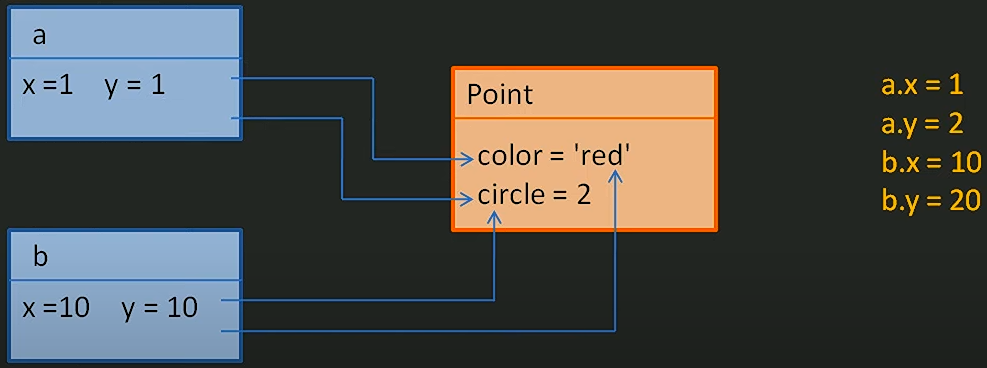
class Point:  
 color = 'red'  
 circle = 2  
  
  
point1 = Point()  
point1.x = 1  
point1.y = 2  
  
point2 = Point()  
point2.x = 3  
point2.y = 4

Образует:



При этом:





**Примечание 11:**

point1.x, point1.y и point2.x, point2.y – являются локальными индивидуальными свойствами для объектов point1 и point2 соответственно.

В любом классе мы можем определять его описание в виде начальной строки с помощью тройных–двойных кавычек """:

class Point:  
 *"""Класс для представления координат точек на плоскости"""* color = 'red'  
 circle = 2

Увидеть описание класса можно с помощью специального атрибута \_\_doc\_\_:

Point.\_\_doc\_\_ => 'Класс для представления координат точек на плоскости'

**Примечание 12:**

Все объекты в Python имеют специальный атрибут \_\_doc\_\_ , предназначенный для хранения строки документации — docstring. Вот как определено понятие docstring в официальной документации: «Docstring — строковый литерал, который встречается как первый оператор в определении модуля, функции, класса или метода.

**Итоги урока №1**

* **getattr(obj, name, [, default])** – возвращает значение атрибута объекта;
* **hasattr(obj, name)** – проверяет наличие атрибута name в obj;
* **setattr(obj, name, value)** – задает значение атрибута (если атрибут не существует, создаётся);
* **delattr(obj, name)** – удаляет атрибут с именем name.
* \_\_doc\_\_ – содержит строку с описание класса;
* \_\_dict\_\_ – содержит набор атрибутов экземпляра класса.

**Урок №2: Методы классов. Параметр self.**

**Методы классов**

Напомню, что класс может содержать как свойства (атрибуты) так и методы (функции). Благодаря этому внутри класса можно реализовывать любые алгоритмы.

**Примечание 1**

Функции принято называть глаголами, так как они выполняют ряд определенных действий, проще говоря, что-то делают. Например: set\_value, get\_param, smart, stop и т.д.

Для примера объявим в нашем классе Point, метод set\_coords:

class Point:  
 color = 'red'  
 circle = 2  
  
 def set\_coords(self):

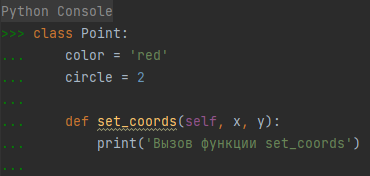
pass

Изначально при написании функции, у нас автоматический появится параметр self. Давайте для наглядности сначала уберём его.

class Point:  
 color = 'red'  
 circle = 2  
  
 def set\_coords():

print('Вызов функции set\_coords')

Давайте посмотрим, как это всё будет работать. Для наглядности я создам этот же класс в Python Console:





При:

Point.set\_coords => <function Point.set\_coords at 0x00000228BDD97250> => Из этой записи, мы понимаем что некий атрибут set\_coords связан с определённой функцией, и раз это так, то мы можем вызвать этот метод:

Point.set\_coords() => Вызов функции set\_coords

На данный момент мы не имеем никаких ошибок, несмотря на то, что мы намеренно удалили автоматически-созданный параметр self. Итак, продолжим.

Создадим экземпляр (объект) класса Point:

pt = Point() =>

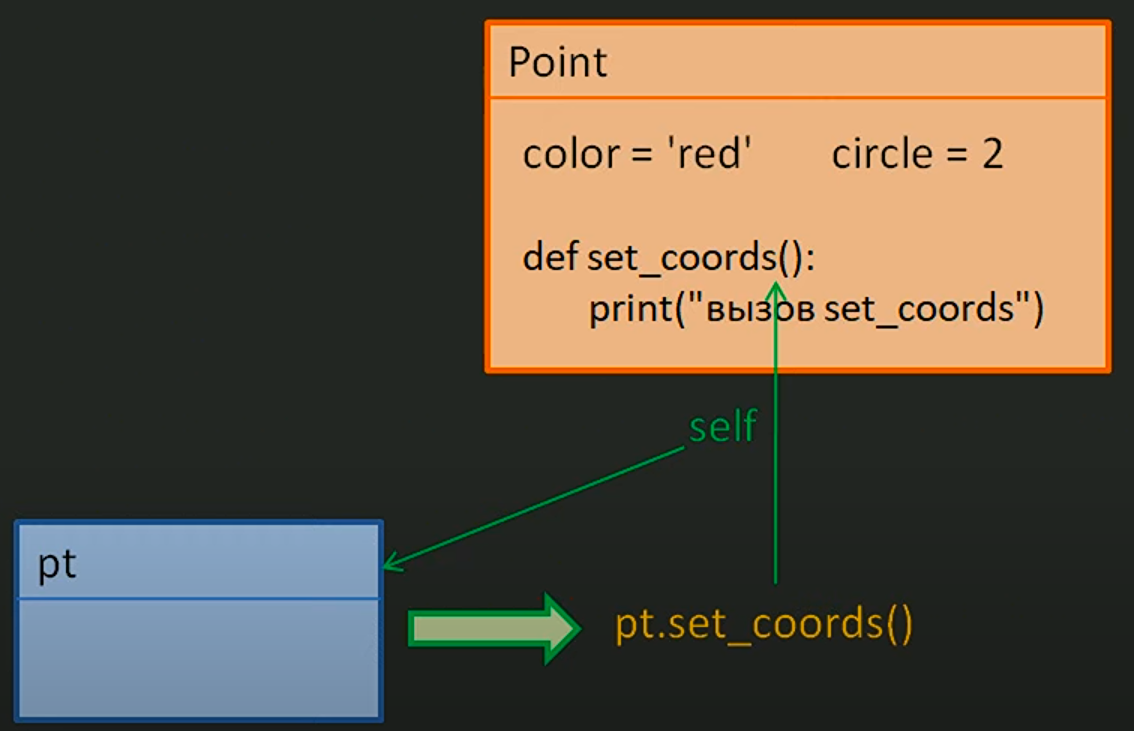


pt.set\_coords => <bound method Point.set\_coords of <\_\_main\_\_.Point object at 0x00000228BDCD50C0>> => Такой атрибут существует, никакой ошибки нет.

Давайте теперь, попробуем вызвать нашу функцию через объект:

pt.set\_coords() => Получаем ошибку => TypeError: Point.set\_coords() takes 0 positional arguments but 1 was given (Ошибка типа: Point.set\_coords() принимает 0 позиционных аргументов, но был задан 1)

**Но спрашивается, в каком месте я задал один атрибут? Но всё дело в том, что интерпретатор Python, когда вызывается тот или иной метод, через объект класса делает следущее:**



Когда мы выполняем команду: pt.set\_coords() => то на самом деле интерпретатор языка Python автоматический в качестве первого аргумента подставляет параметр self => Он является ссылкой на экземпляр класса Point, то есть в нашем случае, на объект pt. И так происходит абсолютно всегда для любого объекта класса, если мы объявляем методы как функции внутри класса.

**Примечание 2**

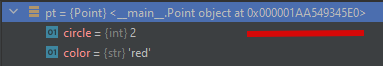
Параметр self – представляет собой ссылку на экземпляр (объект) класса.

Поэтому если мы внутри класса хотим объявить какой либо метод, чтобы мы потом могли вызвать его через объект класса, то в качестве первого аргумента нужно обязательно указать параметр self, то есть он будет всегда ссылаться на тот экземпляр из которого был вызван данный метод. То есть через этот параметр, мы всегда будем знать с каким объектом работает метод set\_coords.

class Point:  
 color = 'red'  
 circle = 2  
  
 def set\_coords(self):  
 print('Вызов функции set\_coords ' + str(self))

pt = Point()

pt.set\_coords() => Вызов функции set\_coords <\_\_main\_\_.Point object at 0x000001AA549345E0> => И теперь никаких ошибок нету.



=> Адреса совпадают.

Правда, теперь если мы захотим вызвать этот же метод через класс Point:

Point.set\_coords() => Приведёт к ошибке => TypeError: Point.set\_coords() missing 1 required positional argument: 'self' (Ошибка типа: Point.set\_coords() отсутствует 1 требуемый позиционный аргумент: 'self') => так как мы не передаём ссылки на экземпляр.

Но, чтобы исправить эту ошибку мы можем сделать так:

Point.set\_coords(pt) => Вызов функции set\_coords <\_\_main\_\_.Point object at 0x000001AA549345E0> => Явно указав параметр.

Для чего-же по итогу нужен self?

Предположим что мы хотим чтобы метод set\_coords в любой из объектов класса добавлял координаты x, y, как раз для таких целей мы можем использовать self:

class Point:  
 color = 'red'  
 circle = 2  
  
 def set\_coords(self, x, y):  
 self.x = x  
 self.y = y

pt1 = Point()

pt1.set\_coords(2, 4)

print(pt1.\_\_dict\_\_)

Получим:

{'x': 2, 'y': 4}

**Примечание 3**

Вот для этого и нужен параметр self, чтобы мы могли работать с локальными атрибутами конкретного экземпляра класса.

Когда у нас в программе создаётся много экземпляров одного и того же класса то метод set\_coords (метод класса) не копируется в отдельные экземпляры, он как был в классе Point, так в этом классе и останется, и только благодаря параметру self мы знаем с каким экземпляром класса в данный момент работает метод set\_coords.

И также через него вполне можно создавать или менять уже существующие локальные свойства экземпляра класса.

Для примера напишем следующий код:

class Point:  
 color = 'red'  
 circle = 2  
  
 def set\_coords(self, x, y):  
 self.x = x  
 self.y = y

pt1 = Point()  
pt1.set\_coords(2, 4)  
print(pt1.\_\_dict\_\_)  
  
pt2 = Point()  
pt2.set\_coords(8, 16)  
print(pt2.\_\_dict\_\_)

Результат будет:

{'x': 2, 'y': 4}

{'x': 8, 'y': 16}

Для следующего примера напишем следующий код:

class Point:  
 color = 'red'  
 circle = 2  
  
 def set\_coords(self, x, y):  
 self.x = x  
 self.y = y  
  
 def get\_coords(self):  
 return self.x, self.y  
  
   
pt1 = Point()  
pt1.set\_coords(2, 4)  
print(pt1.get\_coords())  
pt2 = Point()  
pt2.set\_coords(8, 16)  
print(pt2.get\_coords())

Результат будет:

(2, 4)

(8, 16)

**Грубо говоря, можно сказать что все методы, написанные внутри класса это тоже атрибуты класса, просто они ведут не на данные а на функции**. И так как это атрибуты класса, мы можем получить к ним доступ через метод getattr(). Следующий код:

class Point:  
 color = 'red'  
 circle = 2  
  
 def set\_coords(self, x, y):  
 self.x = x  
 self.y = y  
  
 def get\_coords(self):  
 return self.x, self.y  
  
  
pt1 = Point()  
pt1.set\_coords(2, 4)  
f1 = getattr(pt1, 'get\_coords')  
print(f1)  
print(f1())  
  
pt2 = Point()  
pt2.set\_coords(8, 16)  
f2 = getattr(pt2, 'get\_coords')  
print(f2)  
print(f2())

Выведет:

<bound method Point.get\_coords of <\_\_main\_\_.Point object at 0x000001E652083FD0>>

(2, 4)

<bound method Point.get\_coords of <\_\_main\_\_.Point object at 0x000001E652083EB0>>

(8, 16)

**Урок №3: Инициализатор \_\_init\_\_ и финализатор \_\_del\_\_**

\_\_Магический\_метод\_\_ => Начинается с \_\_ и заканчивается \_\_.

\_\_init\_\_ (self) => инициализатор объекта класса.

Вызывается сразу после создания экземпляра класса.

class Point:  
 color = 'red'  
 circle = 2  
  
 def set\_coords(self, x, y):  
 self.x = x  
 self.y = y  
  
 def get\_coords(self):  
 return self.x, self.y

Использовать данный класс на практике не очень удобно. Для начала создадим экземпляр класса:

pt = Point()

А далее чтобы в этом объекте появились локальные свойства x, y – мы по идее должны использовать метод set\_coords:

pt.set\_coords(1, 2)

print(pt.\_\_dict\_\_) => {'x': 1, 'y': 2}

Однако, было бы хорошо сделать эти действия в момент создания экземпляра класса. И в этом нам как раз поможет магический метод – \_\_init\_\_ (self). Добавим следующий код:

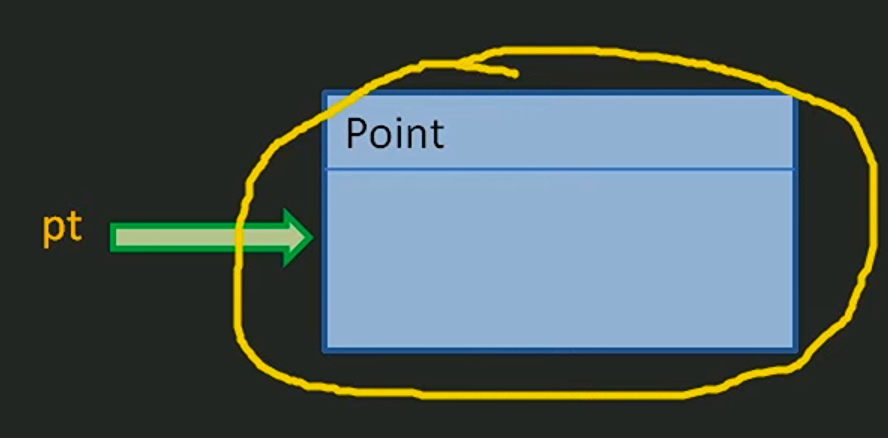
class Point:  
 color = 'red'  
 circle = 2  
  
 def \_\_init\_\_(self):

print('Вызов \_\_init\_\_')  
 self.x = 2  
 self.y = 3  
  
 def set\_coords(self, x, y):  
 self.x = x  
 self.y = y  
  
 def get\_coords(self):  
 return self.x, self.y

pt = Point() => Вызов \_\_init\_\_

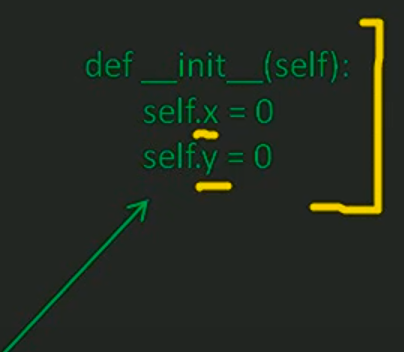
print(pt.\_\_dict\_\_) => {'x': 2, 'y': 3}

Детальнее это работает следующим образом:



В начале происходит создание объекта в памяти устройства и непосредственно перед его созданием вызывается магический метод \_\_new\_\_.

Затем, после успешного создания объекта вызывается следующий магический метод \_\_init\_\_ и в нашем случае он создаёт два локальных свойства x и y:



В результате в новом созданном объекте появляются свойства x, y со значениями 0.

Улучшить этот код можно так:

class Point:  
 color = 'red'  
 circle = 2  
  
 def \_\_init\_\_(self, a, b):  
 self.x = a  
 self.y = b  
  
 def set\_coords(self, x, y):  
 self.x = x  
 self.y = y  
  
 def get\_coords(self):  
 return self.x, self.y

pt = Point(5, 6)

print(pt.\_\_dict\_\_) => {'x': 5, 'y': 6}

Такая запись, приведёт к ошибке:

pt = Point() => TypeError: Point.\_\_init\_\_() missing 2 required positional arguments: 'a' and 'b' (Ошибка типа: Точка.\_\_init\_\_() отсутствуют 2 обязательных позиционных аргумента: 'a' и 'b')

Улучшить читаемость кода и исправить ошибку можно следующим образом:

Поправим функцию \_\_init\_\_:

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

1. По идее когда создают инициализаторы, то параметры называют также как и создаваемые локальные свойства.

2. Как и в обычной функции мы можем прописывать как фактические параметры, так и формальные, то есть указать значение по умолчанию.

В таком случае мы имеем право сделать так:

pt = Point() => Не приведёт к ошибке.

И соответственно значение локальных атрибутов будет = 0:

print(pt.\_\_dict\_\_) => {'x': 0, 'y': 0}

\_\_del\_\_(self) => финализатор класса.

Вызывается перед удалением/уничтожением экземпляра класса.

Давайте добавим и этот магический метод в наш код:

class Point:  
 color = 'red'  
 circle = 2  
  
 def \_\_init\_\_(self, x=0, y=0):  
 self.x = x  
 self.y = y  
  
 def \_\_del\_\_(self):  
 print('Удаление экземпляра:', str(self))  
  
 def set\_coords(self, x, y):  
 self.x = x  
 self.y = y  
  
 def get\_coords(self):  
 return self.x, self.y

pt = Point()

print(pt.\_\_dict\_\_) => {'x': 0, 'y': 0}

=> Удаление экземпляра: <\_\_main\_\_.Point object at 0x000001EFB12F5ED0>

Как только программа завершилась, наш объект был удалён. Но возникает вопрос, когда и в какой момент вообще происходит удаление объекта? Работает всё достаточно просто – интерпретатор языка Python – имеет так называемый сборщик мусора. Это алгоритм который отслеживает объекты, и как только они становятся не нужными – удаляет их. Но как он определяет нужный объект или ненужный? Пока на какой либо объект ведёт хотя бы одна внешняя ссылка он считается нужным. Но как только пропадают все внешние ссылки, сборщик мусора решит что этот объект уже не нужен и автоматически его удалит, то есть освободит память которую он занимал.

И непосредственно перед удалением объекта будет вызван финализатор \_\_del\_\_ (магический метод \_\_del\_\_), после вызова \_\_del\_\_ объект/ы гарантированно перестанет/ут существовать.

**Урок №4: Магический метод \_\_new\_\_. Пример паттерна Singleton**

**Урок №5: Методы класса (classmethod) и статические методы (staticmethod)**