# **Режимы доступа public, private, protected. Сеттеры и геттеры**

[Смотреть материал на видео](https://www.youtube.com/watch?v=RipfqbH0eqY&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

На прошлых занятиях мы научились с вами создавать экземпляры классов и объявлять в них атрибуты и методы. Пришла пора сделать следующий шаг и познакомиться с механизмом ограничения доступа к данным и методам класса извне. Это основа механизма **инкапсуляции**.

Давайте предположим, что мы описываем класс представления точки на плоскости:

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

Когда создается экземпляр этого класса:

pt = Point(1, 2)

то имеется полный доступ ко всем его локальным атрибутам:

**print**(pt.x, pt.y)

а, значит, их всегда можно изменить через ссылку pt:

pt.x = 200 pt.y = "coord\_y"

и присвоить любые значения, в том числе и недопустимые (например, строку).

Так вот, чтобы указать программисту (пользователю класса), чтобы он не обращался напрямую к атрибутам, их следует делать «закрытыми». Что это такое? Смотрите, в Python возможны следующие варианты доступа к данным:

* attribute (без одного или двух подчеркиваний вначале) – публичное свойство (**public**);
* \_attribute (с одним подчеркиванием) – режим доступа **protected** (служит для обращения внутри класса и во всех его дочерних классах)
* \_\_attribute (с двумя подчеркиваниями) – режим доступа **private** (служит для обращения только внутри класса).

Давайте разберем это подробнее. До сих пор все атрибуты в классе, либо в экземплярах класса мы делали публичными, то есть, не использовали одно или два подчеркивания перед их именами. Давайте посмотрим, что изменится, если добавить одно подчеркивание перед локальными атрибутами x и y:

**class** Point:     **def** \_\_init\_\_(self, x=0, y=0):         self.\_x = x         self.\_y = y

Так реализуется режим protected в Python. Если кто из вас программирует на других языках, например, С++ или Java, то сейчас ожидают, что мы не сможем обращаться к свойствам \_x и \_y через ссылку pt, так как они определены как защищенные (protected). Давайте проверим и попробуем вывести их в консоль:

**print**(pt.\_x, pt.\_y)

Как видим, никаких ошибок не возникает и все работает так, словно это публичные свойства экземпляра класса. 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 класса и такое свойство может перестать существовать, т.к. никто не предполагал доступа к нему извне. Так что, к таким атрибутам лучше не обращаться напрямую – одно нижнее подчеркивание указывает нам, что это внутренние, служебные переменные.

Давайте теперь посмотрим, как работает режим доступа private. Пропишем у локальных свойств два подчеркивания:

**class** Point:     **def** \_\_init\_\_(self, x=0, y=0):         self.\_\_x = x         self.\_\_y = y

и также попробуем обратиться к ним напрямую:

**print**(pt.\_\_x, pt.\_\_y)

После запуска программы видим ошибку, что такие свойства не определены. Это говорит о том, что извне, через переменную pt мы не можем напрямую к ним обращаться. А вот внутри класса доступ к ним открыт.

Пропишем метод set\_coord, который будет менять локальные свойства \_\_x и \_\_y экземпляра класса:

**def** set\_coord(self, x, y):         self.\_\_x = x         self.\_\_y = y

А ниже, вызовем его для экземпляра pt:

pt.set\_coord(1, 2)

Как видите, никаких ошибок не возникает и чтобы убедиться в изменении локальных приватных свойств, определим еще один метод:

**def** get\_сoord(self):         **return** self.\_\_x, self.\_\_y

И вызовем его:

**print**(pt.get\_сoord())

После запуска программы видим измененные координаты точки. В результате, мы с вами определили два вспомогательных метода: set\_coord и get\_coord, через которые предполагается работа с защищенными данными класса. Такие методы в ООП называются **сеттерами**и**геттерами** или еще **интерфейсными методами**.

Зачем понадобилось в классах создавать приватные атрибуты да еще и определять дополнительно методы для работы с ними извне. Я об этом уже говорил на самом первом занятии по ООП, когда объяснял принцип инкапсуляции. Но, скажу еще раз. Класс в ООП следует воспринимать как некое единое целое, и чтобы случайно или намеренно не нарушить целостность работы алгоритма внутри этого класса, то следует взаимодействовать с ним только через публичные свойства и методы. В этом суть принципа инкапсуляции. Опять же, представьте автомобиль, в котором согласованно работают тысячи узлов. А управление им предполагается только через разрешенные интерфейсы: руль, коробка передач, педали газа и тормоза и т.п. Если во время движения вмешиваться напрямую в его узлы, например, будем на ходу спускать воздух из шин, то, наверное, ничего хорошего не получится. То же самое, можно сказать и о программисте, который намеренно обходит запрет и обращается к скрытым атрибутам класса напрямую, а не через сеттеры или геттеры. Так делать не нужно.

Назначение интерфейсных методов 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 type и записать сеттер следующим образом:

**def** set\_coord(self, x, y):         **if** type(x) **in** (int, float) **and** type(y) **in** (int, float):             self.\_\_x = x             self.\_\_y = y         **else**:             **raise** ValueError("Координаты должны быть числами")

Здесь мы проверяем, что обе переданные переменные x и y должны иметь тип int или float и только после этого приватным атрибутам экземпляра класса присваиваются новые значения. Иначе, генерируется исключение ValueError. Об исключениях мы с вами еще будем говорить.

Теперь, если передавать недопустимые значения координат:

pt.set\_coord('1', 2)

то увидим ошибку ValueError.

Продолжим совершенствовать наш класс Point и добавим приватный метод для проверки корректности координат. Приватный метод объявляется также как и приватная переменная – двумя подчеркиваниями и, кроме того, сделаем его методом уровня класса (о декораторе classmethod мы с вами говорили на предыдущем занятии):

    @classmethod     **def** \_\_check\_value (cls, x):         **return** type(x) **in** (int, float)

Соответственно, в сеттере и в инициализаторе воспользуемся этим методом:

**def** \_\_init\_\_(self, x=0, y=0):         self.\_\_x = self.\_\_y = 0           **if** self.\_\_check\_value (x) **and** self.\_\_check\_value (y):             self.\_\_x = x             self.\_\_y = y       **def** set\_coord(self, x, y):         **if** self.\_\_check\_value (x) **and** self.\_\_check\_value (y):             self.\_\_x = x             self.\_\_y = y         **else**:             **raise** ValueError("Координаты должны быть числами")

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

На самом деле, в Python можно относительно легко обратиться и к приватным атрибутам извне. Если распечатать все атрибуты экземпляра:

**print**(dir(pt))

то среди прочих мы увидим, следующие:

'\_Point\_\_x', '\_Point\_\_y'

Это и есть кодовые имена приватных атрибутов, к которым мы можем обратиться через ссылку pt:

**print**(pt.\_Point\_\_x, pt.\_Point\_\_y)

и менять их. Однако, так делать крайне не рекомендуется и двойное подчеркивание должно сигнализировать программисту, что работать с такими атрибутами нужно только через разрешенные интерфейсные методы. Иначе, возможны непредвиденные ошибки.

Если у вас появилась необходимость лучше защитить методы класса от доступа извне, то это можно сделать с помощью модуля accessify. Для его установки нужно выполнить команду:

pip install accessify

И, затем, импортировать из него два декоратора:

**from** accessify **import** private, protected

Далее, нужный декоратор просто применяем к методу и он становится либо приватным (private), либо защищенным (protected):

    @private     @classmethod     **def** check\_value(cls, x):         **return** type(x) **in** (int, float)

Все, теперь мы можем обратиться к check\_value только внутри класса, но не извне:

pt.check\_value(5)

Я, думаю, из этого занятия вам стало понятно, как реализуются режимы доступа public, protected и private, а также, как правильно обращаться к скрытым атрибутам через интерфейсные методы – сеттеры и геттеры.