# **Магические методы \_\_setattr\_\_, \_\_getattribute\_\_, \_\_getattr\_\_ и \_\_delattr\_\_**

[Смотреть материал на видео](https://www.youtube.com/watch?v=CAx-NLFc-Z4&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

На этом занятии мы поговорим о работе с атрибутами класса и его экземплярами. Я напомню, что класс можно воспринимать как некое пространство имен, в котором записаны свойства и методы. Например, если вернуться к классу Point (представления точки на плоскости):

**class** Point:     MAX\_COORD = 100     MIN\_COORD = 0       **def** \_\_init\_\_(self, x, y):         self.x = x         self.y = y       **def** set\_coord(self, x, y):         self.x = x         self.y = y

то здесь мы видим определение четырех атрибутов: двух свойств MAX\_COORD и MIN\_COORD и двух методов \_\_init\_\_ и set\_coord. Это атрибуты класса и при создании экземпляров:

pt1 = Point(1, 2) pt2 = Point(10, 20)

Эти атрибуты остаются в пространстве имен класса, не копируются в экземпляры. Но из экземпляров мы можем совершенно спокойно к ним обращаться, так как пространство имен объектов содержит ссылку на внешнее пространство имен класса. Если какой-либо атрибут не существует в экземпляре, то поиск переходит во внешнее пространство, то есть, в класс и поиск продолжается там. Поэтому мы совершенно спокойно можем через экземпляр обратиться к свойству класса MAX\_COORD:

**print**(pt1.MAX\_COORD)

И получается, что атрибуты и методы класса – это общие данные для всех его экземпляров.

Далее, когда мы обращаемся к атрибутам класса внутри методов, объявленных в этом классе, то должны не просто прописать их имена:

**def** set\_coord(self, x, y):         **if** MIN\_COORD <= x <= MAX\_COORD:             self.x = x             self.y = y

а явно указать перед ними ссылку на класс, то есть, на пространство имен. Либо так:

**if** Point.MIN\_COORD <= x <= Point.MAX\_COORD:

но лучше через self:

**if** self.MIN\_COORD <= x <= self.MAX\_COORD:

Здесь self – это ссылка на экземпляр класса, из которого метод вызывается, поэтому мы можем через этот параметр обращаться к атрибутам класса.

Обо всем этом мы с вами уже говорили, я лишь еще раз повторил эти важные моменты. А теперь один нюанс, о который спотыкаются многие начинающие программисты. Давайте предположим, что нам нужен метод, который бы изменял значение атрибута класса MIN\_COORD. Пропишем его как обычный метод:

**def** set\_bound(self, left):         self.MIN\_COORD = left

Иногда ошибочно здесь рассуждают так. Мы обращаемся к атрибуту класса MIN\_COORD и присваиваем ему новое значение left. Те из вас, кто внимательно смотрел предыдущие занятия, понимают, в чем ошибочность такого рассуждения. Да, когда мы через self (ссылку на объект) записываем имя атрибута и присваиваем ему какое-либо значение, то оператор присваивания создает этот атрибут в локальной области видимости, то есть, в самом объекте. В результате, у нас появляется новое локальное свойство в экземпляре класса:

pt1.set\_bound(-100)**print**(pt1.\_\_dict\_\_)

А в самом классе одноименный атрибут остается без изменений:

**print**(Point.\_\_dict\_\_)

Поэтому, правильнее было бы здесь объявить метод уровня класса и через него менять значения атрибутов MIN\_COORD и MAX\_COORD:

    @classmethod     **def** set\_bound(cls, left):         cls.MIN\_COORD = left

Тогда в самом объекте не будет создаваться никаких дополнительных свойств, а в классе изменится значение переменной MIN\_COORD, так, как мы этого и хотели.

Будем полагать, что мы теперь с вами хорошо понимаем, как обращаться к атрибутам класса и его объектов. Во второй части занятия я хочу вам рассказать о четырех магических методах, которые используются при работе с атрибутами:

* \_\_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).

Работают они достаточно просто. Начнем с метода \_\_getattribute\_\_ и с его помощью ограничим доступ к приватным свойствам \_\_x и \_\_y экземпляра. Для простоты я переопределю класс Point, следующим образом:

**class** Point:     MAX\_COORD = 100     MIN\_COORD = 0       **def** \_\_init\_\_(self, x, y):         self.\_\_x = x         self.\_\_y = y       **def** \_\_getattribute\_\_(self, item):         **print**("\_\_getattribute\_\_")         **return** object.\_\_getattribute\_\_(self, item)

Здесь добавлен новый магический метод \_\_getattribute\_\_. Он автоматически вызывается, когда идет считывание атрибута через экземпляр класса. Например, при обращении к свойству MIN\_COORD:

**print**(pt1.MIN\_COORD)

Или к приватному свойству через специальное имя:

**print**(pt1.\_Point\_\_x)

Но раз это так, то давайте явно запретим считывать такой атрибут из экземпляра класса. Для этого пропишем в методе \_\_getattribute\_\_ проверку:

**def** \_\_getattribute\_\_(self, item):         **if** item == "\_Point\_\_x":             **raise** ValueError("Private attribute")         **else**:             **return** object.\_\_getattribute\_\_(self, item)

То есть, мы смотрим, если идет обращение к приватному атрибуту по внешнему имени \_Point\_\_x, то генерируем исключение ValueError. И, действительно, после запуска программы видим отображение этой ошибки в консоли. Вот так, через магический метод \_\_getattribute\_\_ можно реализовывать определенную логику при обращении к атрибутам через экземпляр класса.

Следующий магический метод \_\_setattr\_\_ автоматически вызывается в момент присваивания атрибуту нового значения. Пропишем формально этот метод в классе Point:

**def** \_\_setattr\_\_(self, key, value):         **print**("\_\_setattr\_\_")         object.\_\_setattr\_\_(self, key, value)

После запуска видим несколько сообщений «\_\_setattr\_\_». Это связано с тем, что в момент создания экземпляров класса в инициализаторе \_\_init\_\_ создавались локальные свойства \_\_x и \_\_y. В этот момент вызывался данный метод. Также в переопределенном методе \_\_setattr\_\_ мы должны вызывать соответствующий метод из базового класса object, иначе, локальные свойства в экземплярах создаваться не будут.

Давайте теперь для примера через этот магический метод запретим создание локального свойства с именем z. Сделаем это следующим образом:

**def** \_\_setattr\_\_(self, key, value):         **if** key == 'z':             **raise** AttributeError("недопустимое имя атрибута")         **else**:             object.\_\_setattr\_\_(self, key, value)

Обратите внимание, что внутри метода \_\_setattr\_\_ нельзя менять свойства напрямую:

**def** \_\_setattr\_\_(self, key, value):         **if** key == 'z':             **raise** AttributeError("недопустимое имя атрибута")         **else**:             self.\_\_x = value

В этом случае метод \_\_setattr\_\_ начнет выполняться по рекурсии, пока не возникнет ошибка достижения максимальной глубины рекурсии. Если нужно сделать что-то подобное, то используйте коллекцию \_\_dict\_\_:

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

или, если требуется стандартное поведение метода, то вызывайте его из класса object, как это мы прописывали вначале:

object.\_\_setattr\_\_(self, key, value)

Следующий магический метод \_\_getattr\_\_ автоматически вызывается, если идет обращение к несуществующему атрибуту. Добавим его в наш класс:

**def** \_\_getattr\_\_(self, item):         **print**("\_\_getattr\_\_: " + item)

Если ниже обратиться к несуществующему свойству, например:

**print**(pt1.a)

то увидим сообщение «\_\_getattr\_\_: a» и значение None, которое вернул данный метод. Если же прописать существующий атрибут:

**print**(pt1.MAX\_COORD)

то этот магический метод уже не вызывается. Зачем он может понадобиться? Например, нам необходимо определить класс, в котором при обращении к несуществующим атрибутам возвращается значение False, а не генерируется исключение. Для этого записывается метод \_\_getattr\_\_ в виде:

**def** \_\_getattr\_\_(self, item):         **return** False

Наконец, последний магический метод \_\_delattr\_\_ вызывается в момент удаления какого-либо атрибута из экземпляра класса:

**def** \_\_delattr\_\_(self, item):         **print**("\_\_delattr\_\_: "+item)

Добавим новое локальное свойство в экземпляр pt1:

pt1.a = 10

затем выполним команду его удаления:

**del** pt1.a

и видим, что действительно был вызван метод \_\_delattr\_\_, правда, сам атрибут удален не был:

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

Это из-за того, что внутри этого метода нужно вызвать соответствующий метод класса object, который и выполняет непосредственное удаление:

**def** \_\_delattr\_\_(self, item):         object.\_\_delattr\_\_(self, item)

Я думаю, что из этого занятия вы хорошо себе уяснили, как происходит обращение к атрибутам класса и как можно управлять доступом к ним через магические методы.