# **Дескрипторы (data descriptor и non-data descriptor)**

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

На этом занятии мы затронем довольно интересную, но не простую тему – дескрипторов классов. Я начну с одного явного недостатка объектов-свойств, о которых мы с вами уже подробно говорили. Давайте представим, что создаем класс для представления точек в трехмерном пространстве:

**class** Point3D:     **def** \_\_init\_\_(self, x, y, z):         self.\_x = x         self.\_y = y         self.\_z = z

Здесь у нас формируются защищенные локальные свойства для создаваемого объекта класса Point3D. Теперь представим, что согласно заданию координаты должны представляться исключительно целыми числами. Для этого я пропишу следующий метод проверки:

    @classmethod     **def** verify\_coord(cls, coord):         **if** type(coord) != int:             **raise** TypeError("Координата должна быть целым числом")

А вызывать его будут в сеттерах соответствующих свойств класса:

    @property     **def** x(self):         **return** self.\_x       @x.setter     **def** x(self, coord):         self.verify\_coord(coord)         self.\_x = coord       @property     **def** y(self):         **return** self.\_y       @y.setter     **def** y(self, coord):         self.verify\_coord(coord)         self.\_y = coord       @property     **def** z(self):         **return** self.\_z       @z.setter     **def** z(self, coord):         self.verify\_coord(coord)         self.\_z = coord

Здесь все вам должно быть понятно. И теперь мы можем в инициализаторе использовать эти объекты-свойства для формирования локальных атрибутов экземпляров:

**class** Point3D:     **def** \_\_init\_\_(self, x, y, z):         self.x = x         self.y = y         self.z = z

Все, создавая объекты этого класса:

p = Point3D(1, 2, 3)**print**(p.\_\_dict\_\_)

у нас автоматически будут формироваться нужные локальные атрибуты и мы сможем с ними работать через объекты-свойства x, y, z.

Но, смотрите, в нашем классе Point3D получилось своеобразное дублирование: мы три раза прописывали свойства, фактически, с одинаковым функционалом. Менялись только названия методов и локальных атрибутов. Представьте, во что превратится описание этого класса, если нужно будет задать 10 и более таких объектов-свойств! Программист во всем этом просто запутается, да и редактировать такую программу станет непросто. Как можно все это оптимизировать? Здесь нам на помощь как раз и приходят дескрипторы.

Вначале, что вообще такое дескрипторы? Это класс, который содержит или один магический метод \_\_get\_\_:

**class** A:     **def** \_\_get\_\_(self, instance, owner):         **return** ...

Или класс, в котором дополнительно прописаны методы \_\_set\_\_ и/или \_\_del\_\_:

**class** B:     **def** \_\_get\_\_(self, instance, owner):         **return** ...       **def** \_\_set\_\_(self, instance, value):         ...       **def** \_\_del\_\_(self):         ...

Первый (класс A) называется non-data descriptor (дескриптор не данных), а второй (класс B) – data descriptor (дескриптор данных). Это различие имеет смысл, но об этом позже.

Как вы уже догадались, эти магические методы, по сути, геттеры и сеттеры, а также делитер. Давайте, теперь посмотрим, как дескриптор может упростить наш программный код с обработкой координат точек.

Вначале я покажу все взаимодействие на схеме. Так как все координаты – целые числа, то интерфейс взаимодействия с ними мы определим через дескриптор с названием Integer (это имя мы, конечно же, придумываем сами):

**class** Integer:     **def** \_\_set\_name\_\_(self, owner, name):         self.name = "\_" + name       **def** \_\_get\_\_(self, instance, owner):         **return** instance.\_\_dict\_\_[self.name]       **def** \_\_set\_\_(self, instance, value):         **print**(f"\_\_set\_\_: {self.name} = {value}")         instance.\_\_dict\_\_[self.name] = value

Пока не обращайте внимания на его содержимое, сейчас я все подробно объясню. Затем, в классе Point3D мы создадим три атрибута как объекты класса Integer:

**class** Point3D:     x = Integer()     y = Integer()     z = Integer()       **def** \_\_init\_\_(self, x, y, z):         self.x = x         self.y = y         self.z = z

Эти атрибуты и есть дескрипторы данных, через которые будет проходить взаимодействие. Итак, когда мы создавали экземпляры классов Integer, то автоматически вызывался магический метод \_\_set\_name\_\_, в котором параметр self являлся ссылкой на создаваемый экземпляр класса; owner – ссылка на класс Point3D; name – имя атрибута (для первого объекта x, затем, y и z). В этом методе мы формируем локальное свойство с именем атрибута, добавляя перед ним одно нижнее подчеркивание (так принято делать при определении дескрипторов). В итоге, в экземплярах классов будут храниться имена \_x, \_y, \_z.

Зачем нам это нужно? Смотрите дальше. Предположим, мы создаем экземпляр класса Point3D:

pt = Point3D(1, 2, 3)

Сработает инициализатор, а в нем идет обращение к дескрипторам x, y, z. В частности, мы им присваиваем переданные значения. В этом случае, в классе Integer срабатывает сеттер (магический метод \_\_set\_\_), параметр self – это ссылка на объект дескриптора; instance – ссылка на объект pt, из которого произошло обращение к дескриптору; value – присваиваемое значение. В этом сеттере мы выводим в консоль сообщение, что был вызван данный метод и отображаем сохраненное имя и присваиваемое значение. Следующей строчкой через ссылку instance, то есть, на экземпляр класса pt, формируем в нем локальное свойство с именем self.name и присваиваем значение value. В результате, в объекте pt появляются локальные свойства \_x, \_y, \_z с соответствующими значениями.

Если затем, выполнить считывание данных через дескриптор, например, x, то автоматически сработает геттер (метод \_\_get\_\_), в котором self – это ссылка на объект Integer; instance – ссылка на экземпляр класса pt; owner – ссылка на класс Point3D. Мы здесь через ссылку instance обращаемся к словарю \_\_dict\_\_ и считываем значение нужного локального свойства, которое, затем, возвращается геттером. Это же значение автоматически возвращается и самим дескриптором.

Вот общая схема работы дескрипторов применительно к нашему классу Point3D. Теперь, сколько бы интерфейсов взаимодействия нам не понадобилось, мы легко их можем добавить в наш класс и все будет выглядеть понятно и компактно. На первый взгляд все это может показаться каким-то сложным и запутанным. Но, если внимательно во всем разобраться, то все предельно просто, только несколько громоздко. Именно громоздко, а не сложно. Поэтому, при необходимости, просто посмотрите несколько раз объяснение схемы и я уверен, каждый из вас поймет принцип ее работы. Ну а мы реализуем ее в нашей программе.

После создания экземпляра класса и вывода локальных свойств объекта:

pt = Point3D(1, 2, 3)**print**(pt.\_\_dict\_\_)

увидим в консоли следующие строчки:

\_\_set\_\_: \_x = 1  
\_\_set\_\_: \_y = 2  
\_\_set\_\_: \_z = 3  
{'\_x': 1, '\_y': 2, '\_z': 3}

Последнее, что нужно прописать в дескрипторе – это проверку корректности данных. Для этого у нас уже есть метод verify\_coord, перенесем его в класс Integer и вызовем в сеттере:

**class** Integer:     @classmethod     **def** verify\_coord(cls, coord):         **if** type(coord) != int:             **raise** TypeError("Координата должна быть целым числом")       **def** \_\_set\_name\_\_(self, owner, name):         self.name = "\_" + name       **def** \_\_get\_\_(self, instance, owner):         **return** instance.\_\_dict\_\_[self.name]       **def** \_\_set\_\_(self, instance, value):         self.verify\_coord(value)         instance.\_\_dict\_\_[self.name] = value

Теперь, если при формировании объекта указать неверный тип данных:

pt = Point3D('1', 2, 3)

то увидим сообщение об ошибке.

Еще в классе Integer я сделаю обращение к атрибутам экземпляра через стандартные функции getattr и setattr:

**def** \_\_get\_\_(self, instance, owner):         **return** getattr(instance, self.name)       **def** \_\_set\_\_(self, instance, value):         self.verify\_coord(value)         setattr(instance, self.name, value)

Так будет правильнее, с точки зрения Python, чем обращение напрямую к специальной коллекции \_\_dict\_\_.

В итоге, мы с вами определили дескриптор данных (data descriptor) и на его основе создали три объекта x, y, z для интерфейса взаимодействия с координатами точки объектов класса Point3D.

Теперь вернемся к тому, что вначале я вам говорил, что важно различать дескрипторы данных и не данных. В чем разница? Ну, во-первых, очевидно, дескрипторы не данных не могут менять значения какого-либо свойства, так как не имеют сеттера и делитера. Они служат только для считывания информации. И есть второе важное отличие. Они имеют тот же приоритет доступа, что и обычные атрибуты класса. О чем здесь речь? Давайте я покажу это различие на примере. Добавим в программу еще один дескриптор, только не данных:

**class** ReadIntX:     **def** \_\_set\_name\_\_(self, owner, name):         self.name = "\_x"       **def** \_\_get\_\_(self, instance, owner):         **return** getattr(instance, self.name)

Он у нас будет считывать локальное свойство \_x. Определим его в классе Point3D:

    xr = ReadIntX()

И теперь можем использовать для считывания локального атрибута \_x:

**print**(pt.xr)

Как видите, все работает. Но, что будет, если мы запишем конструкцию:

pt.xr = 5

Произойдет ошибка? Нет! В экземпляре pt будет создано новое локальное свойство с именем xr и мы в этом можем убедиться:

**print**(pt.xr, pt.\_\_dict\_\_)

Кроме того, при обращении к pt.xr мы получаем значение 5, а не 1. Это, как раз и говорит о том, что приоритет доступа к локальным свойствам объекта и к дескриптору не данных одинаков.

Однако, если в дескриптор добавить сеттер и превратить его в дескриптор данных:

**class** ReadIntX:     **def** \_\_set\_name\_\_(self, owner, name):         self.name = "\_x"       **def** \_\_get\_\_(self, instance, owner):         **return** getattr(instance, self.name)       **def** \_\_set\_\_(self, instance, value):         setattr(instance, self.name, value)

А создание локального атрибута xr в объекте pt мы сделаем через коллекцию \_\_dict\_\_:

pt.\_\_dict\_\_['xr'] = 5

то при выполнении:

**print**(pt.xr, pt.\_\_dict\_\_)

увидим значение 1, хотя в объекте существует свойство xr. Это произошло потому, что приоритет обращению к дескриптору данных выше, чем к локальным атрибутам экземпляра класса. То есть, здесь все работает ровно так, как и с доступом к объектам-свойствам, о которых мы говорили на прошлых занятиях.

На этом мы завершим с вами знакомство с этой непростой темой «дескрипторы». Постарался объяснить все, как можно проще. Если тема не до конца понятна, то возможно нужно посмотреть ее несколько раз, либо совсем пропустить, так как это относительно редко используется в практике. Но знать этот функционал нужно, чтобы при необходимости не изобретать велосипед, а использовать уже встроенные возможности языка Python.