# **Свойства property. Декоратор @property**

[Смотреть материал на видео](https://www.youtube.com/watch?v=MxviMwbGl3I&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

На этом занятии вы узнаете о более удобном способе работы с приватными атрибутами через специальный объект property, который переводится как свойство. О чем здесь речь? Давайте представим, что мы разрабатываем класс для хранения и обработки данных о персонале:

**class** Person:     **def** \_\_init\_\_(self, name, old):         self.\_\_name = name         self.\_\_old = old

И для простоты, в нем будут сохраняться имя и возраст сотрудника в виде приватных атрибутов \_\_name и \_\_old. Разумеется, чтобы обращаться к таким закрытым данным, необходимы сеттеры и геттеры. Пропишем их для возраста:

**def** get\_old(self):         **return** self.\_\_old       **def** set\_old(self, old):         self.\_\_old = old

Я, надеюсь, вы помните, для чего делают реализации классов с приватными свойствами, а затем, добавляют еще методы для работы с ними? Мы об этом с вами уже говорили. Да, это необходимо, чтобы не нарушалась внутренняя логика работы алгоритма класса, а взаимодействие с классом и его объектами извне осуществлялась бы только через разрешенные (публичные) методы и свойства. Если вам это не понятно, то посмотрите внимательнее предыдущие занятия, а я продолжу.

Итак, теперь можно создать экземпляр этого класса:

p = Person('Сергей', 20)

и через геттер и сеттер считывать и менять возраст сотрудника:

p.set\_old(35)**print**(p.get\_old())

Это все мы с вами уже умеем и знаем. Но здесь есть одна маленькая проблема. Нам нужно прописывать разные сеттеры и геттеры для разных приватных атрибутов экземпляров класса. Например, добавить еще два для имени. В результате, пользователю этого класса (программисту) придется запоминать и держать в голове названия имен всех этих сеттеров и геттеров. Как можно было бы упростить работу с таким классом? Один из способов – воспользоваться объектом property. Давайте посмотрим на конкретном нашем примере, как это можно сделать.

В самом классе Person мы пропишем атрибут и придумаем ему имя, допустим, old. Этот атрибут класса будет ссылаться на объект property, которому мы передадим ссылку на геттер и сеттер:

old = property(get\_old, set\_old)

Что у нас тут с вами получилось? Смотрите. Из каждого экземпляра класса мы совершенно спокойно можем обращаться к атрибуту класса old. Этот атрибут является объектом property. Данный объект так устроен, что при считывании данных он вызывает первый метод get\_old, этот метод возвращает значение приватного локального свойства \_\_old экземпляра класса p и именно это значение дальше возвращается атрибутом old. Поэтому переменная a будет ссылаться на значение текущего возраста сотрудника.

Если же мы обращаемся к атрибуту класса old и присваиваем ему какое-то значение:

p.old = 35

то автоматически вызывается второй метод set\_old и в локальное свойство \_\_old заносится значение, указанное после оператора присваивания. В итоге, в текущем объекте p меняется локальное свойство \_\_old на новое.

Здесь у вас может возникнуть резонный вопрос, почему строчка:

p.old = 35

не создает новое локальное свойство внутри объекта p, как это у нас было ранее в программах, а обращается именно к атрибуту класса Person? Все дело в приоритете. Если в классе задан атрибут как объект-свойство, то в первую очередь выбирается оно, даже если в экземпляре класса есть локальное свойство с таким же именем. В этом легко убедиться. Давайте создадим свойство с именем old прямо в объекте p через словарь \_\_dict\_\_:

p.\_\_dict\_\_['old'] = 'old in object p'

А, затем выведем всю информацию в консоль:

**print**(p.old, p.\_\_dict\_\_)

Отображается значение 35, а не строка, то есть, было обращение именно к объекту-свойству old класса Person. А если свойству old в классе присвоить, какое-либо числовое значение, например, то будет отображена строка из объекта p. Здесь уже срабатывают знакомые нам приоритеты: сначала локальная область видимости объекта, затем, класса. Вот этот момент нужно хорошо знать, при работе с объектами-свойствами.

Итак, теперь у нас есть класс и мы можем менять приватное свойство \_\_old экземпляров этого класса через единый атрибут old (считывать информацию и записывать). Это гораздо удобнее использования сеттеров и геттеров. Здесь всего один атрибут и через него естественным образом происходит взаимодействие с закрытым свойством \_\_old.

## **Декоратор @property**

Я, думаю, из этого примера вы хорошо поняли, как создается объект property и для чего он нужен. Однако, в нашей реализации есть некое функциональное дублирование: мы можем работать с приватным свойством \_\_old и через сеттер/геттер и через свойство класса old. Конечно, это не критичный момент и на него можно не обращать внимания. Но, на мой взгляд было бы лучше, если бы у нас был один интерфейс взаимодействия со свойством \_\_old. Как это можно сделать?

Смотрите, вот этот класс property позволяет нам на уровне его объектов, использовать функции-декораторы. Если в консоли прописать:

a = property()

то через ссылку a нам будут доступны эти самые функции:

* a.getter() – декоратор для сеттера;
* a.setter() – декоратор для геттера;
* a.deleter() – декоратор для делитера.

Что такое декораторы вы должны уже знать, мы с вами об этом говорили в базовом курсе по Python. Ссылка на этот урок будет под этим видео. В двух словах, декоратор – это функция, которая расширяет функционал другой функции. То есть, вот эту строчку:

old = property(get\_old, set\_old)

можно переписать и так:

    old = property()     old = old.setter(set\_old)     old = old.getter(get\_old)

Это будет одно и то же. При вызове метода setter осуществляется встраиванием метода set\_old в алгоритм работы объекта property. И то же самое делает метод getter только для геттера. В обоих случаях они возвращают ссылку на объект property, который мы должны сохранять.

Так вот, мы можем использовать эти декораторы, чтобы сразу нужный нам метод класса превратить в объект-свойство property. Делается это очень просто. Перед геттером (обратите внимание, именно перед геттером, а не сеттером или делитером) прописывается декоратор:

    @property     **def** get\_old(self):         **return** self.\_\_old

По идее, он теперь представляет объект-свойство с именем get\_old:

**print**(p.get\_old)

Но пока присваивание не работает:

p.get\_old = 35

так как мы не прописали декоратор для сеттера. Делается это просто. Метод set\_old нужно переименовать в get\_old, чтобы имена совпадали (это обязательное условие) и перед ним прописать декоратор:

    @get\_old.setter     **def** get\_old(self, old):         self.\_\_old = old

Все, мы сформировали новый объект-свойство с именем get\_old. Давайте его переименуем просто в old, а строчки ниже удалим, получим следующий класс Person:

**class** Person:     **def** \_\_init\_\_(self, name, old):         self.\_\_name = name         self.\_\_old = old       @property     **def** old(self):         **return** self.\_\_old       @old.setter     **def** old(self, old):         self.\_\_old = old

То, что мы сделали, эквивалентно предыдущему варианту с тем лишь отличием, что теперь напрямую вызывать сеттер или геттер для локального свойства \_\_old не получится. У нас остался один интерфейс взаимодействия – объект-свойство old. Именно так чаще всего делают на практике.

В заключение этого занятия добавлю еще один метод делитер, который вызывается при удалении свойства:

    @old.deleter     **def** old(self):         **del** self.\_\_old

Теперь, если выполнить команду:

**del** p.old**print**(p.\_\_dict\_\_)

то сработает делитер и мы увидим, что локальное свойство \_\_old было удалено. Конечно, его легко снова создать, достаточно выполнить присвоение, то есть, вызвать сеттер:

p.old = 10

Вот так гибко можно работать с приватными (закрытыми) локальными свойствами через объект-свойство property. Надеюсь, из этого занятия вы узнали, что это такое, зачем надо и как формировать свои объекты-свойства или через класс property или через декораторы.