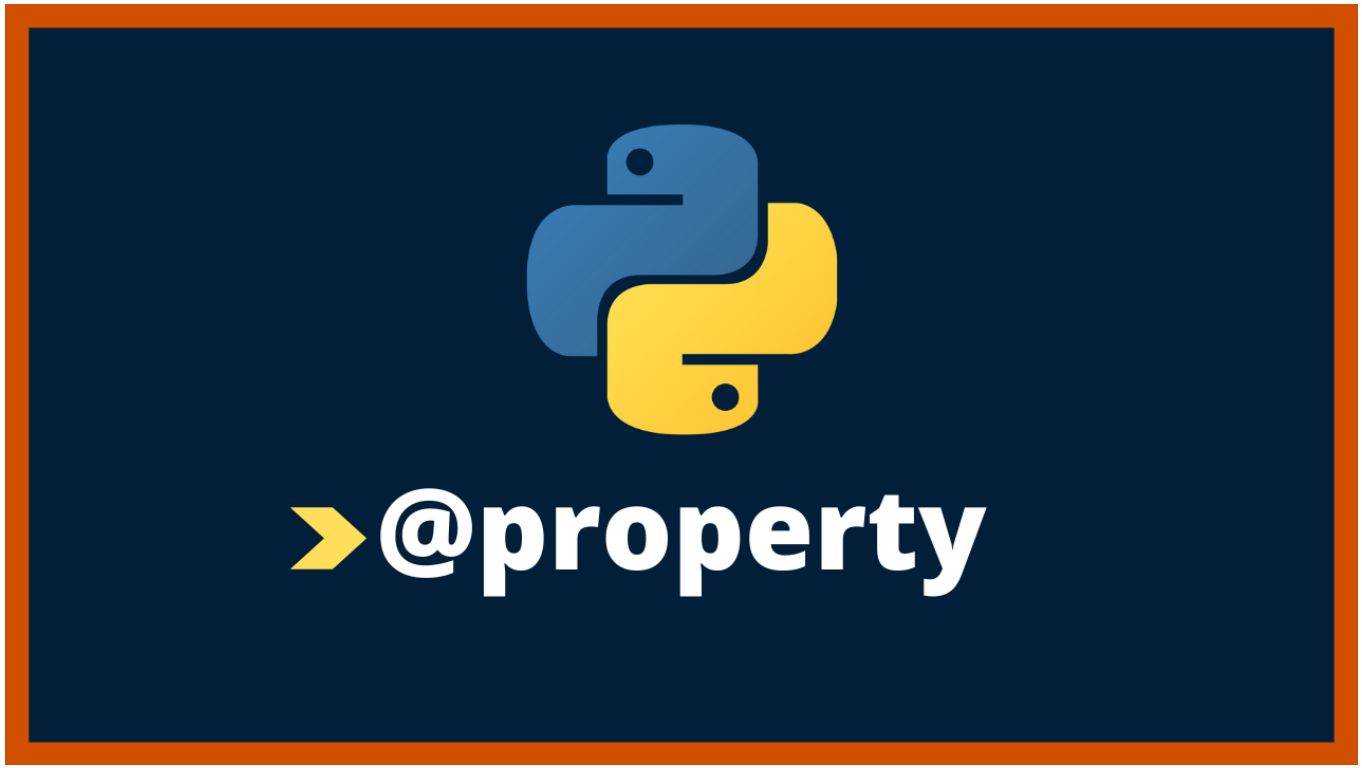
19 DECEMBER 2019 / #PYTHON

The @property Decorator in Python: Its Use Cases, Advantages, and Syntax

Estefania Cassingena Navone

Computer Science & Mathematics Student | Udemy Instructor | Author at freeCodeCamp News



Welcome! In this article, you will learn how to work with the `@property` decorator in Python.

You will learn:

- The advantages of working with properties in Python.
- The basics of decorator functions: what they are and how they are related to @property.
- How you can use @property to define getters, setters, and deleters.

1️⃣ Advantages of Properties in Python

Let's start with a little bit of context. **Why** would you use properties in Python?

Properties can be considered the "Pythonic" way of working with attributes because:

- The syntax used to define properties is very concise and readable.
- You can access instance attributes exactly as if they were public attributes while using the "magic" of intermediaries (getters and setters) to validate new values and to avoid accessing or modifying the data directly.
- By using @property, you can "reuse" the name of a property to avoid creating new names for the getters, setters, and deleters.

These advantages make properties a really awesome tool to help you write more concise and readable code. 👍

2️⃣ Intro to Decorators

A **decorator function** is basically a function that adds new functionality to a function

sprinkles to an ice cream 🍨. It lets us add new functionality to an existing function without modifying it.
In the example below, you can see what a typical decorator function looks like in Python:

```
def decorator(f):
    def new_function():
        print("Extra Functionality")
        f()
    return new_function

@decorator
def initial_function():
    print("Initial Functionality")

initial_function()
```

Let's analyze these elements in detail:

- We first find the decorator function `def decorator(f)` (the sprinkles ✨) that takes a function `f` as an argument.

```
def decorator(f):
    def new_function():
        print("Extra Functionality")
        f()
    return new_function
```

- This decorator function has an nested function, `new_function`. Notice how `f` is called inside `new_function` to achieve the same functionality while adding new functionality before the function call (we could also add new functionality after the function call).
- The decorator function itself returns the nested function `new_function`.

`initial_function` . Notice the very peculiar syntax (`@decorator`) above the function header.

```
@decorator
def initial_function():
    print("Initial Functionality")

initial_function()
```

If we run the code, we see this output:

```
Extra Functionality
Initial Functionality
```

Notice how the decorator function runs even if we are only calling `initial_function()` . This is the magic of adding @decorator 🔮.

🔔 **Note:** In general, we would write `@<decorator_function_name>` , replacing the name of the decorator function after the @ symbol.

I know you may be asking: how is this related to @property? @property is a built-in decorator for the property() function in Python. It is used to give "special" functionality to certain methods to make them act as getters, setters, or deleters when we define properties in a class.

Now that you are familiar with decorators, let's see a real scenario of the use of @property!

🔶 Real-World Scenario: @property

class (at the moment, the class only has a *price* instance attribute defined):

```
class House:

    def __init__(self, price):
        self.price = price
```

This instance attribute is public because its name doesn't have a leading underscore. Since the attribute is currently public, it is very likely that you and other developers in your team accessed and modified the attribute **directly** in other parts of the program using dot notation, like this:

```
# Access value
obj.price

# Modify value
obj.price = 40000
```

💡 **Tip:** *obj* represents a variable that references an instance of `House`.

So far everything is working great, right? **But let's say that you are asked to make this attribute protected (non-public) and validate the new value before assigning it**. Specifically, you need to check if the value is a positive float. How would you do that? Let's see.

Changing your Code

At this point, if you decide to add getters and setters, you and your team will probably panic 😱. This is because each line of code that accesses or modifies the value of the attribute will have to be modified to call the getter or setter, respectively. Otherwise, the code will break ⚠️.

```
# Changed from obj.price
obj.get_price()

# Changed from obj.price = 40000
obj.set_price(40000)
```

But... Properties come to the rescue! With `@property`, you and your team will not need to modify any of those lines because you will able to add getters and setters "behind the scenes" without affecting the syntax that you used to access or modify the attribute when it was public.

Awesome, right?🎉

🔶 @property: Syntax and Logic

If you decide to use `@property`, your class will look like the example below:

```
class House:

    def __init__(self, price):
        self._price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, new_price):
        if new_price > 0 and isinstance(new_price, float):
            self._price = new_price
        else:
            print("Please enter a valid price")

    @price.deleter
    def price(self):
        del self._price
```

Specifically, you can define **three methods** for a property:

- A **setter** - to set the value of the attribute.
- A **deleter** - to delete the instance attribute.

Price is now "Protected"
Please note that the *price* attribute is now considered "protected" because we added a leading underscore to its name in `self._price`:

```
self._price = price
```

In Python, by convention, when you add a leading underscore to a name, you are telling other developers that it should not be accessed or modified directly outside of the class. It should only be accessed through intermediaries (getters and setters) if they are available.

📌 Getter

Here we have the getter method:

```
@property
def price(self):
    return self._price
```

Notice the syntax:

- `@property` - Used to indicate that we are going to define a property. Notice how this immediately improves readability because we can clearly see the purpose of this method.

property that we are defining: *price*. This is the name that we will use to access and modify the attribute outside of the class. The method only takes one formal parameter, self, which is a reference to the instance.

- `return self._price` - This line is exactly what you would expect in a regular getter. The value of the protected attribute is returned.

Here is an example of the use of the getter method:

```
>>> house = House(50000.0) # Create instance
>>> house.price            # Access value
50000.0
```

Notice how we access the *price* attribute as if it were a public attribute. We are not changing the syntax at all, but we are actually using the getter as an intermediary to avoid accessing the data directly.

📌 Setter

Now we have the setter method:

```
@price.setter
def price(self, new_price):
    if new_price > 0 and isinstance(new_price, float):
        self._price = new_price
    else:
        print("Please enter a valid price")
```

Notice the syntax:

- `@price.setter` - Used to indicate that this is the *setter* method for the *price* property. Notice that we are **not** using *@**property**.setter*, we are using *@**price**.setter*. The name of the property is included before *.setter*.

how the name of the property is used as the name of the setter. We also have a second formal parameter (*new_price*), which is the new value that will be assigned to the *price* attribute (if it is valid).

- Finally, we have the body of the setter where we **validate** the argument to check if it is a positive float and then, if the argument is valid, we update the value of the attribute. If the value is not valid, a descriptive message is printed. You can choose how to handle invalid values according the needs of your program.

This is an example of the use of the setter method with @property:

```
>>> house = House(50000.0)  # Create instance
>>> house.price = 45000.0   # Update value
>>> house.price             # Access value
45000.0
```

Notice how we are not changing the syntax, but now we are using an intermediary (the setter) to validate the argument before assigning it. The new value (45000.0) is passed as an argument to the setter :

```
house.price = 45000.0
```

If we try to assign an invalid value, we see the descriptive message. We can also check that the value was not updated:

```
>>> house = House(50000.0)
>>> house.price = -50
Please enter a valid price
```

💡 **Tip:** This proves that the setter method is working as an intermediary. It is being called "behind the scenes" when we try to update the value, so the descriptive message is displayed when the value is not valid.

📌 Deleter

Finally, we have the deleter method:

```
@price.deleter
def price(self):
    del self._price
```

Notice the syntax:

- `@price.deleter` - Used to indicate that this is the *deleter* method for the *price* property. Notice that this line is very similar to @price.setter, but now we are defining the deleter method, so we write @price.**deleter**.
- `def price(self):` - The header. This method only has one formal parameter defined, self.
- `del self._price` - The body, where we delete the instance attribute.

💡 **Tip:** Notice that the name of the property is "reused" for all three methods.

This is an example of the use of the deleter method with @property:

```
# Create instance
>>> house = House(50000.0)
```

```
50000.0

# Delete the instance attribute
>>> del house.price

# The instance attribute doesn't exist
>>> house.price
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    house.price
  File "<pyshell#20>", line 8, in price
    return self._price
AttributeError: 'House' object has no attribute '_price'
```

The instance attribute was deleted successfully 👍. When we try to access it again, an error is thrown because the attribute doesn't exist anymore.

🔔 Some final Tips

You don't necessarily have to define all three methods for every property. You can define read-only properties by only including a getter method. You could also choose to define a getter and setter without a deleter.

If you think that an attribute should only be set when the instance is created or that it should only be modified internally within the class, you can omit the setter.

You can choose which methods to include depending on the context that you are working with.

📣 In Summary

- You can define properties with the @property syntax, which is more compact and readable.
- @property can be considered the "pythonic" way of defining getters, setters, and deleters.

without affecting the program, so you can add getters, setters, and deleters that act as intermediaries "behind the scenes" to avoid accessing or modifying the data directly.

I really hope you liked my article and found it helpful. 👍 To learn more about Properties and Object Oriented Programming in Python, check out my online course, which includes 6+ hours of video lectures, coding exercises, and mini projects.
