Python @property decorator

programiz.com/python-programming/property

Join our newsletter for the latest updates.

In this tutorial, you will learn about Python @property decorator; a pythonic way to use getters and setters in object-oriented programming.

Python programming provides us with a built-in `@property` decorator which makes usage of getter and setters much easier in Object-Oriented Programming.

Before going into details on what `@property` decorator is, let us first build an intuition on why it would be needed in the first place.

Class Without Getters and Setters

Let us assume that we decide to make a class that stores the temperature in degrees Celsius. It would also implement a method to convert the temperature into degrees Fahrenheit. One way of doing this is as follows:

```
class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32
```

We can make objects out of this class and manipulate the `temperature` attribute as we wish:

```
# Basic method of setting and getting attributes in Python
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32


# Create a new object
human = Celsius()

# Set the temperature
human.temperature = 37

# Get the temperature attribute
print(human.temperature)

# Get the to_fahrenheit method
print(human.to_fahrenheit())
```

Output

```
37
98.60000000000001
```

The extra decimal places when converting into Fahrenheit is due to the floating point arithmetic error. To learn more, visit Python Floating Point Arithmetic Error.

Whenever we assign or retrieve any object attribute like `temperature` as shown above, Python searches it in the object's built-in `__dict__` dictionary attribute.

```
>>> human.__dict__
{'temperature': 37}
```

Therefore, `man.temperature` internally becomes `man.__dict__['temperature']`.

Using Getters and Setters

Suppose we want to extend the usability of the *Celsius* class defined above. We know that the temperature of any object cannot reach below -273.15 degrees Celsius (Absolute Zero in Thermodynamics)

Let's update our code to implement this value constraint.

An obvious solution to the above restriction will be to hide the attribute `temperature` (make it private) and define new getter and setter methods to manipulate it. This can be done as follows:

```
# Making Getters and Setter methods
class Celsius:
    def __init__(self, temperature=0):
        self.set_temperature(temperature)

    def to_fahrenheit(self):
        return (self.get_temperature() * 1.8) + 32

    # getter method
    def get_temperature(self):
        return self._temperature

    # setter method
    def set_temperature(self, value):
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible.")
        self._temperature = value
```

As we can see, the above method introduces two new `get_temperature()` and `set_temperature()` methods.

Furthermore, `temperature` was replaced with `_temperature`. An underscore `_` at the beginning is used to denote private variables in Python.

Now, let's use this implementation:

```
# Making Getters and Setter methods
class Celsius:
    def __init__(self, temperature=0):
        self.set_temperature(temperature)

    def to_fahrenheit(self):
        return (self.get_temperature() * 1.8) + 32

    # getter method
    def get_temperature(self):
        return self._temperature

    # setter method
    def set_temperature(self, value):
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible.")
        self._temperature = value


# Create a new object, set_temperature() internally called by __init__
human = Celsius(37)

# Get the temperature attribute via a getter
print(human.get_temperature())

# Get the to_fahrenheit method, get_temperature() called by the method itself
print(human.to_fahrenheit())

# new constraint implementation
human.set_temperature(-300)

# Get the to_fahreheit method
print(human.to_fahrenheit())
```

Output

```
37
98.60000000000001
Traceback (most recent call last):
  File "<string>", line 30, in <module>
  File "<string>", line 16, in set_temperature
ValueError: Temperature below -273.15 is not possible.
```

This update successfully implemented the new restriction. We are no longer allowed to set the temperature below -273.15 degrees Celsius.

Note: The private variables don't actually exist in Python. There are simply norms to be followed. The language itself doesn't apply any restrictions.

```
>>> human._temperature = -300
>>> human.get_temperature()
-300
```

However, the bigger problem with the above update is that all the programs that implemented our previous class have to modify their code from `obj.temperature` to `obj.get_temperature()` and all expressions like `obj.temperature = val` to `obj.set_temperature(val)` .

This refactoring can cause problems while dealing with hundreds of thousands of lines of codes.

All in all, our new update was not backwards compatible. This is where `@property` comes to rescue.

The property Class

A pythonic way to deal with the above problem is to use the `property` class. Here is how we can update our code:

```
# using property class
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    # getter
    def get_temperature(self):
        print("Getting value...")
        return self._temperature

    # setter
    def set_temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible")
        self._temperature = value

    # creating a property object
    temperature = property(get_temperature, set_temperature)
```

We added a `print()` function inside `get_temperature()` and `set_temperature()` to clearly observe that they are being executed.

The last line of the code makes a property object `temperature` . Simply put, property attaches some code (`get_temperature` and `set_temperature`) to the member attribute accesses (`temperature`).

Let's use this update code:

```
# using property class
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    # getter
    def get_temperature(self):
        print("Getting value...")
        return self._temperature

    # setter
    def set_temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible")
        self._temperature = value

    # creating a property object
    temperature = property(get_temperature, set_temperature)


human = Celsius(37)

print(human.temperature)

print(human.to_fahrenheit())

human.temperature = -300
```

Output

```
Setting value...
Getting value...
37
Getting value...
98.60000000000001
Setting value...
Traceback (most recent call last):
  File "<string>", line 31, in <module>
  File "<string>", line 18, in set_temperature
ValueError: Temperature below -273 is not possible
```

As we can see, any code that retrieves the value of `temperature` will automatically call `get_temperature()` instead of a dictionary (__dict__) look-up. Similarly, any code that assigns a value to `temperature` will automatically call `set_temperature()` .

We can even see above that `set_temperature()` was called even when we created an object.

```
>>> human = Celsius(37)
Setting value...
```

Can you guess why?

The reason is that when an object is created, the `__init__()` method gets called. This method has the line `self.temperature = temperature`. This expression automatically calls `set_temperature()`.

Similarly, any access like `c.temperature` automatically calls `get_temperature()`. This is what property does. Here are a few more examples.

```
>>> human.temperature
Getting value
37
>>> human.temperature = 37
Setting value

>>> c.to_fahrenheit()
Getting value
98.60000000000001
```

By using `property`, we can see that no modification is required in the implementation of the value constraint. Thus, our implementation is backward compatible.

Note: The actual temperature value is stored in the private `_temperature` variable. The `temperature` attribute is a property object which provides an interface to this private variable.

The @property Decorator

In Python, `property()` is a built-in function that creates and returns a `property` object. The syntax of this function is:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

where,

- `fget` is function to get value of the attribute
- `fset` is function to set value of the attribute
- `fdel` is function to delete the attribute
- `doc` is a string (like a comment)

As seen from the implementation, these function arguments are optional. So, a property object can simply be created as follows.

```
>>> property()
<property object at 0x0000000003239B38>
```

A property object has three methods, `getter()`, `setter()`, and `deleter()` to specify `fget`, `fset` and `fdel` at a later point. This means, the line:

```
temperature = property(get_temperature,set_temperature)
```

can be broken down as:

```
# make empty property
temperature = property()
# assign fget
temperature = temperature.getter(get_temperature)
# assign fset
temperature = temperature.setter(set_temperature)
```

These two pieces of codes are equivalent.

Programmers familiar with Python Decorators can recognize that the above construct can be implemented as decorators.

We can even not define the names `get_temperature` and `set_temperature` as they are unnecessary and pollute the class namespace.

For this, we reuse the `temperature` name while defining our getter and setter functions. Let's look at how to implement this as a decorator:

```
# Using @property decorator
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        print("Getting value...")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value


# create an object
human = Celsius(37)

print(human.temperature)

print(human.to_fahrenheit())

coldest_thing = Celsius(-300)
```

Output

```
Setting value...
Getting value...
37
Getting value...
98.60000000000001
Setting value...
Traceback (most recent call last):
  File "<string>", line 29, in <module>
  File "<string>", line 4, in __init__
  File "<string>", line 18, in temperature
ValueError: Temperature below -273 is not possible
```

The above implementation is simple and efficient. It is the recommended way to use `property`.