Python @property

Python has a great concept called property which makes the life of an object oriented programmer much simpler.

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

## An Example To Begin With

Let us assume that you decide to make a class that could store the temperature in degree Celsius. It would also implement a method to convert the temperature into degree 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 could make objects out of this class and manipulate the attribute temperature as we wished. Try these on Python shell.

>>> # create new object

>>> man = Celsius()

>>> # set temperature

>>> man.temperature = 37

>>> # get temperature

>>> man.temperature

37

>>> # get degrees Fahrenheit

>>> man.to\_fahrenheit()

98.60000000000001

The extra decimal places when converting into Fahrenheit is due to the floating point arithmetic error (try 1.1 + 2.2 in the Python interpreter).

Whenever we assign or retrieve any object attribute like temperature, as show above, Python searches it in the object's \_\_dict\_\_ dictionary.

>>> man.\_\_dict\_\_

{'temperature': 37}

Therefore, man.temperature internally becomes man.\_\_dict\_\_['temperature'].

Now, let's further assume that our class got popular among clients and they started using it in their programs. They did all kinds of assignments to the object.

One fateful day, a trusted client came to us and suggested that temperatures cannot go below -273 degree Celsius (students of thermodynamics might argue that it's actually -273.15), also called the absolute zero. He further asked us to implement this value constraint. Being a company that strive for customer satisfaction, we happily heeded the suggestion and released version 1.01 (an upgrade of our existing class).

## Using Getters and Setters

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

class Celsius:

def \_\_init\_\_(self, temperature = 0):

self.set\_temperature(temperature)

def to\_fahrenheit(self):

return (self.get\_temperature() \* 1.8) + 32

# new update

def get\_temperature(self):

return self.\_temperature

def set\_temperature(self, value):

if value < -273:

raise ValueError("Temperature below -273 is not possible")

self.\_temperature = value

We can see above that new methods get\_temperature() and set\_temperature() were defined and furthermore, temperature was replaced with \_temperature. An underscore (\_) at the beginning is used to denote private variables in Python.

>>> c = Celsius(-277)

Traceback (most recent call last):

...

ValueError: Temperature below -273 is not possible

>>> c = Celsius(37)

>>> c.get\_temperature()

37

>>> c.set\_temperature(10)

>>> c.set\_temperature(-300)

Traceback (most recent call last):

...

ValueError: Temperature below -273 is not possible

This update successfully implemented the new restriction. We are no longer allowed to set temperature below -273.

Please note that private variables don't exist in Python. There are simply norms to be followed. The language itself don't apply any restrictions.

>>> c.\_temperature = -300

>>> c.get\_temperature()

-300

But this is not of great concern. The big problem with the above update is that, all the clients who implemented our previous class in their program have to modify their code from obj.temperature to obj.get\_temperature() and all assignments like obj.temperature = val to obj.set\_temperature(val).

This refactoring can cause headaches to the clients with hundreds of thousands of lines of codes.

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

## The Power of @property

The pythonic way to deal with the above problem is to use property. Here is how we could have achieved it.

class Celsius:

def \_\_init\_\_(self, temperature = 0):

self.temperature = temperature

def to\_fahrenheit(self):

return (self.temperature \* 1.8) + 32

def get\_temperature(self):

print("Getting value")

return self.\_temperature

def set\_temperature(self, value):

if value < -273:

raise ValueError("Temperature below -273 is not possible")

print("Setting value")

self.\_temperature = value

temperature = property(get\_temperature,set\_temperature)

And, issue the following code in shell once you run it.

>>> c = Celsius()

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).

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(). This is one cool feature in Python.

We can see above that set\_temperature() was called even when we created an object.

**Can you guess why?**

The reason is that when an object is created, \_\_init\_\_() method gets called. This method has the line self.temperature = temperature. This assignment automatically called set\_temperature().

>>> c.temperature

Getting value

0

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

>>> c.temperature = 37

Setting value

>>> c.to\_fahrenheit()

Getting value

98.60000000000001

By using property, we can see that, we modified our class and implemented the value constraint without any change required to the client code. Thus our implementation was backward compatible and everybody is happy.

Finally note that, the actual temperature value is stored in the private variable \_temperature. The attribute temperature is a property object which provides interface to this private variable.

## Digging Deeper into Property

In Python, property() is a built-in function that creates and returns a property object. The signature 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 and 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 delete() to specify fget, fset and fdel at a later point. This means, the line

temperature = property(get\_temperature,set\_temperature)

could have been 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 decorators in Python can recognize that the above construct can be implemented as decorators.

We can further go on and not define names get\_temperature and set\_temperature as they are unnecessary and pollute the class namespace. For this, we reuse the name temperaturewhile defining our getter and setter functions. This is how it can be done.

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):

if value < -273:

raise ValueError("Temperature below -273 is not possible")

print("Setting value")

self.\_temperature = value

The above implementation is both, simple and recommended way to make properties. You will most likely encounter these types of constructs when looking for property in Python.

Well that's it for today.

## Simple Caculator by Making Functions

''' Program make a simple calculator that can add, subtract, multiply and divide using functions '''

# This function adds two numbers

def add(x, y):

return x + y

# This function subtracts two numbers

def subtract(x, y):

return x - y

# This function multiplies two numbers

def multiply(x, y):

return x \* y

# This function divides two numbers

def divide(x, y):

return x / y

print("Select operation.")

print("1.Add")

print("2.Subtract")

print("3.Multiply")

print("4.Divide")

# Take input from the user

choice = input("Enter choice(1/2/3/4):")

num1 = int(input("Enter first number: "))

num2 = int(input("Enter second number: "))

if choice == '1':

print(num1,"+",num2,"=", add(num1,num2))

elif choice == '2':

print(num1,"-",num2,"=", subtract(num1,num2))

elif choice == '3':

print(num1,"\*",num2,"=", multiply(num1,num2))

elif choice == '4':

print(num1,"/",num2,"=", divide(num1,num2))

else:

print("Invalid input")