Replacing Setters and Getters with a Python property

WE'LL COVER THE FOLLOWING

Let's pretend that we have some legacy code that someone wrote who didn't understand Python very well. If you're like me, you've already seen this kind of code before:

```
from decimal import Decimal

class Fees(object):
    """"""

    def __init__(self):
        """Constructor"""
        self._fee = None

    def get_fee(self):
        """
        Return the current fee
        """
        return self._fee

    def set_fee(self, value):
        """
        Set the fee
        """
        if isinstance(value, str):
            self._fee = Decimal(value)
        elif isinstance(value, Decimal):
            self._fee = value
```

To use this class, we have to use the setters and getters that are defined:

```
f = Fees()
f.set_fee("1")
f.get_fee()
print(Decimal('1'))
```

If you want to add the normal dot notation access of attributes to this code without breaking all the applications that depend on this piece of code, you can change it very simply by adding a property:

```
from decimal import Decimal

class Fees(object):
    """"""

    def __init__(self):
        """Constructor"""
        self._fee = None

    def get_fee(self):
        """
        Return the current fee
        """
        return self._fee

    def set_fee(self, value):
        """
        Set the fee
        """
        if isinstance(value, str):
            self._fee = Decimal(value)
        elif isinstance(value, Decimal):
            self._fee = value

    fee = property(get_fee, set_fee)
```

We added one line to the end of this code. Now we can do stuff like this:

```
f = Fees()
f.set_fee("1")
f.fee
print(Decimal('1'))

f.fee = "2"
f.get_fee()
print(Decimal('2'))
```

As you can see, when we use **property** in this manner, it allows the fee property to set and get the value itself without breaking the legacy code. Let's

rewrite this code using the property decorator and see if we can get it to allow setting.

```
from decimal import Decimal

class Fees(object):
    """"""

    def __init__(self):
        """Constructor"""
        self._fee = None

    @property
    def fee(self):
        """
        The fee property - the getter
        """
        return self._fee

    @fee.setter
    def fee(self, value):
        """
        The setter of the fee property
        """
        if isinstance(value, str):
            self._fee = Decimal(value)
        elif isinstance(value, Decimal):
            self._fee = value

if __name__ == "__main__":
    f = Fees()
```

The code above demonstrates how to create a “setter” for the **fee** property. You can do this by decorating a second method that is also called **fee** with a decorator called <**@fee.setter**>. The setter is invoked when you do something like this:

```
f = Fees()
f.fee = "1"
```

If you look at the signature for **property**, it has fget, fset, fdel and doc as “arguments”. You can create another decorated method using the same name to correspond to a delete function using <**@fee.deleter*>* if you want to catch the **del** command against the attribute.

Wrapping Up

At this point you should know how to create your own decorators and how to use a few of Python’s built-in decorators. We looked at @classmethod, @property and @staticmethod. I would be curious to know how my readers

use the built-in decorators and how they use their own custom decorators.