Python Property Decorator

Summary: in this tutorial, you'll learn about Python property decorator (@property) and more importantly how it works.

Introduction to the Python property decorator

In the previous tutorial, you learned how to use the property (https://www.pythontutorial.net/python-oop/python-properties/) class to add a property to a class. Here's the syntax of the `property` class:

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

The following defines a `Person` class with two attributes `name` and `age` :

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

To define a getter for the `age` attribute, you use the `property` class like this:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age

    def get_age(self):
        return self._age

    age = property(fget=get_age)
```

The `property` accepts a getter and returns a property object.

The following creates an instance of the `Person` class and get the value of the `age` property via the instance:

```
john = Person('John', 25)
print(john.age)
```

Output:

```
25
```

Also, you can call the `get_age()` method of the `Person` object directly like this:

```
print(john.get_age())
```

So to get the `age` of a `Person` object, you can use either the `age` property or the `get_age()` method. This creates an unnecessary redundancy.

To avoid this redundancy, you can rename the `get_age()` method to the `age()` method like this:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age

    def age(self):
        return self._age

    age = property(fget=age)
```

The `property()` accepts a callable (age) and returns a callable. Therefore, it is a decorator. Therefore, you can use the `@property` decorator to decorate the `age()` method as follows:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age

    @property
    def age(self):
        return self._age
```

So by using the `@property` decorator, you can simplify the property definition for a class.

Setter decorators

The following adds a setter method (`set_age`) to assign a value to `_age` attribute to the `Person` class:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age
    @property
    def age(self):
        return self._age

    def set_age(self, value):
        if value <= 0:
            raise ValueError('The age must be positive')
        self._age = value
```

To assign the `set_age` to the `fset` of the `age` property object, you call the `setter()` method of the `age` property object like the following:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age

    @property
    def age(self):
        return self._age

    def set_age(self, value):
        if value <= 0:
            raise ValueError('The age must be positive')
        self._age = value

    age = age.setter(set_age)
```

The `setter()` method accepts a callable and returns another callable (a `property` object). Therefore, you can use the decorator `@age.setter` for the `set_age()` method like this:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def set_age(self, value):
        if value <= 0:
            raise ValueError('The age must be positive')
        self._age = value
```

Now, you can change the `set_age()` method to the `age()` method and use the `age` property in the `__init__()` method:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value <= 0:
            raise ValueError('The age must be positive')
        self._age = value
```

To summarize, you can use decorators to create a property using the following pattern:

```
class MyClass:
    def __init__(self, attr):
        self.prop = attr

    @property
    def prop(self):
        return self.__attr

    @prop.setter
    def prop(self, value):
        self.__attr = value
```

In this pattern, the `__attr` is the private attribute and `prop` is the property name.

The following example uses the `@property` decorators to create the `name` and `age` properties in the `Person` class:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value <= 0:
            raise ValueError('The age must be positive')
        self._age = value

    @property
    def name(self):
        return self._age

    @name.setter
    def name(self, value):
        if value.strip() == '':
            raise ValueError('The name cannot be empty')
        self._age = value
```

Summary

- Use the `@property` decorator to create a property for a class.