Python Property

Summary: in this tutorial, you'll learn about the Python `property` class and how to use it to define properties for a class.

Introduction to class properties

The following defines a `Person` class (https://www.pythontutorial.net/python-oop/python-class/) that has two attributes (https://www.pythontutorial.net/python-basics/python-variables/) `name` and `age`, and create a new instance of the `Person` class:

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


john = Person('John', 18)
```

Since `age` is the instance attribute (https://www.pythontutorial.net/python-oop/python-instance-variables/) of the `Person` class, you can assign it a new value like this:

```
john.age = 19
```

The following assignment is also technically valid:

```
john.age = -1
```

However, the age is semantically incorrect.

To ensure that the age is not zero or negative, you use the `if` (https://www.pythontutorial.net/python-basics/python-if/) statement to add a check as follows:

```
age = -1
if age <= 0:
    raise ValueError('The age must be positive')
else:
    john.age = age
```

And you need to do this every time you want to assign a value to the `age` attribute. This is repetitive and difficult to maintain.

To avoid this repetition, you can define a pair of methods called getter and setter.

Getter and setter

The getter and setter methods provide an interface for accessing an instance attribute:

- The getter returns the value of an attribute
- The setter sets a new value for an attribute

In our example, you can make the `age` attribute private (https://www.pythontutorial.net/python-oop/python-private-attributes/) (by convention) and define a getter and a setter to manipulate the `age` attribute.

The following shows the new `Person` class with a getter and setter for the `age` attribute:

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

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

    def get_age(self):
        return self._age
```

How it works.

In the `Person` class, the `set_age()` is the setter and the `get_age()` is the getter. By convention the getter and setter have the following name: `get_<attribute>()` and `set_<attribute>()` .

In the `set_age()` method, we raise a `ValueError` if the `age` is less than or equal to zero. Otherwise, we assign the `age` argument to the `_age` attribute:

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

The `get_age()` method returns the value of the `_age` attribute:

```python
def get_age(self):
    return self._age
```

In the `__init__()` method, we call the `set_age()` setter method to initialize the `_age` attribute:

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

The following attempts to assign an invalid value to the `age` attribute:

```
john = Person('John', 18)
john.set_age(-19)
```

And Python issued a `ValueError` as expected.

```
ValueError: The age must be positive
```

This code works just fine. But it has a backward compatibility issue.

Suppose you released the `Person` class for a while and other developers have been already using it. And now you add the getter and setter, all the code that uses the Person won't work anymore.

To define a getter and setter method while achieving backward compatibility, you can use the `property()` class.

The Python property class

The property class returns a `property` object. The `property()` class has the following syntax:

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

The `property()` has the following parameters:

- `fget` is a function to get the value of the attribute, or the getter method.
- `fset` is a function to set the value of the attribute, or the setter method.
- `fdel` is a function to delete the attribute.
- `doc` is a docstring i.e., a comment.

The following uses the `property()` function to define the `age` property for the `Person` class.

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

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

    def get_age(self):
        return self._age

    age = property(fget=get_age, fset=set_age)
```

In the `Person` class, we create a new property object by calling the `property()` and assign the property object to the age attribute. Note that the `age` is a class attribute (https://www.pythontutorial.net/python-oop/python-class-attributes/) , not an instance attribute (https://www.pythontutorial.net/python-oop/python-instance-variables/) .

The following shows that the `Person.age` is a `property` object:

```
print(Person.age)
```

Output:

```
<property object at 0x000001F5F5149180>
```

The following creates a new instance of the `Person` class and access the `age` attribute:

```
john = Person('John', 18)
```

The `john.__dict__` stores the instance attributes of the john object. The following shows the contents of the `john.__dict__` :

```
print(john.__dict__)
```

Output:

```
{'_age': 18, 'name': 'John'}
```

As you can see clearly from the output, the `john.__dict__` doesn't have the `age` attribute.

The following assigns a value to the `age` attribute of the john object:

```
john.age = 19
```

In this case, Python looks up the `age` attribute in the `john.__dict__` first. Because Python doesn't find the `age` attribute in the `john.__dict__` , it'll then find the `age` attribute in the `Person.__dict__` .

The `Person.__dict__` stores the class attributes of the Person class. The following shows the contents of the `Person.__dict__` :

```
pprint(Person.__dict__)
```

Output:

```
mappingproxy({'__dict__': <attribute '__dict__' of 'Person' objects>,
              '__doc__': None,
              '__init__': <function Person.__init__ at 0x000002242F5B2670>,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Person' objects>,
              'age': <property object at 0x000002242EE39180>,
              'get_age': <function Person.get_age at 0x000002242F5B2790>,
              'set_age': <function Person.set_age at 0x000002242F5B2700>})
```

Because Python finds the `age` attribute in the `Person.__dict__` , it'll call the `age` property object.

When you assign a value to the `age` object:

```
john.age = 19
```

Python will call the function assigned to the `fset` argument, which is the `set_age()` .

Similarly, when you read from the `age` property object, Python will execute the function assigned to the `fget` argument, which is the `get_age()` method.

By using the `property()` class, we can add a property to a class while maintaining backward compatibility. In practice, you will define the attributes first. Later, you can add the property to the class if needed.

Putting it all together.

```
from pprint import pprint


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

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

    def get_age(self):
        return self._age
    age = property(fget=get_age, fset=set_age)


print(Person.age)

john = Person('John', 18)
pprint(john.__dict__)

john.age = 19
pprint(Person.__dict__)
```

Summary

- Use the Python `property()` class to define a property for a class.