Class Creation: Getter/Setter vs Property

Getter and Setter Pattern comes from OOP languages such as C++ and Java in order to adhere to

- Encapsulation
  - the bundling of data/attributes and methods.
  - restricting access to implementation details and violating state invariants

In Python it is considered unpythonic. Instead, use @property decorator instead

Class without Getter and Setter

```
# Basic method of setting and getting attributes in Python
# If this class had no methods, and only an __init__, then this would be fine.
# This has no Encapsulation because it directly accesses the attributes


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())


>> 37
>> 98.60000000000001
```

Getter and Setter Pattern

- Class Attributes...
    - are variables where you can access through the instance, class, or both.
    - holds an internal state.
    - Two ways to access class attributes:
        1. Directly (breaks encapsulation)
            - If accessed directly, they become part of Public API
            - Only use if sure that no behavior (methods) will ever be attached to the variables (@dataclass?)
            - Changing implementation will be problematic:
                - Converting stored attribute -> computed attribute. Want to be able to store attributes instead of recomputing each time
            - No internal implementation
        2. Methods (ideal)
            - Two methods to respect encapsulation
                - Getter: Allows Access
                - Setter: Allows Setting or Mutation

- Implementation
    1. Making attributes Non-public
        - use underscore in variable name (_name)
    2. Writing Getter and Setter Methods for each attribute
        - In a class, __init__ is the attributes (variables)

```
# Making Getters and Setter methods
# PRO: Adheres to Encapsulation since get/set methods is used to change the attributes instead of direct access
# PRO: Adheres to a faux Information Hiding since converting temperature to _temperature, but no
private/protected vars in Python
# CON: Has more code to change


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())

>> 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.
```

Property Class

```
# 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
```

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.

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

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)
```

Here,

- `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.

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 code 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)

>>Setting value...
>>Getting value...
>>37
>>Getting value...
>>98.60000000000001.
>>Setting value...
>>Traceback (most recent call last):
>>  File "", line 29, in
>>  File "", line 4, in __init__
>>  File "", line 18, in temperature
```

```
>>ValueError: Temperature below -273 is not possible
```

Revision #5
Created 30 March 2024 07:17:02 by victor
Updated 30 March 2024 22:07:07 by victor