## Python Activity 26: Properties

### Learning Objectives
Students will be able to:

**Content:**
- Define **setters** and **getters**
- Define **property** and **setter decorators**
- Explain the different ways to access instance attributes

**Process:**
- Write code that creates a property and calls that method.
- Write code that creates a setter-decorated method and calls it.

**Prior Knowledge**
- Python concepts from Activities 1-20. Classes. 28 (Arguments)

*Folks, this is a brand new activity. If you encounter any issues/typos, please let Iris know!*

### Critical Thinking Questions:

1. Examine the following code below, that defines a class to store the temperature in degree Celsius.

   ```python
   class Celsius:
   __slots__ = ['_temperature']
   def fahrenheit(self):
       return (self._temperature * 1.8) + 32
   
   if __name__ == '__main__':
       humanC = Celsius()
       humanC._temperature = 37
   ```

   a. What are the instance attribute(s) of a Celsius object?
   
   b. On what line do we instantiate a new Celsius object? __________

   c. On what line do we assign a value to `humanC`’s instance attributes? __________

   d. Write a line of code to change `humanC`’s `temperature` to 0:

   e. Write a line of code to print the instance attribute(s) of this Celsius object:

---

**FYI:** In Python, we use an underscore at the start of an object name to indicate that we don’t want external modules or programmers to access these objects. In this class, we call that **underscore guilt** because you should feel like you’re doing something wrong when using classes, variables, or functions that start with an underscore.
2. Examine the sample code to store a temperature in degrees Fahrenheit.

```
class Fahrenheit:
    __slots__ = ['_temperature']
    def celsius(self):
        return (self._temperature - 32) / 1.8
    def temperature(self):
        return self._temperature
    def setTemperature(self, v):
        self._temperature = v

if __name__ == '__main__':
    humanF = Fahrenheit()
    humanF.setTemperature(98.6)
```

a. What are the instance attribute(s) of a Fahrenheit object?

b. On what line do we assign a value to humanF’s instance attributes?

c. How does assigning values to instance attributes differ for this Fahrenheit class, as compared to the Celsius class in the previous question?

d. Write a line of code to change humanF’s temperature to 32:

e. Write a line of code to print the instance attribute(s) of this Fahrenheit object:

f. How do you know a line of code is changing a value, versus referencing a value?

**FYI:** Getter methods, such as `temperature()` in this example, allow access to objects that we would normally protect with underscore guilt. Setter methods, such as `setTemperature(v)` in this example, provide a way to modify the value of an attribute without using the underscored name. Getters and setters, implemented in this way, are not good python programming.
3. Examine the sample code below, it stores temperature in Kelvin.

```python
class Kelvin:
    __slots__ = ['_temperature']
    def celsius(self):
        return self._temperature - 273.15
    @temperature.setter
    def temperature(self, v):
        self._temperature = v

if __name__ == '__main__':
    humanK = Kelvin()
    humanK.temperature = 310.15
```

a. What are the instance attribute(s) of a Kelvin object?

b. On what line do we assign a value to humanK’s instance attributes? __________

c. How does assigning values to instance attributes differ for this Kelvin class, as compared to the Fahrenheit class in the previous question?

 d. Write a line of code to change humanK’s temperature to 278.15:

 e. How did we modify the _temperature, in the first three questions for these classes:
   Celsius.py: ________________________________
   Fahrenheit.py: ________________________________
   Kelvin.py: ________________________________

FYI: We can implement setter methods in a more pythonic way, using the @functionName.setter decorator. The decorator is the line just prior to the function header that begins with an '@'. The text between the '@' and '.setter' is the name of the function below, and allows us to modify the values of instance attributes. The specified setter function is called when its name appears on the lefthand side of an assignment statement.

f. Why might it be preferable to use the setter decorator instead of modifying instance attributes directly?

_______________________________________________________________________
4. Examine the sample code below, it is the same as the previous Kelvin class, but with lines a-e added:

```python
class Kelvin:
    __slots__ = ['_temperature']
    def celsius(self):
        return self._temperature - 273.15
    @temperature.setter
    def temperature(self, v):
        self._temperature = v
    @property
    def temperature(self):
        return self._temperature

if __name__ == '__main__':
    humanK = Kelvin()
    humanK.temperature = 310.15
    print(humanK.temperature)
    print(humanK.celsius())
```

a. On what line do we reference the values stored in `humanK`'s `_temperature` attribute?
   __________

b. What might line d output?
   _______________________________________________________________________

c. How does referencing values stored in instance attributes differ for this Kelvin class, as compared to the Fahrenheit class?
   _______________________________________________________________________

d. How did we access the `_temperature`, in this activity’s questions for these classes:
   Celsius.py: ___________________________________________________________________
   Fahrenheit.py: ___________________________________________________________________
   Kelvin.py: ___________________________________________________________________

FYI: When we place the `@property` decorator before a function heading, the method is called when its name is referenced for accessing a value (i.e., not on the lefthand side of an assignment).

f. What does line e output? __________

g. If we added the following line right before line 2, what other code would we have to change: `@property`
   _______________________________________________________________________

h. Why might it be preferable to use the property decorator instead of referencing instance attributes directly?
   _______________________________________________________________________
   _______________________________________________________________________
5. Examine the following code, a new example!

```python
class Automobile:
    __slots__ = ['_make', '_model']

    @property
def brand(self):
        return self._make

    @brand.setter
def brand(self, m):
        self._make = m

if __name__ == '__main__':
dreamCar = Automobile()
dreamCar.brand = 'Tesla'
print(dreamCar.brand)
```

a. What are the instance attribute(s) of an Automobile object?

b. On what line do we instantiate a new Automobile object? ____________

c. What is stored in dreamCar’s instance attributes at the end? ____________

d. What are we trying to do on line 10? __________________________________

e. What method is called on line 10? ________________________________

f. What are we trying to do on line 11? ________________________________

g. What method is called on line 11? ________________________________

h. Write a @setter method for the _model instance attribute:

________________________________

_______________________________________________________________________
_______________________________________________________________________

i. How would you call this setter? _______________________________________

j. Write a @property method for the _model instance attribute:

_______________________________________________________________________
_______________________________________________________________________
_______________________________________________________________________

_______________________________________________________________________

k. How would you call this property? ____________________________________
Application Questions: Use Python to check your work

1. a. Write a class, `Currency`, that has the instance attribute `_usd`, representing the currency in US Dollars. `_usd` should be implemented as a property:

```python
class Currency():
    # Implementation
```

1. b. Add a `@setter` method to the previously defined class, `Currency`, that allows the user to modify the value of `_usd`:

```python
    # Implementation
```

1. c. Add a pair of `property` and `@setter` methods to reference and modify the instance attribute using any other recognized currency (you’ll need to apply a conversion rate):

```python
    # Implementation
```

Write a class, `Course`, that has the instance attribute `_grade`, which is a string and should also be implemented as a property. It has a setter method that takes in a decimal number representing the percentage grade in the course. Depending on that percentage, the `@.setter` method will assign an appropriate letter grade to the instance attribute:

```python
class Course():
    # Implement the property and setter method here.
```

3. Review lab assignments and Homeworks for more applications of properties and setters.