Name:_____ **Partner:** _____

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)

If you encounter any issues/typos, please let Iris know! Questions? Ask Iris or the POGILing forum

Critical Thinking Questions:

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

Celsius.py

```
0 class Celsius:
1     __slots__ = ['_temperature']
2     def fahrenheit(self):
3           return (self._temperature * 1.8 ) + 32

4 if __name__ == '__main__':
5     humanC = Celsius()
6     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.

Fahrenheit.py

```
0 class Fahrenheit:
1     __slots__ = ['_temperature']
2     def celsius(self):
3          return (self._temperature - 32 ) / 1.8
5     def temperature(self):
6          return self._temperature
7     def setTemperature(self, v):
8          self._temperature = v

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

d

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: ***Accessor*** methods, such as `temperature()` in this example, allow access to objects that we would normally protect with underscore guilt. ***Mutator*** methods, such as `setTemperature(v)` in this example, provide a way to modify the value of an attribute without using the underscored name. Accessors and mutators, implemented in this way, are not good python programming.

3. Examine the sample code below, it stores temperature in Kelvin.

Kelvin.py

```
0 class Kelvin:
1     __slots__ = ['_temperature']
2     def celsius(self):
3           return self._temperature – 273.15
4     @temperature.setter
5     def temperature(self,v):
6           self._temperature = v

7 if __name__ == '__main__':
8     humanK = Kelvin()
9     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:

Kelvin.py

```
0 class Kelvin:
1     __slots__ = ['_temperature']
2     def celsius(self):
3         return self._temperature – 273.15
4     @property
5     def temperature(self):
6         return self._temperature
a     @temperature.setter
b     def temperature(self, v):
c         self._temperature = v


7 if __name__ == '__main__':
8     humanK = Kelvin()
9     print(humanK.temperature)
d     humanK.temperature = 310.15
e     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:` _____

f. **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).

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!

```
0 class Automobile:
1     __slots__ = ['_make', '_model']
2     @property
3     def brand(self):
4           return self._make
5     @brand.setter
6     def brand(self, m):
7           self._make = m

8 if __name__ == '__main__':
9     dreamCar = Automobile()
10    dreamCar.brand = 'Tesla'
11    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`:

```
class Currency():
```

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

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):

2. 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:

```
class Course():
```

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