Built-in Decorators

WE'LL COVER THE FOLLOWING

Python comes with several built-in decorators. The big three are:

- @classmethod
- @staticmethod
- @property

There are also decorators in various parts of Python's standard library. One example would be **functools.wraps**. We will be limiting our scope to the three above though.

@classmethod and @staticmethod

I have never actually used these myself, so I did a fair bit of research. The <**@classmethod*>* decorator can be called with with an instance of a class or directly by the class itself as its first argument. According to the Python documentation: *It can be called either on the class (such as C.f()) or on an instance (such as C().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument*. The primary use case of a @classmethod decorator that I have found in my research is as an alternate constructor or helper method for initialization.

The <**@staticmethod*>* decorator is just a function inside of a class. You can call it both with and without instantiating the class. A typical use case is when

you have a function where you believe it has a connection with a class. It's a stylistic choice for the most part.

It might help to see a code example of how these two decorators work:

```
class DecoratorTest(object):
    """
    Test regular method vs @classmethod vs @staticmethod
    """

    def __init__(self):
        """Constructor"""
        pass

    def doubler(self, x):
        """"""
        print("running doubler")
        return x*2

    @classmethod
    def class_tripler(klass, x):
        """"""
        print("running tripler: %s" % klass)
        return x*3

    @staticmethod
    def static_quad(x):
        """"""
        print("running quad")
        return x*4

if __name__ == "__main__":
    decor = DecoratorTest()
    print(decor.doubler(5))
    print(decor.class_tripler(3))
    print(DecoratorTest.class_tripler(3))
    print(DecoratorTest.static_quad(2))
    print(decor.static_quad(3))

    print(decor.doubler)
    print(decor.class_tripler)
    print(decor.static_quad)
```

This example demonstrates that you can call a regular method and both decorated methods in the same way. You will notice that you can call both the @classmethod and the @staticmethod decorated functions directly from the class or from an instance of the class. If you try to call a regular function with the class (i.e. DecoratorTest.doubler(2)) you will receive a **TypeError**. You will also note that the last print statement shows that decor.static_quad returns a regular function instead of a bound method.

Python Properties

Python has a neat little concept called a property that can do several useful things. We will be looking into how to do the following:

> - Convert class methods into read-only attributes
> - Reimplement setters and getters into an attribute

One of the simplest ways to use a property is to use it as a decorator of a method. This allows you to turn a class method into a class attribute. I find this useful when I need to do some kind of combination of values. Others have found it useful for writing conversion methods that they want to have access to as methods. Let's take a look at a simple example:

```
class Person(object):
    """"""

    def __init__(self, first_name, last_name):
        """Constructor"""
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        """
        Return the full name
        """
        return "%s %s" % (self.first_name, self.last_name)
```

In the code above, we create two class attributes or properties: **self.first_name** and **self.last_name**. Next we create a **full_name** method that has a <**@property*>* decorator attached to it. This allows us to do the following in an interpreter session:

```
person = Person("Mike", "Driscoll")
print(person.full_name)
#'Mike Driscoll'

print(person.first_name)
#'Mike'

person.full_name = "Jackalope"
#Traceback (most recent call last):
#  File "/usercode/__ed_file.py", line 23, in <module>
#    person.full_name = "Jackalope"
#AttributeError: can't set attribute
```

As you can see, because we turned the method into a property, we can access it using normal dot notation. However, if we try to set the property to something different, we will cause an **AttributeError** to be raised. The only way to change the **full_name** property is to do so indirectly:

```
person = Person("Mike", "Driscoll")

person.first_name = "Dan"
print(person.full_name)
#'Dan Driscoll'
```

This is kind of limiting, so let's look at another example where we can make a property that does allow us to set it.