CMU MSP 36602 Python Decorators April 29, 2019

1. **Decorators** are a confusing, but useful and often used feature of Python. They are used to add specific features to a function. In this sense, they are a form of "code injection".
2. **Background**
   a. In Python, functions are objects like any other objects.
   b. Functions can be arguments to other functions.
   c. Code-injecting functions can be written like this example (from Decorators.py), which makes a function more verbose (tells the user when it is entered and exited, for debugging purposes).

```
def counter(n):
    for i in range(n):
        print(i)
    return 2*n

print("Calling counter()")
print(counter(4))


# Make a "wrapper" function that adds enter/exit to
# any function.
def verbose(fun):
    def new_fun(*args, **kwargs):
        print("entering {}".format(fun.__name__))
        rtn = fun(*args, **kwargs)
        print("exiting {}".format(fun.__name__))
        return rtn
    return new_fun


# Demonstrate using the wrapper on counter()
counter = verbose(counter)
print("\nCalling the wrapped counter()")
print(counter(4))
```

3. The **decorator syntax**:
   Placing `@my_decorator` before the definition of a function called "spam" is the same as defining "spam", then running `spam = my_decorator(spam)`. This is called "syntactic sugar". You can use `@my_decorator(arg)` for `spam = my_decorator(spam, arg)`.

```
# Demonstrate the same thing using the "decorator"
# syntax (simpler and ? clearer)
@verbose
def counter(n):
    for i in range(n):
        print(i)
    return 2*n


print("\nCalling the decorated counter()")
print(counter(4))
```

4. **Argument checking example**

   Imaging you have many functions that have the same arguments and the arguments need to be checked. Writing a decorator is a DRY solution.

```
from functools import wraps

# Argument checking decorator example
# Check that 'result' is a dict and 'method' is a str.
def arg_check(fun):
    @wraps(fun)
    def checked_fun(result, method):
        if not isinstance(result, dict):
            raise TypeError("'result' must be a 'dict'")
        if not isinstance(method, str):
            raise TypeError("'method' must be a 'str'")
        return fun(result, method)
    return checked_fun


@arg_check
def summarize(result, method):
    min_v = min(result.values())
    max_v = max(result.values())
    counts = dict(zip(range(min_v, max_v+1),
                      [0 for _ in range(min_v, max_v+1)]))
    for k in result.keys():
        counts[result[k]] += 1
    return counts

print("\nCalling summarize correctly")
print(summarize({'a': 1, 'b': 1, 'c': 2, 'd': 0}, "hclust"))
summarize(3, 3)
print("\nCalling summarize in correctly")
```

5. Some **other possible uses** for decorator functions
   a. Timing
   b. Logging
   c. Thread locking
   d. Checking if the user is logged in
6. The **`property()`**(built-in) function and the **`@property`** (built-in) decorator
   a. Note that, in a class, outside of a function definition, assignment without "self" creates an attribute for the instance.
   b. Assignment of the form `my_prop = property(fget=None, fset=None, fdel=None, fdoc=None)` creates a special "property" attribute called "my_prop". When Python sees a code of the form `my_val = my_object.my_prop`, it checks if "my_prop" is a property attribute, and if so it calls `my_val = fget(my_prop)`. Similarly, attempting to set a property attribute will result in the `fset()` function being called, and attempting to delete a property attribute causes `fdel()` to be called. This can be used to enforce a valid set of values for an attribute, or to prevent deletion, or to make the attribute read-only (by having the `fset()` method raise an exception.

c. Example:

```
class C(object):
    def __init__(self, x=None):
        self._x = x  # "hidden" _x property

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value
        #  or raise AttributeError("read only!")

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "'x' property")
```

d. Example using the built-in property decorator:

```
class C(object):
    def __init__(self, x=None):
        self._x = x

    @property
    def getx(self):
        return self._x

    @x.setter
    def setx(self, value):
        self._x = value
        #  or raise AttributeError("read only!")

    @x.deleter
    def delx(self):
        del self._x
```

Note that `@property` must be doing some magic stuff including defining the `x.setter` and `x.deleter` decorators.

7. Other built-in decorators

a. `@staticmethod` changes the way the method function defined below it is run. It prevents Python from passing "self" as the first argument. So when you define static methods in a class, you call them using the `my_instance.my_function()` syntax, but the function never gets to see any instance level variables. If your class needs "utility" functions that are related to class functionality, but do not use instance data, they should be decorated with `@staticmethod` when they are defined. An example would be one or more distance functions in a class implementing clustering.

b. `@classmethod` similarly changes the way the function is called by adding the class name as the first argument instead of "self". This allows the function to be called with the syntax `MyClass.my_function()` in addition to the `my_instance.my_function()` syntax.