Python Decorators

Python is rich with powerful features and expressive syntax. One of the favorites is decorators. In the context of design patterns, decorators dynamically alter the functionality of a function, method or class without having to directly use subclasses. This is ideal when you need to extend the functionality of functions that you don't want to modify. We can implement the decorator pattern anywhere, but Python facilitates the implementation by providing much more expressive features and syntax for that.

Essentially, decorators work as wrappers, modifying the behavior of the code before and after a target function execution, without the need to modify the function itself, augmenting the original functionality, thus decorating it.

What we need to know about functions

Assign functions to variables

```
def greet(name):
    return "hello "+name


greet_someone = greet
print greet_someone("John")

# Outputs: hello John
```

Define functions inside other functions

```
def greet(name):
    def get_message():
        return "Hello "

    result = get_message()+name
    return result

print greet("John")


# Outputs: Hello John
```

Functions can be passed as parameters to other functions

```
def greet(name):
   return "Hello " + name
```

```
def call_func(func):
    other_name = "John"
    return func(other_name)

print call_func(greet)

# Outputs: Hello John
```

Functions can return other functions

```
def compose_greet_func():
    def get_message():
        return "Hello there!"

    return get_message

greet = compose_greet_func()
print greet()

# Outputs: Hello there!
# Just thing of this as functions generating other functions.
```

Inner functions have access to the enclosing scope

More commonly known as a **closure**. A very powerful pattern that we will come across while building decorators. Another thing to note, Python only allows read access to the outer scope and not assignment. Notice how we modified the example above to read a "**name**" argument from the enclosing scope of the inner function and return the new function.

```
def compose_greet_func(name):
    def get_message():
        return "Hello there "+name+"!"

    return get_message

greet = compose_greet_func("John")
print greet()

# Outputs: Hello there John!
```

Composition of Decorators

Function decorators are simply wrappers to existing functions. Putting the ideas mentioned above together, we can build a decorator. In this example let's consider a function that wraps the string output of another function by p tags.

```
def get_text(name):
   return "lorem ipsum, {0} dolor sit amet".format(name)

def p_decorate(func):
   def func_wrapper(name):
       return "<p>{0}</p>".format(func(name))
   return func_wrapper

my_get_text = p_decorate(get_text)

print my_get_text("John")

# <p>Outputs lorem ipsum, John dolor sit amet</p>
```

That was our first decorator. A function that takes another function as an argument, generates a new function, augmenting the work of the original function, and returning the generated function so we can use it anywhere. To have get_text itself be decorated by p_decorate, we just have to assign get_text to the result of p_decorate.

```
get_text = p_decorate(get_text)

print get_text("John")

# Outputs lorem ipsum, John dolor sit amet
```

Another thing to notice is that our decorated function takes a name argument. All what we had to do in the decorator is to let the wrapper of get_text pass that argument.

Python's Decorator Syntax

Python makes creating and using decorators a bit cleaner and nicer for the programmer through some syntactic sugar To decorate get_text we don't have to get_text = p_decorator(get_text) There is a neat shortcut for that, which is to mention the name of the decorating function before the function to be decorated. The name of the decorator should be perpended with an **@** symbol.

```
def p_decorate(func):
   def func_wrapper(name):
       return "<p>{0}</p>".format(func(name))
   return func_wrapper

@p_decorate
def get_text(name):
   return "lorem ipsum, {0} dolor sit amet".format(name)

print get_text("John")

# Outputs <p>lorem ipsum, John dolor sit amet</p>
```

Now let's consider we wanted to decorate our get_text function by 2 other functions to wrap a div and strong tag around the string output.

```
def p_decorate(func):
   def func_wrapper(name):
       return "<p>{0}</p>".format(func(name))
   return func_wrapper

def strong_decorate(func):
    def func_wrapper(name):
        return "<strong>{0}</strong>".format(func(name))
    return func_wrapper

def div_decorate(func):
    def func_wrapper(name):
        return "<div>{0}</div>".format(func(name))
    return func_wrapper
```

With the basic approach, decorating get_text would be along the lines of

```
get_text = div_decorate(p_decorate(strong_decorate(get_text)))
```

With Python's decorator syntax, same thing can be achieved with much more expressive power.

```
@div_decorate
@p_decorate
@strong_decorate
def get_text(name):
   return "lorem ipsum, {0} dolor sit amet".format(name)

print get_text("John")

# Outputs <div><p><strong>lorem ipsum, John dolor sit
amet</strong></p></div>
```

One important thing to notice here is that the order of setting our decorators matters. If the order was different in the example above, the output would have been different.

Decorating Methods

In Python, methods are functions that expect their first parameter to be a reference to the current object. We can build decorators for methods the same way, while taking self into consideration in the wrapper function.

```
def p_decorate(func):
   def func_wrapper(self):
       return "<p>{0}</p>".format(func(self))
```

```
    return func_wrapper

class Person(object):
    def __init__(self):
        self.name = "John"
        self.family = "Doe"

    @p_decorate
    def get_fullname(self):
        return self.name+" "+self.family

my_person = Person()
print my_person.get_fullname()
```

A much better approach would be to make our decorator useful for functions and methods alike. This can be done by putting *args and **kwargs as parameters for the wrapper, then it can accept any arbitrary number of arguments and keyword arguments.

```
def p_decorate(func):
   def func_wrapper(*args, **kwargs):
       return "<p>{0}</p>".format(func(*args, **kwargs))
   return func_wrapper

class Person(object):
    def __init__(self):
        self.name = "John"
        self.family = "Doe"

    @p_decorate
    def get_fullname(self):
        return self.name+" "+self.family

my_person = Person()

print my_person.get_fullname()
```

Passing arguments to decorators

Looking back at the example before the one above, you can notice how redundant the decorators in the example are. 3 decorators(div_decorate, p_decorate, strong_decorate) each with the same functionality but wrapping the string with different tags. We can definitely do much better than that. Why not have a more general implementation for one that takes the tag to wrap with as a string?

```
def tags(tag_name):
    def tags_decorator(func):
        def func_wrapper(name):
            return "<{0}>{1}</{0}>".format(tag_name, func(name))
```

```
            return func_wrapper
        return tags_decorator

@tags("p")
def get_text(name):
    return "Hello "+name

print get_text("John")

# Outputs <p>Hello John</p>
```

It took a bit more work in this case. Decorators expect to receive a function as an argument, that is why we will have to build a function that takes those extra arguments and generate our decorator on the fly. In the example above tags, is our decorator generator.