Decorators

--created by David Song

Contents

The Basics 1

Getting the arguments 2

Dynamic decorator 3

Mimic Java AOP Interceptor 4

Summary 4

The Basics

Decorators modify functions. More specifically, a decorator is a function **that transforms another function**.

When you use a decorator, Python passes the decorated function -- we'll call this the *target function* -- to the *decorator function*, and replaces it with the result. Without decorators, it would look something like this:

```
def decorator_function(target):
    # Do something with the target function
    target.attribute = 1
    return target

def target(a,b):
    return a + b

# This is what the decorator actually does
target = decorator_function(target)
```

The code below has the same effect, put the *decorator function*'s name, prefaced with a `@`, on the line before the *target function* definition. Python internally will transform the *target* by applying the decorator to it and replacing it with the returned value.

```
@decorator_function
def target(a,b):
   return a + b

t=target(1,2)
print t                  # will print 3
print target.attribute   # will print 1
```

The *decorator function* can return absolutely **anything**, and Python will replace the *target function* with that return value. If the returned value is not a function, then it is not a callable and cannot be called. A decorator, by its name, means to intercept the call, doing some meaningful stuff, and returns a function having the same signature (same parameters as in Python) as the target function (the function to be decorated).

Getting the arguments

Since the returned *wrapper function* replaces the *target function*, the *wrapper function* will **receive the arguments** intended for the *target function*. Assuming you want your decorator to work for any *target function*, your *wrapper function* then should accept arbitrary non-keyword arguments and arbitrary keyword arguments, add, remove, or modify arguments if necessary, and pass the arguments to the *target function*.

```
def decorator(target):
    def wrapper(*args, **kwargs):
        #
        # Edit the keyword arguments -- here,
        # enable debug mode no matter what
        #
        kwargs.update({'debug': True})

        print 'Calling function "%s" with arguments %s and keyword arguments %s' %
(target.__name__, args, kwargs)                              # 2nd print


        return target(*args, **kwargs)

    wrapper.attribute = 1
    print 'wrapper.attribute 1:', wrapper.attribute          # 1st print

    return wrapper


@decorator
def target(a, b, debug=False):
    if debug: print '[Debug] I am the target function'       # 3rd print
    return a + b


target(1, 2)
print 'target.attribute: "%s"' % target.attribute            # 4th print


target(1, 2, debug=True)

output:
wrapper.attribute 1: 1
Calling function "target" with arguments (1, 2) and keyword arguments
{'debug': True}
[Debug] I am the target function
target.attribute: "1"
```

Note that, to take parameters, the decorator has 2 layers; the outer takes the target as the parameter, the inner (wrapper) takes the actual parameters of the decorated function (target), and return the target, and the outer returns the wrapper. The **point** is: a function can be decorated in many layers, but only the function finally gets returned does the trick, and the inner function can see all the outer functions parameters.

Dynamic decorator

Sometimes you might want to customize behavior by passing arbitrary options to your *decorator function*. Below example shows how to change target behavior by passing a debug level value to the target:

```
def options(debug_level):

     def decorator(target):

         def wrapper(*args, **kwargs):
             # Edit the keyword arguments
             # here, set debug level to whatever specified in the options
             kwargs.update({'debug_level': debug_level})
             retval = target(*args, **kwargs)
             print 'wrapper, after calling target'
             return retval
         return wrapper
     return decorator


@options(5)
def target(a, b, debug_level=0):
     if debug_level:
         print '[Debug Level %s] I am the target function' % debug_level
     return a+b

print target(1,2)
```

How to use the classic way to use this 3 layers decoration without use the @ notation? The answer is wrap the function twice:

```
#Two layers wraping does the same trick as @ annotation

target1=options('aha')

target=target1(target)

print target(1,2)
```

The trick is assign a fun f to the one resides right inside the decorator, and then uses this f to get the one inside the decorator one layer deep, until it reaches the target.

Note that, the above definition won't have any effect, but `@options(5)` in front of the target function actually calls the option function and its inner function decorator, causes executions or output if there are any.

Mimic Java AOP Interceptor

In real world software development, there are many circumstances you will need to apply a function wrap around another method in many places, such as apply logging information before and after calling a function. In Java, this can be achieved via AOP. Below code illustrate the AOP interception behavior in Python:

```
import logging
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s (%(threadName)-2s) %(message)s',
                    )

def before_after(target):
    def wrapper(*args, **kwargs):
        logging.info('Before %s start working', target.__name__)
        result = target(*args, **kwargs)
        logging.info('After %s finished work', target.__name__)
        return result

    return wrapper

@before_after
def worker(a, b):
    logging.info('worker is doing work')
    return a+b


w = worker(1,2)
logging.info('result %d', w)
```

Summary

A decorator is a function that transforms another function. From one point of view, it is a wrapper that intercepts the call to the decorated (target) function, does useful works and returns another function. The returned function should have the same signature as the decorated function to make it callable and meaningful, although the decorator can return anything.

The decorator function can be nested many layers as needed. Each outer layer returns its inner layer function, and the inner most function returns the target.

The decorator can use the `@` annotation in front of the decorated function or use the classic way to assign the decorator which takes the decorated fun as the parameter to the decorated one. For multi layers decoration, one `@` annotation is sufficient. To achieve the same from the classic way, we will need multiple assignments, layer by layer, and each time gets an inner function until reaches the inner target.