

Decorators in Python

Matt Johnson

Decorators are a powerful feature of the Python programming language. They prove their usefulness in many common programming scenarios. Logging the execution time of a function, or tracking the number of times a function is called can be elegantly handled with decorators. In web applications, decorators can be added to functions to enforce access control restrictions. Multithreaded Python programs can exploit decorators to synchronize functions. Decorators can be used as type checking mechanisms in Python's dynamically typed environment. As you will see later, decorators can even be used to add memoization capabilities to a function. But all these capabilities can be achieved without decorators, so why use them in the first place? The real power of decorators comes from their ability to modify the behavior of a function *without* changing any of the code inside the function.

Decorators are functions that wrap around other functions. This allows decorators to execute code before and after a function is called. A decorator takes a function object as an argument (which is called the *decoratee*) and returns a new function object that will be executed in its place. The function object constructed inside the decorator usually calls the decoratee. A decorator is executed exactly once, at run time, *for each* function it decorates. The function object returned by the decorator is executed in place of the decoratee every time the decoratee is called (except, of course, in the decorator itself).

Consider a decorator that adds memoization capabilities to a function:

```
def memoize(fn): # fn is the function object of the decoratee
    cache = {}
    def wrapper(*args, **kwargs):
        # assume the first positional argument to the decoratee is an
        # appropriate key for the cache
        n = args[0]

        if n in cache:
            return cache[n]

        # Call the decoratee and cache its result
        result = fn(*args, **kwargs)
        cache[n] = result
        return result

    # return the function object that was just created to replace the
    # decoratee
    return wrapper
```

Memoize is defined exactly like a normal function in Python. It takes a single argument, a function object. Memoize initializes a dictionary object and, because Python supports closures, this variable will be bound for the life of the inner function, wrapper. Wrapper performs the simple memorization procedure by checking to see if the value has already been calculated, and calling the decoratee if needed. For readers unfamiliar with Python, the special `*args`, `**kwargs` variables are worthy of an explanation. In the interests of brevity, a complete one will not be provided here. Put simply, `*args`

represent the positional arguments to the decoratee and ****kwargs** represent the keyword arguments to the decoratee. They can be accessed like an array and dictionary, respectively.

Decorating a function is easy. Simply add the name of the decorator to the top of a function, prepended with an "@". Python automatically calls the decorator, passing the appropriate function object, and replaces every call to the decoratee with the function returned by the decorator. The following example shows how to use the previously defined memoize decorator:

```
@memoize
def factorial(n):
    if n == 0 or n == 1:
        return 1

    return factorial(n - 1) * n

@memoize
def fib(n):
    if n == 0 or n == 1:
        return n

    return fib(n - 1) + fib(n - 2)

# Compute in linear time!
print fib(10)
print factorial(8)

# Only one calculation is needed to compute the next fibonacci and factorial
# because the cache is preserved
print fib(11)
print factorial(9)
```

Decorators are a useful tool in Python because they allow the programmer to modify the behavior of a function without touching any of the code inside the function. Decorators are advantageous when code needs to be injected before and after a function. Logging, access control checks, synchronization, type checking, memoization and many other tasks are perfectly suited to use decorators. Decorators are simply syntactic sugar because everything they do can be achieved through other means. But when used wisely, decorators can improve the readability and maintainability of a program while eliminating duplicate code.