CHAPTER 7

Function Decorators and Closures

> There's been a number of complaints about the choice of the name "decorator" for this feature. The major one is that the name is not consistent with its use in the GoF book.[1] The name *decorator* probably owes more to its use in the compiler area—a syntax tree is walked and annotated.
>
> — PEP 318 — Decorators for Functions and Methods

Function decorators let us "mark" functions in the source code to enhance their behavior in some way. This is powerful stuff, but mastering it requires understanding closures.

One of the newest reserved keywords in Python is `nonlocal`, introduced in Python 3.0. You can have a profitable life as a Python programmer without ever using it if you adhere to a strict regimen of class-centered object orientation. However, if you want to implement your own function decorators, you must know closures inside out, and then the need for `nonlocal` becomes obvious.

Aside from their application in decorators, closures are also essential for effective asynchronous programming with callbacks, and for coding in a functional style whenever it makes sense.

The end goal of this chapter is to explain exactly how function decorators work, from the simplest registration decorators to the rather more complicated parameterized ones. However, before we reach that goal we need to cover:

- How Python evaluates decorator syntax
- How Python decides whether a variable is local
- Why closures exist and how they work

1. That's the 1995 *Design Patterns* book by the so-called Gang of Four.

- What problem is solved by `nonlocal`

With this grounding, we can tackle further decorator topics:

- Implementing a well-behaved decorator
- Interesting decorators in the standard library
- Implementing a parameterized decorator

We start with a very basic introduction to decorators, and then proceed with the rest of the items listed here.

Decorators 101

A decorator is a callable that takes another function as argument (the decorated function).[2] The decorator may perform some processing with the decorated function, and returns it or replaces it with another function or callable object.

In other words, assuming an existing decorator named `decorate`, this code:

```
@decorate
def target():
    print('running target()')
```

Has the same effect as writing this:

```
def target():
    print('running target()')

target = decorate(target)
```

The end result is the same: at the end of either of these snippets, the `target` name does not necessarily refer to the original `target` function, but to whatever function is returned by `decorate(target)`.

To confirm that the decorated function is replaced, see the console session in Example 7-1.

Example 7-1. A decorator usually replaces a function with a different one

```
>>> def deco(func):
...     def inner():
...         print('running inner()')
...     return inner  ❶
...
>>> @deco
... def target():  ❷
```

2. Python also supports class decorators. They are covered in Chapter 21.

```
...     print('running target()')
...
>>> target()  ❸
running inner()
>>> target  ❹
<function deco.<locals>.inner at 0x10063b598>
```

❶ `deco` returns its `inner` function object.

❷ `target` is decorated by `deco`.

❸ Invoking the decorated `target` actually runs `inner`.

❹ Inspection reveals that `target` is a now a reference to `inner`.

Strictly speaking, decorators are just syntactic sugar. As we just saw, you can always simply call a decorator like any regular callable, passing another function. Sometimes that is actually convenient, especially when doing *metaprogramming*—changing program behavior at runtime.

To summarize: the first crucial fact about decorators is that they have the power to replace the decorated function with a different one. The second crucial fact is that they are executed immediately when a module is loaded. This is explained next.

When Python Executes Decorators

A key feature of decorators is that they run right after the decorated function is defined. That is usually at *import time* (i.e., when a module is loaded by Python). Consider *registration.py* in Example 7-2.

Example 7-2. The registration.py module

```
registry = []  ❶

def register(func):  ❷
    print('running register(%s)' % func)  ❸
    registry.append(func)  ❹
    return func  ❺

@register  ❻
def f1():
    print('running f1()')

@register
def f2():
    print('running f2()')

def f3():  ❼
    print('running f3()')

def main():  ❽
```

```
    print('running main()')
    print('registry ->', registry)
    f1()
    f2()
    f3()

if __name__=='__main__':
    main()   ❾
```

❶ `registry` will hold references to functions decorated by `@register`.

❷ `register` takes a function as argument.

❸ Display what function is being decorated, for demonstration.

❹ Include `func` in `registry`.

❺ Return `func`: we must return a function; here we return the same received as argument.

❻ `f1` and `f2` are decorated by `@register`.

❼ `f3` is not decorated.

❽ `main` displays the `registry`, then calls `f1()`, `f2()`, and `f3()`.

❾ `main()` is only invoked if *registration.py* runs as a script.

The output of running *registration.py* as a script looks like this:

```
$ python3 registration.py
running register(<function f1 at 0x100631bf8>)
running register(<function f2 at 0x100631c80>)
running main()
registry -> [<function f1 at 0x100631bf8>, <function f2 at 0x100631c80>]
running f1()
running f2()
running f3()
```

Note that `register` runs (twice) before any other function in the module. When `register` is called, it receives as an argument the function object being decorated—for example, `<function f1 at 0x100631bf8>`.

After the module is loaded, the `registry` holds references to the two decorated functions: `f1` and `f2`. These functions, as well as `f3`, are only executed when explicitly called by `main`.

If *registration.py* is imported (and not run as a script), the output is this:

```
>>> import registration
running register(<function f1 at 0x10063b1e0>)
running register(<function f2 at 0x10063b268>)
```

At this time, if you look at the `registry`, here is what you get:

```
>>> registration.registry
[<function f1 at 0x10063b1e0>, <function f2 at 0x10063b268>]
```

The main point of Example 7-2 is to emphasize that function decorators are executed as soon as the module is imported, but the decorated functions only run when they are explicitly invoked. This highlights the difference between what Pythonistas call *import time* and *runtime*.

Considering how decorators are commonly employed in real code, Example 7-2 is unusual in two ways:

- The decorator function is defined in the same module as the decorated functions. A real decorator is usually defined in one module and applied to functions in other modules.
- The `register` decorator returns the same function passed as argument. In practice, most decorators define an inner function and return it.

Even though the `register` decorator in Example 7-2 returns the decorated function unchanged, that technique is not useless. Similar decorators are used in many Python web frameworks to add functions to some central registry—for example, a registry mapping URL patterns to functions that generate HTTP responses. Such registration decorators may or may not change the decorated function. The next section shows a practical example.

Decorator-Enhanced Strategy Pattern

A registration decorator is a good enhancement to the ecommerce promotional discount from "Case Study: Refactoring Strategy" on page 168.

Recall that our main issue with Example 6-6 is the repetition of the function names in their definitions and then in the `promos` list used by the `best_promo` function to determine the highest discount applicable. The repetition is problematic because someone may add a new promotional strategy function and forget to manually add it to the `promos` list—in which case, `best_promo` will silently ignore the new strategy, introducing a subtle bug in the system. Example 7-3 solves this problem with a registration decorator.

Example 7-3. The promos list is filled by the promotion decorator

```
promos = []  ❶

def promotion(promo_func):  ❷
    promos.append(promo_func)
    return promo_func

@promotion  ❸
def fidelity(order):
    """5% discount for customers with 1000 or more fidelity points"""
```

```
    return order.total() * .05 if order.customer.fidelity >= 1000 else 0

@promotion
def bulk_item(order):
    """10% discount for each LineItem with 20 or more units"""
    discount = 0
    for item in order.cart:
        if item.quantity >= 20:
            discount += item.total() * .1
    return discount

@promotion
def large_order(order):
    """7% discount for orders with 10 or more distinct items"""
    distinct_items = {item.product for item in order.cart}
    if len(distinct_items) >= 10:
        return order.total() * .07
    return 0

def best_promo(order):   ❹
    """Select best discount available
    """
    return max(promo(order) for promo in promos)
```

❶ The `promos` list starts empty.

❷ `promotion` decorator returns `promo_func` unchanged, after adding it to the `promos` list.

❸ Any function decorated by `@promotion` will be added to `promos`.

❹ No changes needed to `best_promos`, because it relies on the `promos` list.

This solution has several advantages over the others presented in "Case Study: Refactoring Strategy" on page 168:

- The promotion strategy functions don't have to use special names (i.e., they don't need to use the `_promo` suffix).
- The `@promotion` decorator highlights the purpose of the decorated function, and also makes it easy to temporarily disable a promotion: just comment out the decorator.
- Promotional discount strategies may be defined in other modules, anywhere in the system, as long as the `@promotion` decorator is applied to them.

Most decorators do change the decorated function. They usually do it by defining an inner function and returning it to replace the decorated function. Code that uses inner functions almost always depends on closures to operate correctly. To understand clo-

sures, we need to take a step back a have a close look at how variable scopes work in Python.

Variable Scope Rules

In Example 7-4, we define and test a function that reads two variables: a local variable `a`, defined as function parameter, and variable `b` that is not defined anywhere in the function.

Example 7-4. Function reading a local and a global variable

```
>>> def f1(a):
...     print(a)
...     print(b)
...
>>> f1(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f1
NameError: global name 'b' is not defined
```

The error we got is not surprising. Continuing from Example 7-4, if we assign a value to a global `b` and then call `f1`, it works:

```
>>> b = 6
>>> f1(3)
3
6
```

Now, let's see an example that may surprise you.

Take a look at the `f2` function in Example 7-5. Its first two lines are the same as `f1` in Example 7-4, then it makes an assignment to `b`, and prints its value. But it fails at the second `print`, before the assignment is made.

Example 7-5. Variable b is local, because it is assigned a value in the body of the function

```
>>> b = 6
>>> def f2(a):
...     print(a)
...     print(b)
...     b = 9
...
>>> f2(3)
3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f2
UnboundLocalError: local variable 'b' referenced before assignment
```

Note that the output starts with `3`, which proves that the `print(a)` statement was executed. But the second one, `print(b)`, never runs. When I first saw this I was surprised, thinking that `6` should be printed, because there is a global variable `b` and the assignment to the local `b` is made after `print(b)`.

But the fact is, when Python compiles the body of the function, it decides that `b` is a local variable because it is assigned within the function. The generated bytecode reflects this decision and will try to fetch `b` from the local environment. Later, when the call `f2(3)` is made, the body of `f2` fetches and prints the value of the local variable `a`, but when trying to fetch the value of local variable `b` it discovers that `b` is unbound.

This is not a bug, but a design choice: Python does not require you to declare variables, but assumes that a variable assigned in the body of a function is local. This is much better than the behavior of JavaScript, which does not require variable declarations either, but if you do forget to declare that a variable is local (with `var`), you may clobber a global variable without knowing.

If we want the interpreter to treat `b` as a global variable in spite of the assignment within the function, we use the `global` declaration:

```
>>> def f3(a):
...     global b
...     print(a)
...     print(b)
...     b = 9
...
>>> f3(3)
3
6
>>> b
9

>>> f3(3)
a = 3
b = 8
b = 30
>>> b
30
>>>
```

After this closer look at how variable scopes work in Python, we can tackle closures in the next section, "Closures" on page 192. If you are curious about the bytecode differences between the functions in Examples 7-4 and 7-5, see the following sidebar.

Comparing Bytecodes

The `dis` module provides an easy way to disassemble the bytecode of Python functions. Read Examples 7-6 and 7-7 to see the bytecodes for `f1` and `f2` from Examples 7-4 and 7-5.

Example 7-6. Disassembly of the f1 function from Example 7-4

```
>>> from dis import dis
>>> dis(f1)
  2           0 LOAD_GLOBAL              0 (print)  ❶
              3 LOAD_FAST                0 (a)  ❷
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP

  3          10 LOAD_GLOBAL              0 (print)
             13 LOAD_GLOBAL              1 (b)  ❸
             16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             19 POP_TOP
             20 LOAD_CONST               0 (None)
             23 RETURN_VALUE
```

❶ Load global name `print`.

❷ Load local name `a`.

❸ Load global name `b`.

Contrast the bytecode for `f1` shown in Example 7-6 with the bytecode for `f2` in Example 7-7.

Example 7-7. Disassembly of the f2 function from Example 7-5

```
>>> dis(f2)
  2           0 LOAD_GLOBAL              0 (print)
              3 LOAD_FAST                0 (a)
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 POP_TOP

  3          10 LOAD_GLOBAL              0 (print)
             13 LOAD_FAST                1 (b)  ❶
             16 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             19 POP_TOP

  4          20 LOAD_CONST               1 (9)
             23 STORE_FAST               1 (b)
             26 LOAD_CONST               0 (None)
             29 RETURN_VALUE
```

❶ Load *local* name `b`. This shows that the compiler considers `b` a local variable, even if the assignment to `b` occurs later, because the nature of the variable—whether it is local or not—cannot change the body of the function.

The CPython VM that runs the bytecode is a stack machine, so the operations `LOAD` and `POP` refer to the stack. It is beyond the scope of this book to further describe the Python opcodes, but they are documented along with the `dis` module in dis — Disassembler for Python bytecode.

Closures

In the blogosphere, closures are sometimes confused with anonymous functions. The reason why many confuse them is historic: defining functions inside functions is not so common, until you start using anonymous functions. And closures only matter when you have nested functions. So a lot of people learn both concepts at the same time.

Actually, a closure is a function with an extended scope that encompasses nonglobal variables referenced in the body of the function but not defined there. It does not matter whether the function is anonymous or not; what matters is that it can access nonglobal variables that are defined outside of its body.

This is a challenging concept to grasp, and is better approached through an example.

Consider an `avg` function to compute the mean of an ever-increasing series of values; for example, the average closing price of a commodity over its entire history. Every day a new price is added, and the average is computed taking into account all prices so far.

Starting with a clean slate, this is how `avg` could be used:

```
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Where does `avg` come from, and where does it keep the history of previous values?

For starters, Example 7-8 is a class-based implementation.

Example 7-8. average_oo.py: A class to calculate a running average

```
class Averager():

    def __init__(self):
        self.series = []

    def __call__(self, new_value):
```

```
        self.series.append(new_value)
        total = sum(self.series)
        return total/len(self.series)
```

The `Averager` class creates instances that are callable:

```
>>> avg = Averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Now, Example 7-9 is a functional implementation, using the higher-order function `make_averager`.

Example 7-9. average.py: A higher-order function to calculate a running average

```
def make_averager():
    series = []

    def averager(new_value):
        series.append(new_value)
        total = sum(series)
        return total/len(series)

    return averager
```

When invoked, `make_averager` returns an `averager` function object. Each time an `averager` is called, it appends the passed argument to the series, and computes the current average, as shown in Example 7-10.

Example 7-10. Testing Example 7-9

```
>>> avg = make_averager()
>>> avg(10)
10.0
>>> avg(11)
10.5
>>> avg(12)
11.0
```

Note the similarities of the examples: we call `Averager()` or `make_averager()` to get a callable object `avg` that will update the historical series and calculate the current mean. In Example 7-8, `avg` is an instance of `Averager`, and in Example 7-9 it is the inner function, `averager`. Either way, we just call `avg(n)` to include `n` in the series and get the updated mean.

It's obvious where the `avg` of the `Averager` class keeps the history: the `self.series` instance attribute. But where does the `avg` function in the second example find the `series`?

Note that `series` is a local variable of `make_averager` because the initialization `series = []` happens in the body of that function. But when `avg(10)` is called, `make_averager` has already returned, and its local scope is long gone.

Within `averager`, `series` is a *free variable*. This is a technical term meaning a variable that is not bound in the local scope. See Figure 7-1.

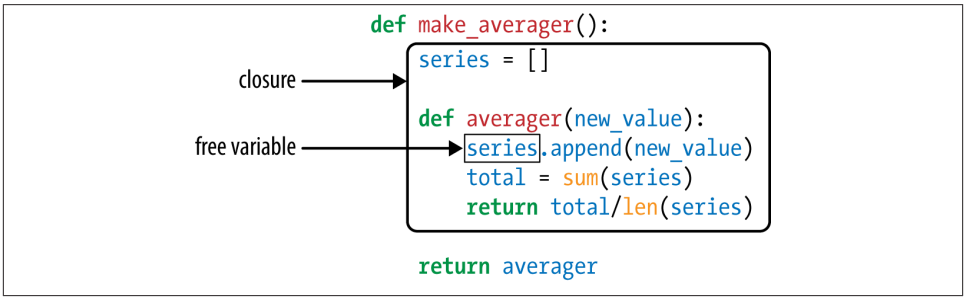

Figure 7-1. The closure for averager extends the scope of that function to include the binding for the free variable series.

Inspecting the returned `averager` object shows how Python keeps the names of local and free variables in the `__code__` attribute that represents the compiled body of the function. Example 7-11 demonstrates.

Example 7-11. Inspecting the function created by make_averager in Example 7-9

```
>>> avg.__code__.co_varnames
('new_value', 'total')
>>> avg.__code__.co_freevars
('series',)
```

The binding for `series` is kept in the `__closure__` attribute of the returned function `avg`. Each item in `avg.__closure__` corresponds to a name in `avg.__code__.co_freevars`. These items are `cells`, and they have an attribute called `cell_contents` where the actual value can be found. Example 7-12 shows these attributes.

Example 7-12. Continuing from Example 7-10

```
>>> avg.__code__.co_freevars
('series',)
>>> avg.__closure__
(<cell at 0x107a44f78: list object at 0x107a91a48>,)
>>> avg.__closure__[0].cell_contents
[10, 11, 12]
```

To summarize: a closure is a function that retains the bindings of the free variables that exist when the function is defined, so that they can be used later when the function is invoked and the defining scope is no longer available.

Note that the only situation in which a function may need to deal with external variables that are nonglobal is when it is nested in another function.

The nonlocal Declaration

Our previous implementation of `make_averager` was not efficient. In Example 7-9, we stored all the values in the historical series and computed their `sum` every time `averager` was called. A better implementation would just store the total and the number of items so far, and compute the mean from these two numbers.

Example 7-13 is a broken implementation, just to make a point. Can you see where it breaks?

Example 7-13. A broken higher-order function to calculate a running average without keeping all history

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        count += 1
        total += new_value
        return total / count

    return averager
```

If you try Example 7-13, here is what you get:

```
    >>> avg = make_averager()
    >>> avg(10)
    Traceback (most recent call last):
      ...
    UnboundLocalError: local variable 'count' referenced before assignment
    >>>
```

The problem is that the statement `count += 1` actually means the same as `count = count + 1`, when `count` is a number or any immutable type. So we are actually assigning to `count` in the body of `averager`, and that makes it a local variable. The same problem affects the `total` variable.

We did not have this problem in Example 7-9 because we never assigned to the `series` name; we only called `series.append` and invoked `sum` and `len` on it. So we took advantage of the fact that lists are mutable.

But with immutable types like numbers, strings, tuples, etc., all you can do is read, but never update. If you try to rebind them, as in `count = count + 1`, then you are implicitly creating a local variable `count`. It is no longer a free variable, and therefore it is not saved in the closure.

To work around this, the `nonlocal` declaration was introduced in Python 3. It lets you flag a variable as a free variable even when it is assigned a new value within the function. If a new value is assigned to a `nonlocal` variable, the binding stored in the closure is changed. A correct implementation of our newest `make_averager` looks like Example 7-14.

Example 7-14. Calculate a running average without keeping all history (fixed with the use of nonlocal)

```
def make_averager():
    count = 0
    total = 0

    def averager(new_value):
        nonlocal count, total
        count += 1
        total += new_value
        return total / count

    return averager
```

Getting by without nonlocal in Python 2

The lack of `nonlocal` in Python 2 requires workarounds, one of which is described in the third code snippet of PEP 3104 — Access to Names in Outer Scopes, which introduced `nonlocal`. Essentially the idea is to store the variables the inner functions need to change (e.g., `count`, `total`) as items or attributes of some mutable object, like a `dict` or a simple instance, and bind that object to a free variable.

Now that we have Python closures covered, we can effectively implement decorators with nested functions.

Implementing a Simple Decorator

Example 7-15 is a decorator that clocks every invocation of the decorated function and prints the elapsed time, the arguments passed, and the result of the call.

Example 7-15. A simple decorator to output the running time of functions

```
import time
```

```
def clock(func):
    def clocked(*args):  # ❶
        t0 = time.perf_counter()
        result = func(*args)  # ❷
        elapsed = time.perf_counter() - t0
        name = func.__name__
        arg_str = ', '.join(repr(arg) for arg in args)
        print('[%0.8fs] %s(%s) -> %r' % (elapsed, name, arg_str, result))
        return result
    return clocked  # ❸
```

❶ Define inner function `clocked` to accept any number of positional arguments.

❷ This line only works because the closure for `clocked` encompasses the `func` free variable.

❸ Return the inner function to replace the decorated function.

Example 7-16 demonstrates the use of the `clock` decorator.

Example 7-16. Using the clock decorator

```
# clockdeco_demo.py

import time
from clockdeco import clock

@clock
def snooze(seconds):
    time.sleep(seconds)

@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

if __name__=='__main__':
    print('*' * 40, 'Calling snooze(.123)')
    snooze(.123)
    print('*' * 40, 'Calling factorial(6)')
    print('6! =', factorial(6))
```

The output of running Example 7-16 looks like this:

```
$ python3 clockdeco_demo.py
**************************************** Calling snooze(123)
[0.12405610s] snooze(.123) -> None
**************************************** Calling factorial(6)
[0.00000191s] factorial(1) -> 1
[0.00004911s] factorial(2) -> 2
[0.00008488s] factorial(3) -> 6
[0.00013208s] factorial(4) -> 24
[0.00019193s] factorial(5) -> 120
```

```
[0.00026107s] factorial(6) -> 720
6! = 720
```

How It Works

Remember that this code:

```
@clock
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)
```

Actually does this:

```
def factorial(n):
    return 1 if n < 2 else n*factorial(n-1)

factorial = clock(factorial)
```

So, in both examples, `clock` gets the `factorial` function as its `func` argument (see Example 7-15). It then creates and returns the `clocked` function, which the Python interpreter assigns to `factorial` behind the scenes. In fact, if you import the `clockdeco_demo` module and check the `__name__` of `factorial`, this is what you get:

```
>>> import clockdeco_demo
>>> clockdeco_demo.factorial.__name__
'clocked'
>>>
```

So `factorial` now actually holds a reference to the `clocked` function. From now on, each time `factorial(n)` is called, `clocked(n)` gets executed. In essence, `clocked` does the following:

1. Records the initial time `t0`.
2. Calls the original factorial, saving the result.
3. Computes the elapsed time.
4. Formats and prints the collected data.
5. Returns the result saved in step 2.

This is the typical behavior of a decorator: it replaces the decorated function with a new function that accepts the same arguments and (usually) returns whatever the decorated function was supposed to return, while also doing some extra processing.

In *Design Patterns* by Gamma et al., the short description of the Decorator pattern starts with: "Attach additional responsibilities to an object dynamically." Function decorators fit that description. But at the implementation level, Python decorators bear little resemblance to the classic Decorator described in the original *Design Patterns* work. "Soapbox" on page 213 has more on this subject.

The `clock` decorator implemented in Example 7-15 has a few shortcomings: it does not support keyword arguments, and it masks the `__name__` and `__doc__` of the decorated function. Example 7-17 uses the `functools.wraps` decorator to copy the relevant attributes from `func` to `clocked`. Also, in this new version, keyword arguments are correctly handled.

Example 7-17. An improved clock decorator

```
# clockdeco2.py

import time
import functools

def clock(func):
    @functools.wraps(func)
    def clocked(*args, **kwargs):
        t0 = time.time()
        result = func(*args, **kwargs)
        elapsed = time.time() - t0
        name = func.__name__
        arg_lst = []
        if args:
            arg_lst.append(', '.join(repr(arg) for arg in args))
        if kwargs:
            pairs = ['%s=%r' % (k, w) for k, w in sorted(kwargs.items())]
            arg_lst.append(', '.join(pairs))
        arg_str = ', '.join(arg_lst)
        print('[%0.8fs] %s(%s) -> %r ' % (elapsed, name, arg_str, result))
        return result
    return clocked
```

`functools.wraps` is just one of the ready-to-use decorators in the standard library. In the next section, we'll meet two of the most impressive decorators that `functools` provides: `lru_cache` and `singledispatch`.

Decorators in the Standard Library

Python has three built-in functions that are designed to decorate methods: `property`, `classmethod`, and `staticmethod`. We will discuss `property` in "Using a Property for

Attribute Validation" on page 604 and the others in "classmethod Versus staticmethod" on page 252.

Another frequently seen decorator is `functools.wraps`, a helper for building well-behaved decorators. We used it in Example 7-17. Two of the most interesting decorators in the standard library are `lru_cache` and the brand-new `singledispatch` (added in Python 3.4). Both are defined in the `functools` module. We'll cover them next.

Memoization with functools.lru_cache

A very practical decorator is `functools.lru_cache`. It implements memoization: an optimization technique that works by saving the results of previous invocations of an expensive function, avoiding repeat computations on previously used arguments. The letters LRU stand for Least Recently Used, meaning that the growth of the cache is limited by discarding the entries that have not been read for a while.

A good demonstration is to apply `lru_cache` to the painfully slow recursive function to generate the *n*th number in the Fibonacci sequence, as shown in Example 7-18.

Example 7-18. The very costly recursive way to compute the nth number in the Fibonacci series

```
from clockdeco import clock

@clock
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__=='__main__':
    print(fibonacci(6))
```

Here is the result of running *fibo_demo.py*. Except for the last line, all output is generated by the `clock` decorator:

```
    $ python3 fibo_demo.py
    [0.00000095s] fibonacci(0) -> 0
    [0.00000095s] fibonacci(1) -> 1
    [0.00007892s] fibonacci(2) -> 1
    [0.00000095s] fibonacci(1) -> 1
    [0.00000095s] fibonacci(0) -> 0
    [0.00000095s] fibonacci(1) -> 1
    [0.00003815s] fibonacci(2) -> 1
    [0.00007391s] fibonacci(3) -> 2
    [0.00018883s] fibonacci(4) -> 3
    [0.00000000s] fibonacci(1) -> 1
    [0.00000095s] fibonacci(0) -> 0
    [0.00000119s] fibonacci(1) -> 1
    [0.00004911s] fibonacci(2) -> 1
```

```
[0.00009704s] fibonacci(3) -> 2
[0.00000000s] fibonacci(0) -> 0
[0.00000000s] fibonacci(1) -> 1
[0.00002694s] fibonacci(2) -> 1
[0.00000095s] fibonacci(1) -> 1
[0.00000095s] fibonacci(0) -> 0
[0.00000095s] fibonacci(1) -> 1
[0.00005102s] fibonacci(2) -> 1
[0.00008917s] fibonacci(3) -> 2
[0.00015593s] fibonacci(4) -> 3
[0.00029993s] fibonacci(5) -> 5
[0.00052810s] fibonacci(6) -> 8
8
```

The waste is obvious: `fibonacci(1)` is called eight times, `fibonacci(2)` five times, etc. But if we just add two lines to use `lru_cache`, performance is much improved. See Example 7-19.

Example 7-19. Faster implementation using caching

```
import functools

from clockdeco import clock

@functools.lru_cache() # ❶
@clock  # ❷
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-2) + fibonacci(n-1)

if __name__=='__main__':
    print(fibonacci(6))
```

❶ Note that `lru_cache` must be invoked as a regular function—note the parentheses in the line: `@functools.lru_cache()`. The reason is that it accepts configuration parameters, as we'll see shortly.

❷ This is an example of stacked decorators: `@lru_cache()` is applied on the function returned by `@clock`.

Execution time is halved, and the function is called only once for each value of `n`:

```
$ python3 fibo_demo_lru.py
[0.00000119s] fibonacci(0) -> 0
[0.00000119s] fibonacci(1) -> 1
[0.00010800s] fibonacci(2) -> 1
[0.00000787s] fibonacci(3) -> 2
[0.00016093s] fibonacci(4) -> 3
[0.00001216s] fibonacci(5) -> 5
[0.00025296s] fibonacci(6) -> 8
```

In another test, to compute `fibonacci(30)`, Example 7-19 made the 31 calls needed in 0.0005s, while the uncached Example 7-18 called `fibonacci` 2,692,537 times and took 17.7 seconds in an Intel Core i7 notebook.

Besides making silly recursive algorithms viable, `lru_cache` really shines in applications that need to fetch information from the Web.

It's important to note that `lru_cache` can be tuned by passing two optional arguments. Its full signature is:

```
functools.lru_cache(maxsize=128, typed=False)
```

The `maxsize` argument determines how many call results are stored. After the cache is full, older results are discarded to make room. For optimal performance, `maxsize` should be a power of 2. The `typed` argument, if set to `True`, stores results of different argument types separately, i.e., distinguishing between float and integer arguments that are normally considered equal, like `1` and `1.0`. By the way, because `lru_cache` uses a `dict` to store the results, and the keys are made from the positional and keyword arguments used in the calls, all the arguments taken by the decorated function must be *hashable*.

Now let's consider the intriguing `functools.singledispatch` decorator.

Generic Functions with Single Dispatch

Imagine we are creating a tool to debug web applications. We want to be able to generate HTML displays for different types of Python objects.

We could start with a function like this:

```
import html

def htmlize(obj):
    content = html.escape(repr(obj))
    return '<pre>{}</pre>'.format(content)
```

That will work for any Python type, but now we want to extend it to generate custom displays for some types:

- `str`: replace embedded newline characters with `'<br>\n'` and use `<p>` tags instead of `<pre>`.
- `int`: show the number in decimal and hexadecimal.
- `list`: output an HTML list, formatting each item according to its type.

The behavior we want is shown in Example 7-20.

Example 7-20. htmlize generates HTML tailored to different object types

```
>>> htmlize({1, 2, 3})   ❶
'<pre>{1, 2, 3}</pre>'
```

```
>>> htmlize(abs)
'<pre>&lt;built-in function abs&gt;</pre>'
>>> htmlize('Heimlich & Co.\n- a game')   ❷
'<p>Heimlich &amp; Co.<br>\n- a game</p>'
>>> htmlize(42)   ❸
'<pre>42 (0x2a)</pre>'
>>> print(htmlize(['alpha', 66, {3, 2, 1}]))   ❹
<ul>
<li><p>alpha</p></li>
<li><pre>66 (0x42)</pre></li>
<li><pre>{1, 2, 3}</pre></li>
</ul>
```

❶ By default, the HTML-escaped `repr` of an object is shown enclosed in `<pre></pre>`.

❷ `str` objects are also HTML-escaped but wrapped in `<p></p>` with `<br>` line breaks.

❸ An `int` is shown in decimal and hexadecimal, inside `<pre></pre>`.

❹ Each list item is formatted according to its type, and the whole sequence rendered as an HTML list.

Because we don't have method or function overloading in Python, we can't create variations of `htmlize` with different signatures for each data type we want to handle differently. A common solution in Python would be to turn `htmlize` into a dispatch function, with a chain of `if/elif/elif` calling specialized functions like `htmlize_str`, `htmlize_int`, etc. This is not extensible by users of our module, and is unwieldy: over time, the `htmlize` dispatcher would become too big, and the coupling between it and the specialized functions would be very tight.

The new `functools.singledispatch` decorator in Python 3.4 allows each module to contribute to the overall solution, and lets you easily provide a specialized function even for classes that you can't edit. If you decorate a plain function with `@singledispatch`, it becomes a *generic function*: a group of functions to perform the same operation in different ways, depending on the type of the first argument.[3] Example 7-21 shows how.

`functools.singledispatch` was added in Python 3.4, but the `singledispatch` package available on PyPI is a backport compatible with Python 2.6 to 3.3.

3. This is what is meant by the term single-dispatch. If more arguments were used to select the specific functions, we'd have multiple-dispatch.

Example 7-21. singledispatch creates a custom htmlize.register to bundle several functions into a generic function

```
from functools import singledispatch
from collections import abc
import numbers
import html

@singledispatch    ❶
def htmlize(obj):
    content = html.escape(repr(obj))
    return '<pre>{}</pre>'.format(content)

@htmlize.register(str)    ❷
def _(text):              ❸
    content = html.escape(text).replace('\n', '<br>\n')
    return '<p>{0}</p>'.format(content)

@htmlize.register(numbers.Integral)    ❹
def _(n):
    return '<pre>{0} (0x{0:x})</pre>'.format(n)

@htmlize.register(tuple)    ❺
@htmlize.register(abc.MutableSequence)
def _(seq):
    inner = '</li>\n<li>'.join(htmlize(item) for item in seq)
    return '<ul>\n<li>' + inner + '</li>\n</ul>'
```

❶ `@singledispatch` marks the base function that handles the `object` type.

❷ Each specialized function is decorated with `@«base_function».register(«type»)`.

❸ The name of the specialized functions is irrelevant; _ is a good choice to make this clear.

❹ For each additional type to receive special treatment, register a new function. `numbers.Integral` is a virtual superclass of `int`.

❺ You can stack several `register` decorators to support different types with the same function.

When possible, register the specialized functions to handle ABCs (abstract classes) such as `numbers.Integral` and `abc.MutableSequence` instead of concrete implementations like `int` and `list`. This allows your code to support a greater variety of compatible types. For example, a Python extension can provide alternatives to the `int` type with fixed bit lengths as subclasses of `numbers.Integral`.

Using ABCs for type checking allows your code to support existing or future classes that are either actual or virtual subclasses of those ABCs. The use of ABCs and the concept of a virtual subclass are subjects of Chapter 11.

A notable quality of the `singledispatch` mechanism is that you can register specialized functions anywhere in the system, in any module. If you later add a module with a new user-defined type, you can easily provide a new custom function to handle that type. And you can write custom functions for classes that you did not write and can't change.

`singledispatch` is a well-thought-out addition to the standard library, and it offers more features than we can describe here. The best documentation for it is PEP 443 — Single-dispatch generic functions.

`@singledispatch` is not designed to bring Java-style method overloading to Python. A single class with many overloaded variations of a method is better than a single function with a lengthy stretch of `if/elif/elif/elif` blocks. But both solutions are flawed because they concentrate too much responsibility in a single code unit—the class or the function. The advantage of `@singledispatch` is supporting modular extension: each module can register a specialized function for each type it supports.

Decorators are functions and therefore they may be composed (i.e., you can apply a decorator to a function that is already decorated, as shown in Example 7-21). The next section explains how that works.

Stacked Decorators

Example 7-19 demonstrated the use of stacked decorators: `@lru_cache` is applied on the result of `@clock` over `fibonacci`. In Example 7-21, the `@htmlize.register` decorator was applied twice to the last function in the module.

When two decorators `@d1` and `@d2` are applied to a function `f` in that order, the result is the same as `f = d1(d2(f))`.

In other words, this:

```
@d1
@d2
def f():
    print('f')
```

Is the same as:

```
    def f():
        print('f')

    f = d1(d2(f))
```

Besides stacked decorators, this chapter has shown some decorators that take arguments, for example, `@lru_cache()` and the `htmlize.register(«type»)` produced by `@singledispatch` in Example 7-21. The next section shows how to build decorators that accept parameters.

Parameterized Decorators

When parsing a decorator in source code, Python takes the decorated function and passes it as the first argument to the decorator function. So how do you make a decorator accept other arguments? The answer is: make a decorator factory that takes those arguments and returns a decorator, which is then applied to the function to be decorated. Confusing? Sure. Let's start with an example based on the simplest decorator we've seen: `register` in Example 7-22.

Example 7-22. Abridged registration.py module from Example 7-2, repeated here for convenience

```
registry = []

def register(func):
    print('running register(%s)' % func)
    registry.append(func)
    return func

@register
def f1():
    print('running f1()')

print('running main()')
print('registry ->', registry)
f1()
```

A Parameterized Registration Decorator

In order to make it easy to enable or disable the function registration performed by `register`, we'll make it accept an optional `active` parameter which, if `False`, skips registering the decorated function. Example 7-23 shows how. Conceptually, the new `register` function is not a decorator but a decorator factory. When called, it returns the actual decorator that will be applied to the target function.

Example 7-23. To accept parameters, the new register decorator must be called as a function

```
registry = set()    ❶

def register(active=True):    ❷
    def decorate(func):    ❸
        print('running register(active=%s)->decorate(%s)'
              % (active, func))
        if active:    ❹
            registry.add(func)
        else:
            registry.discard(func)    ❺

        return func    ❻
    return decorate    ❼

@register(active=False)    ❽
def f1():
    print('running f1()')

@register()    ❾
def f2():
    print('running f2()')

def f3():
    print('running f3()')
```

❶ `registry` is now a `set`, so adding and removing functions is faster.

❷ `register` takes an optional keyword argument.

❸ The `decorate` inner function is the actual decorator; note how it takes a function as argument.

❹ Register `func` only if the `active` argument (retrieved from the closure) is `True`.

❺ If `not active` and `func in registry`, remove it.

❻ Because `decorate` is a decorator, it must return a function.

❼ `register` is our decorator factory, so it returns `decorate`.

❽ The `@register` factory must be invoked as a function, with the desired parameters.

❾ If no parameters are passed, `register` must still be called as a function—`@register()`—i.e., to return the actual decorator, `decorate`.

The main point is that `register()` returns `decorate`, which is then applied to the decorated function.

The code in Example 7-23 is in a *registration_param.py* module. If we import it, this is what we get:

```
>>> import registration_param
running register(active=False)->decorate(<function f1 at 0x10063c1e0>)
running register(active=True)->decorate(<function f2 at 0x10063c268>)
>>> registration_param.registry
[<function f2 at 0x10063c268>]
```

Note how only the `f2` function appears in the `registry`; `f1` does not appear because `active=False` was passed to the `register` decorator factory, so the `decorate` that was applied to `f1` did not add it to the `registry`.

If, instead of using the `@` syntax, we used `register` as a regular function, the syntax needed to decorate a function `f` would be `register()(f)` to add `f` to the `registry`, or `register(active=False)(f)` to not add it (or remove it). See Example 7-24 for a demo of adding and removing functions to the `registry`.

Example 7-24. Using the registration_param module listed in Example 7-23

```
>>> from registration_param import *
running register(active=False)->decorate(<function f1 at 0x10073c1e0>)
running register(active=True)->decorate(<function f2 at 0x10073c268>)
>>> registry  # ❶
{<function f2 at 0x10073c268>}
>>> register()(f3)  # ❷
running register(active=True)->decorate(<function f3 at 0x10073c158>)
<function f3 at 0x10073c158>
>>> registry  # ❸
{<function f3 at 0x10073c158>, <function f2 at 0x10073c268>}
>>> register(active=False)(f2)  # ❹
running register(active=False)->decorate(<function f2 at 0x10073c268>)
<function f2 at 0x10073c268>
>>> registry  # ❺
{<function f3 at 0x10073c158>}
```

❶ When the module is imported, `f2` is in the `registry`.

❷ The `register()` expression returns `decorate`, which is then applied to `f3`.

❸ The previous line added `f3` to the `registry`.

❹ This call removes `f2` from the `registry`.

❺ Confirm that only `f3` remains in the `registry`.

The workings of parameterized decorators are fairly involved, and the one we've just discussed is simpler than most. Parameterized decorators usually replace the decorated function, and their construction requires yet another level of nesting. Touring such function pyramids is our next adventure.

The Parameterized Clock Decorator

In this section, we'll revisit the `clock` decorator, adding a feature: users may pass a format string to control the output of the decorated function. See Example 7-25.

For simplicity, Example 7-25 is based on the initial `clock` implementation from Example 7-15, and not the improved one from Example 7-17 that uses `@functools.wraps`, adding yet another function layer.

Example 7-25. Module clockdeco_param.py: the parameterized clock decorator

```
import time

DEFAULT_FMT = '[{elapsed:0.8f}s] {name}({args}) -> {result}'

def clock(fmt=DEFAULT_FMT):   ❶
    def decorate(func):       ❷
        def clocked(*_args):  ❸
            t0 = time.time()
            _result = func(*_args)   ❹
            elapsed = time.time() - t0
            name = func.__name__
            args = ', '.join(repr(arg) for arg in _args)   ❺
            result = repr(_result)   ❻
            print(fmt.format(**locals()))   ❼
            return _result   ❽
        return clocked   ❾
    return decorate   ❿

if __name__ == '__main__':

    @clock()   ⓫
    def snooze(seconds):
        time.sleep(seconds)

    for i in range(3):
        snooze(.123)
```

❶ `clock` is our parameterized decorator factory.

❷ `decorate` is the actual decorator.

❸ `clocked` wraps the decorated function.

❹ `_result` is the actual result of the decorated function.

❺ `_args` holds the actual arguments of `clocked`, while `args` is `str` used for display.

❻ `result` is the `str` representation of `_result`, for display.

❼ Using `**locals()` here allows any local variable of `clocked` to be referenced in the `fmt`.

❽ `clocked` will replace the decorated function, so it should return whatever that function returns.

❾ `decorate` returns `clocked`.

❿ `clock` returns `decorate`.

⓫ In this self test, `clock()` is called without arguments, so the decorator applied will use the default format `str`.

If you run Example 7-25 from the shell, this is what you get:

```
$ python3 clockdeco_param.py
[0.12412500s] snooze(0.123) -> None
[0.12411904s] snooze(0.123) -> None
[0.12410498s] snooze(0.123) -> None
```

To exercise the new functionality, Examples 7-26 and 7-27 are two other modules using `clockdeco_param`, and the outputs they generate.

Example 7-26. clockdeco_param_demo1.py

```
import time
from clockdeco_param import clock

@clock('{name}: {elapsed}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

Output of Example 7-26:

```
$ python3 clockdeco_param_demo1.py
snooze: 0.12414693832397461s
snooze: 0.1241159439086914s
snooze: 0.12412118911743164s
```

Example 7-27. clockdeco_param_demo2.py

```
import time
from clockdeco_param import clock

@clock('{name}({args}) dt={elapsed:0.3f}s')
def snooze(seconds):
    time.sleep(seconds)

for i in range(3):
    snooze(.123)
```

Output of Example 7-27:

```
$ python3 clockdeco_param_demo2.py
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
snooze(0.123) dt=0.124s
```

This ends our exploration of decorators as far as space permits within the scope of this book. See "Further Reading" on page 212, in particular Graham Dumpleton's blog and `wrapt` module for industrial-strength techniques when building decorators.

Graham Dumpleton and Lennart Regebro—one of this book's technical reviewers—argue that decorators are best coded as classes implementing `__call__`, and not as functions like the examples in this chapter. I agree that approach is better for non-trivial decorators, but to explain the basic idea of this language feature, functions are easier to understand.

Chapter Summary

We covered a lot of ground in this chapter, but I tried to make the journey as smooth as possible even if the terrain is rugged. After all, we did enter the realm of metaprogramming.

We started with a simple `@register` decorator without an inner function, and finished with a parameterized `@clock()` involving two levels of nested functions.

Registration decorators, though simple in essence, have real applications in advanced Python frameworks. We applied the registration idea to an improvement of our Strategy design pattern refactoring from Chapter 6.

Parameterized decorators almost aways involve at least two nested functions, maybe more if you want to use `@functools.wraps` to produce a decorator that provides better support for more advanced techniques. One such technique is stacked decorators, which we briefly covered.

We also visited two awesome function decorators provided in the `functools` module of standard library: `@lru_cache()` and `@singledispatch`.

Understanding how decorators actually work required covering the difference between *import time* and *runtime*, then diving into variable scoping, closures, and the new `nonlocal` declaration. Mastering closures and `nonlocal` is valuable not only to build decorators, but also to code event-oriented programs for GUIs or asynchronous I/O with callbacks.

Further Reading

Chapter 9, "Metaprogramming," of the *Python Cookbook, Third Edition* by David Beazley and Brian K. Jones (O'Reilly), has several recipes from elementary decorators to very sophisticated ones, including one that can be called as a regular decorator or as a decorator factory, e.g., `@clock` or `@clock()`. That's "Recipe 9.6. Defining a Decorator That Takes an Optional Argument" in that cookbook.

Graham Dumpleton has a series of in-depth blog posts about techniques for implementing well-behaved decorators, starting with "How You Implemented Your Python Decorator is Wrong". His deep expertise in this matter is also nicely packaged in the wrapt module he wrote to simplify the implementation of decorators and dynamic function wrappers, which support introspection and behave correctly when further decorated, when applied to methods and when used as descriptors. (Descriptors are the subject of chapter Chapter 20.)

Michele Simionato authored a package aiming to "simplify the usage of decorators for the average programmer, and to popularize decorators by showing various non-trivial examples," according to the docs. It's available on PyPI as the decorator package.

Created when decorators were still a new feature in Python, the Python Decorator Library wiki page has dozens of examples. Because that page started years ago, some of the techniques shown have been superseded, but the page is still an excellent source of inspiration.

PEP 443 provides the rationale and a detailed description of the single-dispatch generic functions' facility. An old (March 2005) blog post by Guido van Rossum, "Five-Minute Multimethods in Python", walks through an implementation of generic functions (a.k.a. multimethods) using decorators. His code supports multiple-dispatch (i.e., dispatch based on more than one positional argument). Guido's multimethods code is interesting, but it's a didactic example. For a modern, production-ready implementation of multiple-dispatch generic functions, check out Reg by Martijn Faassen—author of the model-driven and REST-savvy Morepath web framework.

"Closures in Python" is a short blog post by Fredrik Lundh that explains the terminology of closures.

PEP 3104 — Access to Names in Outer Scopes describes the introduction of the `nonlocal` declaration to allow rebinding of names that are neither local nor global. It also includes an excellent overview of how this issue is resolved in other dynamic languages (Perl, Ruby, JavaScript, etc.) and the pros and cons of the design options available to Python.

On a more theoretical level, PEP 227 — Statically Nested Scopes documents the introduction of lexical scoping as an option in Python 2.1 and as a standard in Python 2.2,

explaining the rationale and design choices for the implementation of closures in Python.

Soapbox

The designer of any language with first-class functions faces this issue: being first-class objects, functions are defined in a certain scope but may be invoked in other scopes. The question is: how to evaluate the free variables? The first and simplest answer is "dynamic scope." This means that free variables are evaluated by looking into the environment where the function is invoked.

If Python had dynamic scope and no closures, we could improvise `avg`—similar to Example 7-9—like this:

```
>>> ### this is not a real Python console session! ###
>>> avg = make_averager()
>>> series = []  # ❶
>>> avg(10)
10.0
>>> avg(11)  # ❷
10.5
>>> avg(12)
11.0
>>> series = [1]  # ❸
>>> avg(5)
3.0
```

❶ Before using `avg`, we have to define `series = []` ourselves, so we must know that `averager` (inside `make_averager`) refers to a `list` by that name.

❷ Behind the scenes, `series` is used to accumulate the values to be averaged.

❸ When `series = [1]` is executed, the previous list is lost. This could happen by accident, when handling two independent running averages at the same time.

Functions should be black boxes, with their implementation hidden from users. But with dynamic scope, if a function uses free variables, the programmer has to know its internals to set up an environment where it works correctly.

On the other hand, dynamic scope is easier to implement, which is probably why it was the path taken by John McCarthy when he created Lisp, the first language to have first-class functions. Paul Graham's article "The Roots of Lisp" is an accessible explanation of John McCarthy's original paper about the Lisp language: "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". McCarthy's paper is a masterpiece as great as Beethoven's 9th Symphony. Paul Graham translated it for the rest of us, from mathematics to English and running code.

Paul Graham's commentary also shows how tricky dynamic scoping is. Quoting from "The Roots of Lisp":

> It's an eloquent testimony to the dangers of dynamic scope that even the very first example of higher-order Lisp functions was broken because of it. It may be that McCarthy was not fully aware of the implications of dynamic scope in 1960. Dynamic scope remained in Lisp implementations for a surprisingly long time—until Sussman and Steele developed Scheme in 1975. Lexical scope does not complicate the definition of `eval` very much, but it may make compilers harder to write.

Today, lexical scope is the norm: free variables are evaluated considering the environment where the function is defined. Lexical scope complicates the implementation of languages with first-class functions, because it requires the support of closures. On the other hand, lexical scope makes source code easier to read. Most languages invented since Algol have lexical scope.

For many years, Python `lambdas` did not provide closures, contributing to the bad name of this feature among functional-programming geeks in the blogosphere. This was fixed in Python 2.2 (December 2001), but the blogosphere has a long memory. Since then, `lambda` is embarrassing only because of its limited syntax.

Python Decorators and the Decorator Design Pattern

Python function decorators fit the general description of Decorator given by Gamma et al. in *Design Patterns*: "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."

At the implementation level, Python decorators do not resemble the classic Decorator design pattern, but an analogy can be made.

In the design pattern, `Decorator` and `Component` are abstract classes. An instance of a concrete decorator wraps an instance of a concrete component in order to add behaviors to it. Quoting from *Design Patterns*:

> The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients. The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities." (p. 175)

In Python, the decorator function plays the role of a concrete `Decorator` subclass, and the inner function it returns is a decorator instance. The returned function wraps the function to be decorated, which is analogous to the component in the design pattern. The returned function is transparent because it conforms to the interface of the component by accepting the same arguments. It forwards calls to the component and may perform additional actions either before or after it. Borrowing from the previous citation, we can adapt the last sentence to say that "Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added behaviors." That is what enable stacked decorators to work.

Note that I am not suggesting that function decorators should be used to implement the Decorator pattern in Python programs. Although this can be done in specific situations,

in general the Decorator pattern is best implemented with classes to represent the Decorator and the components it will wrap.