Decorators

What's a Decorator?

- Decoration is a way to specify management or augmentation code for functions and classes.
- Decorators themselves take the form of callable objects (e.g., functions) that process other callable objects.
- Function decorators, added in Python 2.4, do name rebinding at function definition time, providing a layer of logic that can manage functions and methods, or later calls to them.
- Class decorators, added in Python 2.6 and 3.0, do name rebinding at class definition time, providing a layer of logic that can manage classes, or the instances created by later calls to them.

What's a Decorator?

- In short, decorators provide a way to insert automatically run code at the end of function and class definition statements - at the end of a def for function decorators, and at the end of a class for class decorators.

Managing Calls and Instances

Proxies	Descriptions
Call proxies	Function decorators install wrapper objects to intercept later function calls and process them as needed, usually passing the call on to the original function to run the managed action.
Interface proxies	Class decorators install wrapper objects to intercept later instance creation calls and process them as required, usually passing the call on to the original class to create a managed instance.

Decorators achieve these effects by automatically rebinding function and class names to other callables, at the end of def and class statements. When later invoked, these callables can perform tasks such as tracing and timing function calls, managing access to class instance attributes, and so on.

Managing Functions and Classes

Managers	Descriptions
Function managers	Function decorators can also be used to manage function objects, instead of or in addition to later calls to them - to register a function to an API, for instance. Our primary focus here, though, will be on their more commonly used call wrapper application.
Class managers	Class decorators can also be used to manage class objects directly, instead of or in addition to instance creation calls - to augment a class with new methods, for example. Because this role intersects strongly with that of metaclasses, we'll see additional use cases in the next chapter. As we'll find, both tools run at the end of the class creation process, but class decorators often offer a lighter-weight solution.

Managing Functions and Classes

- In other words, function decorators can be used to manage both function calls and function objects, and class decorators can be used to manage both class instances and classes themselves.
- By returning the decorated object itself instead of a wrapper, decorators become a simple post-creation step for functions and classes.
- Regardless of the role they play, decorators provide a convenient and explicit way to code tools useful both during program development and in live production systems.

Using and Defining Decorators

- Python itself comes with built-in decorators that have specialized roles - static and class method declaration, property creation, and more.
- In addition, many popular Python toolkits include decorators to perform tasks such as managing database or user-interface logic.
- For more general tasks, programmers can code arbitrary decorators of their own.
- For example, function decorators may be used to augment functions with code that adds call tracing or logging, performs argument validity testing during debugging, automatically acquires and releases thread locks, times calls made to functions for optimization, and so on.

Why Decorators?

- Decorators have a very explicit syntax, which makes them easier to spot than helper function calls that may be arbitrarily far-removed from the subject functions or classes.
- Decorators are applied once, when the subject function or class is defined; it's not necessary to add extra code at every call to the class or function, which may have to be changed in the future.
- Because of both of the prior points, decorators make it less likely that a user of an API will forget to augment a function or class according to API requirements.

Function Decorators

- A function decorator is a kind of runtime declaration about the function whose definition follows.
- The decorator is coded on a line just before the def statement that defines a function or method, and it consists of the @ symbol followed by a reference to a metafunction - a function (or other callable object) that manages another function.

```
@decorator                  # Decorate function
def F(arg):
    ...

F(99)                       # Call function
```

```
def F(arg):
    ...
F = decorator(F)            # Rebind function name to decorator result

F(99)                       # Essentially calls decorator(F)(99)
```

Class Decorators

- Class decorators are strongly related to function decorators; in fact, they use the same syntax and very similar coding patterns.
- Rather than wrapping individual functions or methods, though, class decorators are a way to manage classes, or wrap up instance construction calls with extra logic that manages, or augments instances created from a class. In the latter role, they may manage full object interfaces.

```
@decorator                  # Decorate class
class C:
    ...

x = C(99)                   # Make an instance
```

```
class C:
    ...
C = decorator(C)            # Rebind class name to decorator result

x = C(99)                   # Essentially calls decorator(C)(99)
```

Decorator Nesting

- To support multiple nested steps of augmentation this way, decorator syntax allows you to add multiple layers of wrapper logic to a decorated function or method.
- When this feature is used, each decorator must appear on a line of its own. Decorator syntax of this form.

```
@A
@B
@C
def f(...):
    ...
```

```
def f(...):
    ...
f = A(B(C(f)))
```

Decorator Arguments

- Both function and class decorators can also seem to take arguments, although really these arguments are passed to a callable that in effect returns the decorator, which in turn returns a callable. By nature, this usually sets up multiple levels of state retention.

```
@decorator(A, B)
def F(arg):
    ...

F(99)
```

```
def F(arg):
    ...
F = decorator(A, B)(F)      # Rebind F to result of decorator's return value

F(99)                       # Essentially calls decorator(A, B)(F)(99)
```

Coding Function Decorators – Tracing Calls

```
# File decorator1.py

class tracer:
    def __init__(self, func):              # On @ decoration: save original func
        self.calls = 0
        self.func = func
    def __call__(self, *args):             # On later calls: run original func
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        self.func(*args)

@tracer
def spam(a, b, c):              # spam = tracer(spam)
    print(a + b + c)            # Wraps spam in a decorator object
```

Coding Function Decorators – Tracing Calls

```
>>> from decorator1 import spam

>>> spam(1, 2, 3)              # Really calls the tracer wrapper object
call 1 to spam
6

>>> spam('a', 'b', 'c')        # Invokes __call__ in class
call 2 to spam
abc

>>> spam.calls                 # Number calls in wrapper state information
2
>>> spam
<decorator1.tracer object at 0x02D9A730>
```

Nondecorator Equivalent

```
calls = 0
def tracer(func, *args):
    global calls
    calls += 1
    print('call %s to %s' % (calls, func.__name__))
    func(*args)

def spam(a, b, c):
    print(a, b, c)

>>> spam(1, 2, 3)              # Normal nontraced call: accidental?
1 2 3

>>> tracer(spam, 1, 2, 3)      # Special traced call without decorators
call 1 to spam
1 2 3
```

Coding Class Decorators – Singleton Classes

```
# 3.X and 2.X: global table

instances = {}

def singleton(aClass):                          # On @ decoration
    def onCall(*args, **kwargs):                # On instance creation
        if aClass not in instances:             # One dict entry per class
            instances[aClass] = aClass(*args, **kwargs)
        return instances[aClass]
    return onCall
```

```
@singleton                                  # Person = singleton(Person)
class Person:                               # Rebinds Person to onCall
     def __init__(self, name, hours, rate):  # onCall remembers Person
        self.name = name
        self.hours = hours
        self.rate = rate
     def pay(self):
        return self.hours * self.rate

@singleton                                  # Spam = singleton(Spam)
class Spam:                                 # Rebinds Spam to onCall
    def __init__(self, val):                # onCall remembers Spam
        self.attr = val

bob = Person('Bob', 40, 10)                 # Really calls onCall
print(bob.name, bob.pay())

sue = Person('Sue', 50, 20)                 # Same, single object
print(sue.name, sue.pay())

X = Spam(val=42)                            # One Person, one Spam
Y = Spam(99)
print(X.attr, Y.attr)
```

```
c:\code> python singletons.py
Bob 400
Bob 400
42 42
```