CHAPTER 39
Decorators

In Chapter 32's survey of class odds and ends, we met properties and static and class methods, took a quick look at the `@` decorator syntax Python offers for declaring them, and previewed decorator coding techniques. We also met function decorators briefly while exploring the `property` built-in in Chapter 38, in the context of abstract superclasses in Chapter 29, and in capsule form in Chapter 19.

This chapter picks up where all this previous decorator coverage left off. Here, we'll dig deeper into the mechanics of decorators and study more ways to code new decorators ourselves with tools like arguments and nesting. As we'll find, other concepts we studied earlier—especially state retention—show up regularly in decorators.

This is a somewhat advanced topic, and decorator construction tends to be of more interest to tool builders than to application programmers. Still, given that decorators are becoming increasingly common in popular Python frameworks, a basic understanding can help demystify their role, even if you're just a decorator user.

Besides covering decorator construction details, this chapter serves as a more realistic *case study* of Python in action. Because its examples grow larger than many of the others we've seen in this book, they better illustrate how code comes together into more complete systems and tools. As an extra perk, some of the code we'll write here may be used as general-purpose tools in your day-to-day programs.

What's a Decorator?

Simply put, *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. As suggested earlier in this book, Python decorators come in two related flavors:

- *Function decorators*, added first, 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 later, 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.

In short, decorators provide a way to insert *automatically run code* at the close 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. Such code can play a variety of roles, as described in the following sections.

Managing Calls and Instances

In typical use, this automatically run code may be used to augment calls to functions and classes. It arranges this by installing *wrapper* (a.k.a. *proxy*) objects to be invoked later:

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

Although most examples in this chapter deal with using wrappers to intercept later calls to functions and classes, this is not the only way decorators can be used:

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 or data, for example. Because this role intersects strongly with that of *metaclasses*, we'll explore additional decorator use cases in the next chapter. As detailed there, both tools run at the end of the class creation process, but class decorators often offer a lighter-weight solution.

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

Depending on your job description, you might encounter decorators as a user or a provider. As we've seen, 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. In such cases, we can get by without knowing how the decorators are coded.

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, caches call results, performs argument validity testing during debugging, times calls made to functions for optimization, and so on. Any behavior you can imagine adding to—really, wrapping around—a function call is a candidate for custom function decorators.

On the other hand, function decorators are designed to augment only a specific function or method call, not an entire *object interface*. Class decorators fill the latter role better—because they can intercept instance-creation calls, they can be used to implement arbitrary object interface augmentation or management tasks. For example, custom class decorators can trace, validate, or otherwise augment every attribute reference made for an object. They can also be used to implement proxy objects, singleton classes, and other common coding patterns. In fact, you'll find that many class decorators are a prime application of the *delegation* coding pattern we met in Chapter 31.

Why Decorators?

Like many advanced Python tools, decorators are never required from a purely technical perspective: we can often implement their functionality instead using simple helper function calls or other techniques. And at a base level, we can always manually code the name rebinding that decorators perform automatically.

That said, decorators provide an explicit syntax for such tasks, which makes intent clearer, can minimize augmentation code redundancy, and may help ensure correct API usage:

- 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.

In other words, beyond their technical model, decorators offer some advantages in terms of both code maintenance and consistency. Moreover, as structuring tools, decorators naturally foster *encapsulation* of code, which reduces redundancy and makes future changes easier.

Like most tools, decorators have some potential *drawbacks*, too—when they insert wrapper logic, they can alter the types of the decorated objects, and they may incur extra calls when used as call or interface proxies. On the other hand, the same considerations apply to any technique that adds wrapping logic to objects.

We'll explore these trade-offs in the context of real code later in this chapter. Although the choice to use decorators is ultimately subjective, their advantages are compelling enough to have escalated them to common practice in the Python world. To help you decide for yourself, let's turn to the details.

Decorators Versus Macros

Python's decorators bear similarities to what some call *aspect-oriented programming* in other languages—code inserted to run automatically before or after a function call runs. Their syntax also very closely resembles (and is likely borrowed from) Java's *annotations*, though Python's model may be considered more flexible and general.

Some liken decorators to *macros* too, but this isn't entirely apt and can be misleading. Macros, like C's `#define` preprocessor directive, are associated with textual replacement and expansion and designed for generating code. By contrast, Python's decorators are a *runtime* operation based upon name rebinding, callable objects, and often, proxies. While the two may have use cases that sometimes overlap, decorators and macros are fundamentally different in scope, implementation, and coding patterns. Comparing the two seems akin to comparing Python's `import` with a C `#include`, which similarly confuses a runtime object-based operation with text insertion.

Of course, the term *macro* has also been diluted over time—to some, it now can also refer to any canned series of steps or procedure—and users of other languages might find the analogy to decorators useful anyhow. But they should also keep in mind that decorators are about callable *objects* managing callable *objects*, not text expansion. Python tends to be best understood and used in terms of Python idioms.

The Basics

Let's get started with a first-pass look at decoration behavior from an abstract perspective. We'll write real and more substantial code soon, but since most of the magic of decorators boils down to an automatic rebinding operation, it's important to understand this mapping first—for both functions and classes.

Function Decorator Basics

As previewed earlier in this book, function decorators are largely just syntactic "sugar" that runs one function through another at the end of a `def` statement and rebinds the original function name to the result.

Usage

A function decorator is a sort 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. As of Python 3.9, the code after the `@` can be any *expression* returning a metafunction, but it's usually a simple name.

In terms of code, function decorators automatically map the following syntax:

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

F(99)                    # Call function
```

into this equivalent form, where `decorator` is a one-argument callable object that returns a callable object with the same number of arguments as `F`, if not `F` itself:

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

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

This automatic name rebinding works on any `def` statement, whether it's for a simple *function* or a *method* within a class. When the function `F` is later called, it's actually calling the object *returned* by the decorator, which may be either another object that implements required wrapping logic or the original function itself.

In other words, decoration essentially maps the first of the following into the second—though the decorator is really run only once, at decoration time:

```
func(6, 7)
decorator(func)(6, 7)
```

This automatic name rebinding accounts for the static-method and property decoration syntax we met earlier in the book:

```
class C:
    @staticmethod
    def meth(…): …            # meth = staticmethod(meth)

class C:
    @property
    def name(self): …         # name = property(name)
```

In both cases, the method name is rebound to the result of a built-in function decorator at the end of the `def` statement. Calling the original name later invokes whatever object the decorator returns. In these specific cases, the original names are rebound to a static-method router and property descriptor, but the process is much more general than this—as the next section explains.

Implementation

A decorator itself is a *callable that returns a callable*. That is, it returns the object to be called later when the decorated function is invoked through its original name—either a wrapper object to intercept later calls or the original function augmented in some way. In fact, decorators can *be* any type of callable and *return* any type of callable: any combination of functions and classes may be used, though some are better suited to certain contexts.

For example, to tap into the decoration protocol in order to manage a function just after it is created, we might code a decorator of this form:

```
def decorator(F):
    # Process function F here
    return F

@decorator
def func(): …                 # func = decorator(func)
```

Because the original decorated function is assigned back to its name, this simply adds a post-creation step to function definition. Such a structure might be used to register a function to an API, initialize function attributes, and so on.

In more typical use, to insert logic that intercepts later calls to a function, we might code a decorator to return a different object than the original function—a *proxy* for later calls:

```
def decorator(F):
    # Save or use function F
    # Return a different callable: nested def, class instance with __call__, etc.

@decorator
def func(): …                # func = decorator(func)
```

This decorator is invoked at decoration time, and the callable it returns is invoked when the original function name is later called. The decorator itself receives the decorated function; the callable returned receives whatever arguments are later passed to the decorated function's name. When coded properly, this works the same for class-level *methods*: the implied instance object simply shows up in the first argument of the returned callable.

In skeleton terms, here's one common coding pattern that captures this idea—the decorator returns a wrapper that retains the original function in an enclosing scope:

```
def decorator(F):                    # On @ decoration
    def wrapper(*args):              # On wrapped function call
        # Use F and args
        # F(*args) calls original function
    return wrapper

@decorator                           # func = decorator(func)
def func(x, y):                      # func is passed to decorator's F
    …

func(6, 7)                           # 6, 7 are passed to wrapper's *args
```

When the name `func` is later called, it really invokes the `wrapper` function returned by `decorator`; the `wrapper` function can then run the original `func` because it is still available in an *enclosing scope*. When coded this way, each decorated function produces a new scope to retain state.

To do the same with *classes*, we can overload the call operation and use instance attributes instead of enclosing scopes:

```
class decorator:
    def __init__(self, func):        # On @ decoration
        self.func = func
    def __call__(self, *args):       # On wrapped function call
        # Use self.func and args
        # self.func(*args) calls original function

@decorator
def func(x, y):                      # func = decorator(func)
    …                                # func is passed to __init__

func(6, 7)                           # 6, 7 are passed to __call__'s *args
```

When the name `func` is later called now, it really invokes the `__call__` operator-overloading method of the instance created by `decorator`; the `__call__` method can then run the original `func` because it is still available in an *instance attribute*. When coded this way, each decorated function produces a new instance to retain state.

Supporting method decoration

One subtle point about the prior class-based coding is that while it works to intercept simple *function* calls, it does not quite work when applied to class-level *method* functions:

```
class decorator:
    def __init__(self, func):          # func is method without instance
        self.func = func
    def __call__(self, *args):         # self is decorator instance
        # self.func(*args) fails!      # C instance not in args!

class C:
    @decorator
    def method(self, x, y):            # method = decorator(method)
        …                              # Rebound to decorator instance
```

When coded this way, the decorated method is rebound to an *instance* of the decorator class instead of a simple function.

The problem with this is that the `self` in the decorator's `__call__` receives the `decorator` class instance when the method is later run, and the instance of class `C` is never included in `*args`. This makes it impossible to dispatch the call to the original method—the decorator object retains the original method function, but it has no instance to pass to it.

To support *both* functions and methods, the nested function alternative works better:

```
def decorator(F):                      # F is func or method without instance
    def wrapper(*args):                # class instance in args[0] for method
        # F(*args) runs func or method
    return wrapper

@decorator
def func(x, y):                        # func = decorator(func)
    …
func(6, 7)                             # Really calls wrapper(6, 7)

class C:
    @decorator
    def method(self, x, y):            # method = decorator(method)
        …                              # Rebound to simple function

X = C()
X.method(6, 7)                         # Really calls wrapper(X, 6, 7)
```

When coded this way, `wrapper` receives the `C` class instance in its first argument, so it can dispatch to the original method and access state information.

Technically, this nested-function version works because Python creates a *bound method* object and thus passes the subject class instance to the `self` argument only when a method attribute references a simple function; when it references an instance of a callable class instead, the callable class's instance is passed to `self` to give the callable class access to its own state information. You'll see how this subtle difference can matter in more realistic examples later in this chapter.

Also note that nested functions are perhaps the most straightforward way to support decoration of both functions and methods, but not necessarily the only way. The prior chapter's *descriptors*, for example, receive both the descriptor-class and subject-class instance when called. Though more complex, later in this chapter you'll see how this tool can be leveraged in this context as well.

Class Decorator Basics

Function decorators proved so useful that the model was extended to allow class decoration. They were initially resisted because of role overlap with the next chapter's *metaclasses*; in the end, though, they were adopted because they provide a simpler way to achieve many of the same goals.

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-creation calls with extra logic that manages or augments instances created from a class. In the latter role, they may manage full object *interfaces* instead of a single callable object.

Usage

Syntactically, class decorators appear just before `class` statements, in the same way that function decorators appear just before `def` statements. In symbolic terms, for a `decorator` that must be a one-argument callable that returns a callable, the class decorator syntax:

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

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

is equivalent to the following—the class is automatically passed to the decorator function, and the decorator's result is assigned back to the class name:

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

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

The net effect is that calling the class name later to create an instance winds up triggering the callable returned by the decorator, which may or may not call the original class itself.

Implementation

New class decorators are coded with many of the same techniques used for function decorators, though some may involve *two levels* of augmentation—to manage both instance-construction calls as well as instance-interface access. Because a class decorator is also a *callable that returns a callable*, most combinations of functions and classes suffice.

However it's coded, the decorator's result is what runs when an instance is later created. For example, to simply manage a class just after it is created, return the original class itself:

```
def decorator(C):
    # Process class C here
    return C

@decorator
class C: …                                    # C = decorator(C)
```

To instead insert a wrapper layer that intercepts later instance-creation calls, return a different callable object:

```
def decorator(C):
    # Save or use class C
    # Return a different callable: nested def, class instance with __call__, etc.

@decorator
class C: …                                  # C = decorator(C)
```

The callable returned by such a class decorator typically creates and returns a new instance of the original class, augmented in some way to manage its interface. For example, the following inserts an object that intercepts undefined attributes of a class instance:

```
def decorator(cls):                         # On @ decoration
    class Wrapper:
        def __init__(self, *args):          # On instance creation
            self.wrapped = cls(*args)
        def __getattr__(self, name):        # On attribute fetch
            return getattr(self.wrapped, name)
    return Wrapper

@decorator
class C:                            # C = decorator(C)
    def __init__(self, x, y):       # Run by Wrapper.__init__
        self.attr = 'hack'

x = C(6, 7)                         # Really calls Wrapper(6, 7)
print(x.attr)                       # Runs Wrapper.__getattr__, prints "hack"
```

In this example, the decorator rebinds the class name to another class, which retains the original class in an enclosing scope and creates and embeds an instance of the original class when it's called. When an attribute is later fetched from the instance, it is intercepted by the wrapper's `__getattr__` and delegated to the embedded instance of the original class. Moreover, each decorated class creates a new scope, which remembers the original class. We'll flesh out this example into some more useful code later in this chapter.

Like function decorators, class decorators are commonly coded as either *closure* (a.k.a. "factory") functions that create and return callables, classes that use `__init__` or `__call__` methods to intercept call operations, or some combination thereof. Closure functions typically retain state in enclosing-scope references, and classes retain state in attributes.

Supporting multiple instances

As for function decorators, some callable-type combinations work better for class decorators than others. Consider the following *invalid* alternative to the class decorator of the prior example:

```
class Decorator:
    def __init__(self, C):                  # On @ decoration
        self.C = C
    def __call__(self, *args):              # On instance creation
        self.wrapped = self.C(*args)
        return self
    def __getattr__(self, attrname):        # On attribute fetch
        return getattr(self.wrapped, attrname)

@Decorator
class C: …                                  # C = Decorator(C)

x = C()
y = C()                                     # Overwrites x!
```

This code handles multiple decorated classes (each makes a new `Decorator` instance) and will intercept instance-creation calls (each runs `__call__`). Unlike the prior version, however, this version fails to handle *multiple instances* of a given class—each instance-creation call overwrites the prior saved instance. The prior version does support multiple instances because each instance-creation call makes a new independent wrapper object. More generally, either of the following patterns supports multiple wrapped instances:

```
def decorator(C):                           # On @ decoration
    class Wrapper:
        def __init__(self, *args):          # On instance creation: new Wrapper
            self.wrapped = C(*args)         # Embed instance in instance
    return Wrapper

class Wrapper: …
def decorator(C):                           # On @ decoration
    def onCall(*args):                      # On instance creation: new Wrapper
        return Wrapper(C(*args))            # Embed instance in instance
    return onCall
```

We'll study this phenomenon in a more realistic context later in the chapter too; in practice, though, we must be careful to combine callable types properly to support our intent and choose state policies wisely.

Decorator Nesting

Sometimes, one decorator isn't enough. For instance, suppose you've coded *two* function decorators to be used during development—one to test argument types before function calls and another to test return value types after function calls. You can use either independently, but what to do if you want to employ both on a single function? What you really need is a way to *nest* the two, such that the result of one decorator is the function decorated by the other. It's irrelevant which is nested, as long as both steps run on later calls.

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(…):
    …
```

runs the same as the following:

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

Here, the original function is passed through three different decorators, and the resulting callable object is assigned back to the original name. Each decorator processes the result of the prior, which may be the original function or an inserted wrapper.

If all the decorators insert wrappers, the net effect *stacks* them: when the original function name is called, three different layers of wrapping object logic will be invoked to augment the original function in three different ways. The last decorator listed is the first applied and, thus, the most deeply nested when the original function name is later called.

Just as for functions, multiple class decorators result in multiple nested function calls and possibly multiple levels and steps of wrapper logic around instance-creation calls. For example, the following code:

```
@hack
@code
class C:
    …

X = C()
```

is equivalent to the following:

```
class C:
    …
C = hack(code(C))

X = C()
```

Again, each decorator is free to return either the original class or an inserted wrapper object. With wrappers, when an instance of the original `C` class is finally requested, the call is redirected to the wrapping layer objects provided by both the `hack` and `code` decorators, which may have arbitrarily different roles—they might trace and validate attribute access for example, and both steps would be run in turn on later requests.

For instance, the following do-nothing decorators simply return the decorated function:

```
def d1(F): return F
def d2(F): return F
def d3(F): return F

@d1
@d2
@d3
def func():               # func = d1(d2(d3(func)))
    print('hack')

func()                    # Prints "hack"
```

The same syntax works on classes, as do these same do-nothing decorators.

When decorators insert wrapper function objects, though, they may augment the original function when called—the following concatenates to its result in the decorator layers, as it runs the layers from inner to outer:

```
def d1(F): return lambda: 'X' + F()
def d2(F): return lambda: 'Y' + F()
def d3(F): return lambda: 'Z' + F()

@d1
@d2
@d3
def func():               # func = d1(d2(d3(func)))
    return 'hack'

print(func())             # Prints "XYZhack"
```

We use `lambda` functions to implement wrapper layers here (each retains the wrapped function `F` in an enclosing scope); in practice, wrappers can take the form of functions, callable classes, and more. When designed well, decorator nesting allows us to combine augmentation steps in a wide variety of ways.

Decorator Arguments

Both function and class decorators can also seem to take *arguments*. Really, though, the role of these arguments is simpler than it may seem: decorator arguments are passed to a callable that *returns* the decorator—which in turn returns a callable. By nature, this usually sets up multiple levels of state retention. The following, for instance:

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

F(99)
```

is automatically mapped into this equivalent form, where `decorator` is a callable that *returns* the actual decorator. The returned decorator in turn returns the callable run later for calls to the original function name:

```
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)
```

Decorator arguments are resolved *before* decoration ever occurs, and they are usually used to retain state information for use in later calls. The decorator function in this example, for instance, might take a form like the following:

```
def decorator(A, B):
    # Save or use A, B
    def actualDecorator(F):
        # Save or use function F
        # Return a callable: nested def, class instance with __call__, etc.
        return callable
    return actualDecorator
```

The outer function in this structure generally saves the decorator arguments away as state information for use in the actual decorator, the callable it returns, or both. This code snippet retains the state information argument in enclosing function scope references, but class attributes would work as well.

In other words, decorator arguments often imply *three levels of callables*: a callable to accept decorator arguments, which returns a callable to serve as decorator, which returns a callable to handle calls to the original function or class. Each of the three levels may be a function or class and may retain state in the form of scopes or class attributes.

Decorator arguments can be used to provide attribute initialization values, call-trace message labels, attribute names to be validated, and much more—any sort of configuration *parameter* for objects or their proxies is a candidate. We'll code concrete examples of decorator arguments later in this chapter.

Decorators Manage Functions and Classes, Too

To wrap up, although much of the rest of this chapter focuses on wrapping later calls to functions and classes, it's important to remember that the decorator mechanism is more general than this—it is simply a protocol for passing functions and classes through any callable immediately after they are created. As such, it can also be used to invoke arbitrary post-creation processing:

```
def decorator(O):
    # Augment function or class O
    return O

@decorator
def F(): …                  # F = decorator(F)

@decorator
class C: …                  # C = decorator(C)
```

If we return the original decorated object this way instead of a proxy, we can manage functions and classes *themselves* rather than later calls to them. Such decorators might be used to register callable objects to an API, initialize attributes in functions or classes when they are created, and so on. Decorator roles are limited only by your imagination.

Coding Function Decorators

On to the code. In the rest of this chapter, we are going to study working examples that demonstrate the decorator concepts we just surveyed. This section presents a handful of function decorators in complete form, and the next shows tangible class decorators in action. Following that, we'll close out with two larger case studies that showcase typical decorator roles and code full-scale implementations of class privacy and argument range tests.

Tracing Function Calls

To get started, let's revive the call tracer example we met in Chapter 32. Example 39-1 defines and applies a function decorator that counts the number of calls made to the decorated function and prints a trace message for each call.

Example 39-1. 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(f'call {self.calls} to {self.func.__name__}')
        self.func(*args)

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

Notice how each function decorated with this class will create a new instance with its own saved function object and calls counter. Also, observe how the `*args` argument syntax is used to pack and unpack arbitrarily many passed-in arguments. This generality enables this decorator to be used to wrap any function with any number of positional arguments; this version doesn't yet work on keyword arguments or class-level methods and doesn't return results, but we'll fix these shortcomings later in this section.

Now, if we import this module's function and test it interactively in a REPL, we get the following sort of behavior—each call generates a trace message initially because the decorator class intercepts it:

```
$ python3
>>> from decorator1 import hack

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

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

>>> hack.calls              # Number calls in wrapper state information
2
>>> hack
<decorator1.tracer object at 0x10cafc680>
```

When run, the `tracer` class saves away the decorated function and intercepts later calls to it in order to add a layer of logic that counts and prints each call. Notice how the total number of calls shows up as an attribute of the decorated function—`hack` is really an instance of the `tracer` class when decorated, a finding that may have ramifications for programs that do type checking, but is generally benign.

For function calls, the `@` decoration syntax can be more convenient than modifying each call to account for the extra logic level, and it avoids accidentally calling the original function directly. Consider a *nondecorator* equivalent such as the following:

```
>>> calls = 0
>>> def tracer(func, *args):
        global calls
        calls += 1
        print(f'call {calls} to {func.__name__}')
        func(*args)

>>> def hack(a, b, c):       # Nondecorated function
        print(a, b, c)

>>> hack(1, 2, 3)           # Normal nontraced call: accidental?
1 2 3
>>>
>>> tracer(hack, 1, 2, 3)   # Special traced call without decorators
call 1 to hack
1 2 3
```

This alternative can be used on any function without the special `@` syntax, but unlike the decorator version, it requires extra syntax at every place where the function is *called* in your code. Furthermore, its intent may not be as obvious, and it does not ensure that the extra layer will be invoked for normal calls. Although decorators are never *required* (we can always rebind names manually), they are often the most convenient and uniform augmentation option.

Decorator State Retention Options

The preceding example raises an important point. Decorators have a variety of options for retaining state information provided at decoration time to be used during later calls to decorated objects. They generally need to support multiple decorated objects and multiple later calls, but there are several ways

to implement these goals: instance attributes, global variables, nonlocal closure variables, and function attributes can all be used for retaining state.

This topic parallels the initial state coverage in Chapter 17 but can be fleshed out here with class details, and it is so endemic to decorators that it qualifies as a prerequisite. This topic also applies to both function and class decorators, but let's explore it in the narrower function-decorator realm.

State with class-instance attributes

As an opening act in the state-retention show, Example 39-2 codes an augmented version of the prior example, which adds support for *keyword* arguments with ** syntax and *returns* the wrapped function's result to support more use cases (for nonlinear readers, we first studied keyword arguments in Chapter 18).

Example 39-2. decorator_state_classes.py

```
class tracer:                                # State via instance attributes
    def __init__(self, func):                # On @ decorator
        self.calls = 0                       # Save func for later call
        self.func  = func
    def __call__(self, *args, **kwargs):     # On call to original function
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)

@tracer
def hack(a, b, c):           # Same as: hack = tracer(hack)
    print(a + b + c)         # Triggers tracer.__init__

@tracer
def code(x, y):              # Same as: code = tracer(code)
    print(x ** y)            # Wraps code in a tracer object

if __name__ == '__main__':
    hack(1, 2, 3)            # Really calls tracer instance: runs tracer.__call__
    hack(a=4, b=5, c=6)      # hack is an instance attribute

    code(4, 2)               # Really calls tracer instance: self.func is code
    code(2, y=16)            # self.calls is per-decoration here
```

Like the original, this uses *class instance attributes* to save state explicitly. Both the wrapped function and the calls counter are *per-instance* information—each decoration gets its own copy. When run as a script, the output of this version is as follows; notice how the `hack` and `code` functions each have their own calls counter because each decoration creates a new class instance:

```
$ python3 decorator_state_classes.py
call 1 to hack
6
call 2 to hack
15
call 1 to code
16
call 2 to code
65536
```

While useful for decorating functions, this coding scheme still has issues when applied to *methods*—a shortcoming we'll address in a later revision.

State with global variables

For simpler tasks that don't require per-function data, moving state variables out to the *global scope*, as illustrated by Example 39-3, might suffice. This code still uses an enclosing-scope reference for the original decorated function but pushes the call counter out to the enclosing module.

Example 39-3. decorator_state_globals.py

```
calls = 0
def tracer(func):                           # State via enclosing scope and global
    def wrapper(*args, **kwargs):           # Instead of class attributes
        global calls                        # calls is global, not per-function
        calls += 1
        print(f'call {calls} to {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

@tracer
def hack(a, b, c):              # Same as: hack = tracer(hack)
    print(a + b + c)

@tracer
def code(x, y):                 # Same as: code = tracer(code)
    print(x ** y)

if __name__ == '__main__':
    hack(1, 2, 3)               # Really calls wrapper, assigned to hack
    hack(a=4, b=5, c=6)         # wrapper calls hack

    code(4, 2)                  # Really calls wrapper, assigned to code
    code(2, y=16)               # Global calls is not per-decoration here!
```

Unfortunately, moving the counter out to the common global scope to allow it to be changed like this also means that it will be *shared* by every wrapped function. Unlike class instance attributes, global counters are cross-program, not per-function—the counter is incremented for *any* traced function call. You can tell the difference if you compare this version's output with the prior version's—the single, shared global call counter is incorrectly updated by calls to every decorated function:

```
$ python3 decorator_state_globals.py
call 1 to hack
6
call 2 to hack
15
call 3 to code
16
call 4 to code
65536
```

State with enclosing-scope nonlocals

Shared global state may be what we want in some cases. If we really want a *per-function* counter, though, we can either use classes as before or make use of *closure* functions and the `nonlocal` statement described in Chapter 17. Because this statement allows enclosing function scope variables to be *changed*, they can serve as per-decoration, changeable data. Example 39-4 demos the basics of this scheme.

Example 39-4. decorator_state_nonlocals.py

```
def tracer(func):                       # State via enclosing scope and nonlocal
    calls = 0                           # Instead of class attrs or global
    def wrapper(*args, **kwargs):       # calls is per-function, not global
        nonlocal calls
        calls += 1
        print(f'call {calls} to {func.__name__}')
        return func(*args, **kwargs)
    return wrapper

@tracer
def hack(a, b, c):          # Same as: hack = tracer(hack)
    print(a + b + c)

@tracer
def code(x, y):             # Same as: code = tracer(code)
    print(x ** y)

if __name__ == '__main__':
    hack(1, 2, 3)           # Really calls wrapper, bound to hack
    hack(a=4, b=5, c=6)     # wrapper calls hack

    code(4, 2)              # Really calls wrapper, bound to code
    code(2, y=16)           # Nonlocal calls _is_ per-decoration here
```

Now, because enclosing-scope variables are not cross-program globals, each wrapped function gets its own counter again, just as for classes and attributes. Here's the new output:

```
$ python3 decorator_state_nonlocals.py
call 1 to hack
6
call 2 to hack
15
call 1 to code
16
call 2 to code
65536
```

State with function attributes

Finally, you can also avoid globals and classes by making use of *function attributes* for some changeable state instead of `nonlocal`. As we saw in Chapters 17 and 19, we can attach arbitrary attributes to functions by assignment, with *`func.attr=value`*. Because a factory function makes a new function on each call, its attributes become per-call state. Moreover, you need to use this technique only for state variables that must *change*; enclosing-scope references are still retained and work normally.

To demo, Example 39-5 simply uses `wrapper.calls` for state. It works the same as the preceding `nonlocal` version because the counter is again per-decorated-function.

Example 39-5. decorator_state_attributes.py

```
def tracer(func):                       # State via enclosing scope and func attr
    def wrapper(*args, **kwargs):       # calls is per-function, not global
        wrapper.calls += 1
        print(f'call {wrapper.calls} to {func.__name__}')
        return func(*args, **kwargs)
    wrapper.calls = 0
```

```
    return wrapper

@tracer
def hack(a, b, c):          # Same as: hack = tracer(hack)
    print(a + b + c)

@tracer
def code(x, y):             # Same as: code = tracer(code)
    print(x ** y)

if __name__ == '__main__':
    hack(1, 2, 3)           # Really calls wrapper, assigned to hack
    hack(a=4, b=5, c=6)     # wrapper calls hack

    code(4, 2)              # Really calls wrapper, assigned to code
    code(2, y=16)           # wrapper.calls _is_ per-decoration here
```

As we learned in Chapter 17, this works only because the name `wrapper` is retained in the enclosing `tracer` function's scope. When we later increment `wrapper.calls`, we are not changing the name `wrapper` itself, so no `nonlocal` declaration is required:

```
$ python3 decorator_state_attributes.py
…same output as prior version…
```

This scheme was almost relegated to a footnote because it may be more obscure than `nonlocal` and might be better saved for cases where other schemes don't help. However, function attributes also have a substantial advantage: like class instances, they allow access to the saved state from *outside* the decorator's code; nonlocals can only be seen inside the nested function itself, but function attributes have wider visibility.

We will employ function attributes again in an answer to one of the end-of-chapter questions, where their visibility outside callables becomes an asset. As changeable state associated with a context of use, though, they are equivalent to enclosing-scope nonlocals. As usual, choosing from multiple tools is an inherent part of the programming task.

Because decorators often imply multiple levels of callables, you can combine functions with enclosing scopes, classes with attributes, and function attributes to achieve a variety of coding structures. As you'll see later, though, this sometimes may be subtler than you expect—each decorated function should have its own state, and each decorated class may require state both for itself and for each generated instance.

In fact, as the next section will explain in more detail, if we want to apply function decorators to class-level methods, too, we also have to be careful about the distinction Python makes between decorators based on callable class instance objects and decorators based on nested functions.

Class Pitfall: Decorating Methods

When the preceding section's class-based `tracer` function decorator, Example 39-2, was initially coded, it was assumed that it could also be applied to any *method*—decorated methods should work the same, but the automatic `self` instance argument would simply be included at the front of `*args`. The only real downside to this assumption is that it is *completely wrong*, though the reasons for the failure are far from obvious.

In short, when applied to a class's method, this version of the `tracer` fails because `self` is the instance of the decorator class and the instance of the decorated subject class is not included in `*args` at all. Here's a relisting of the class in question to avoid page flipping:

```
class tracer:                                # State via instance attributes
    def __init__(self, func):                # On @ decorator
        self.calls = 0                       # Save func for later call
        self.func  = func
    def __call__(self, *args, **kwargs):     # On call to original function
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)
```

This phenomenon was introduced abstractly earlier in this chapter, but now we can see it in the context of working code. Example 39-2's class-based decorator works as advertised earlier for plain functions (copy/pasters: don't copy the initial "..." REPL prompts included in this chapter to preserve indentation after decorator lines):

```
>>> from decorator_state_classes import tracer
>>> @tracer
... def hack(a, b, c):                       # hack = tracer(hack)
        print(a + b + c)                     # Triggers tracer.__init__

>>> hack(1, 2, 3)                           # Runs tracer.__call__
call 1 to hack
6
>>> hack(a=4, b=5, c=6)                     # hack saved in an instance attribute
call 2 to hack
15
```

However, decoration of class-level methods fails (more lucid sequential readers might recognize this as an adaptation of our `Person` class resurrected from the object-oriented tutorial in Chapter 28):

```
>>> class Person:
        def __init__(self, name, pay):
            self.name = name
            self.pay  = pay
        @tracer
        def giveRaise(self, percent):        # giveRaise = tracer(giveRaise)
            self.pay *= (1.0 + percent)

>>> pat = Person('Pat Jones', 50_000)        # tracer remembers method funcs
>>> pat.giveRaise(.10)                       # Runs tracer.__call__(???, .10)
call 1 to giveRaise
TypeError: Person.giveRaise() missing 1 required positional argument: 'percent'
```

The root of the problem here is in the `self` argument of the tracer class's `__call__` method—is it a `tracer` instance or a `Person` instance? We ultimately need *both* as it's coded: the `tracer` for decorator state, and the `Person` for routing on to the original method. Really, `self` *must* be the `tracer` object to provide access to `tracer`'s state information (its `calls` and `func`); this is true whether decorating a simple function or a method.

Unfortunately, when our decorated method name is rebound to a class instance object with a `__call__`, Python passes only the `tracer` instance to `self`; it doesn't pass along the `Person` subject in the arguments list at all. Moreover, because the `tracer` knows nothing about the `Person` instance we are trying to process with method calls, there's no way to create a bound method with an instance, and thus, no way to correctly dispatch the call. This isn't a bug, but it's wildly subtle.

In the end, the prior listing winds up passing too few arguments to the decorated method, and results in an error. Add a line to the decorator's `__call__` to print all its arguments to verify this—as you can see, `self` is the `tracer` instance, and the `Person` instance is entirely absent:

```
>>> pat.giveRaise(.10)
<__main__.tracer object at 0x108a02c00> (0.10,) {}
```

As mentioned earlier, this happens because Python passes the implied subject instance to `self` when a method name is bound to a simple function only; when it is an instance of a callable class, that class's instance is passed instead. That is, Python makes a *bound method* object containing the subject instance *only* when the method is a simple function, not when it is a callable instance of another class.

Using nested functions to decorate methods

If you want your function decorators to work on *both* simple functions and class-level methods, the most straightforward solution lies in using one of the other state retention solutions described earlier—code your function decorator as *nested* `def` statements so that you don't depend on a single `self` instance argument to be both the wrapper class instance and the subject class instance.

In fact, we already *have*—both Examples 39-4 and 39-5 work for both functions and class methods by using nested functions along with `nonlocal` variables or function attributes:

```
>>> from decorator_state_nonlocals import tracer       # See Example 39-4

>>> @tracer
... def hack(a, b, c):                                 # Works for functions
        print(a + b + c)

>>> hack(1, 2, 3)
call 1 to hack
6

>>> class Person:                                      # AND works for methods
        def __init__(self, name, pay):
            self.name = name
            self.pay  = pay
        @tracer
        def giveRaise(self, percent):                  # self included in args
            self.pay *= (1.0 + percent)                # Counter in nonlocals

>>> pat = Person('Pat Jones', 50_000)
>>> pat.giveRaise(.10)
call 1 to giveRaise
>>> pat.giveRaise(.10)
call 2 to giveRaise
>>> f'{pat.pay:,.2f}'
'60,500.00'

>>> from decorator_state_attributes import tracer      # See Example 39-5
…same correct results…                                 # Counter in attributes
```

Because decorated methods here are rebound to simple functions instead of instance objects, Python correctly passes the `Person` object as the first argument, and the decorator propagates it on in the first item of `*args` to the `self` argument of the real, decorated methods. Trace through these results and decorators to make sure you have a handle on this model; the next section provides an alternative to it that supports classes but is also substantially more complex.

Using descriptors to decorate methods

Although the nested function solution illustrated in the prior section is the most straightforward way to support decorators that apply to both functions and class-level methods, other schemes are possible. The *descriptor* feature we explored in the prior chapter, for example, can help here as well.

Recall from our discussion in the prior chapter that a descriptor is normally a class attribute assigned to an object with a `__get__` method run automatically whenever that attribute is referenced and fetched:

```
class Descriptor:
    def __get__(self, instance, owner): …

class Subject:
    attr = Descriptor()

X = Subject()
X.attr          # Roughly runs Descriptor.__get__(Subject.attr, X, Subject)
```

Descriptors may also have `__set__` and `__del__` access methods, but we don't need them here. More relevant to this chapter's topic: because the descriptor's `__get__` method receives *both* the descriptor class instance and subject class instance when invoked, it's well suited to decorating methods when we need both the decorator's state and the original class instance for dispatching calls. Consider the alternative tracing decorator in Example 39-6, which *also* happens to be a descriptor when used for a class-level method (its "…" are the same as in the prior REPL session).

Example 39-6. calltracer_desc_class.py

```
class tracer(object):                          # A decorator+descriptor
    def __init__(self, func):                  # On @ decorator
        self.calls = 0                         # Save func for later call
        self.func  = func
    def __call__(self, *args, **kwargs):       # On call to original func/meth
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):        # On method attribute fetch
        return wrapper(self, instance)

class wrapper:
    def __init__(self, desc, subj):            # Save both instances
        self.desc = desc                       # Route calls back to deco/desc
        self.subj = subj
    def __call__(self, *args, **kwargs):
        return self.desc(self.subj, *args, **kwargs)  # Runs tracer.__call__

@tracer
def hack(a, b, c):                             # hack = tracer(hack)
    …                                          # Uses __call__ only

class Person:
    …
    @tracer
    def giveRaise(self, percent):              # giveRaise = tracer(giveRaise)
        …                                      # Makes giveRaise a descriptor
```

This works the same as the preceding nested function coding. Its operation varies by usage context:

- Decorated *functions* invoke only its `__call__`, and never invoke its `__get__`.
- Decorated *methods* invoke its `__get__` first to resolve the method name fetch (on `I.method`); the object returned by `__get__` retains the subject class instance and is then invoked to complete the call expression, thereby triggering the decorator's `__call__` (on `()`).

For example, given the same testing code, the call to:

```
pat.giveRaise(.10)                          # Runs __get__ then __call__
```

runs `tracer.__get__` first because the `giveRaise` attribute in the `Person` class has been rebound to a descriptor by the method function decorator. The call expression then triggers the `__call__` method of the returned `wrapper` object, which in turn invokes `tracer.__call__`. In other words, decorated method calls trigger a *five-step* process: `tracer.__get__`, which invokes `wrapper.__init__`, followed by three call operations—`wrapper.__call__`, `tracer.__call__`, and finally the original wrapped method.

The `wrapper` object retains both descriptor and subject instances, so it can route control back to the original decorator/descriptor class instance. In effect, the `wrapper` object saves the subject class instance available during method attribute fetch and adds it to the later call's arguments list, which is passed to the decorator`__call__`. Routing the call back to the descriptor class instance this way is required in this application so that all calls to a wrapped method use the same `calls` counter state information in the descriptor instance object.

Alternatively, we could use a nested function and enclosing-scope references to achieve the same effect—Example 39-7 works the same as the preceding one by swapping a wrapper class and attributes for a nested function and scope references. It requires noticeably less code but follows a similar multistep process on each decorated method call.

Example 39-7. calltracer_desc_func.py

```
class tracer(object):
    def __init__(self, func):               # On @ decorator
        self.calls = 0                      # Save func for later call
        self.func  = func
    def __call__(self, *args, **kwargs):    # On call to original func
        self.calls += 1
        print(f'call {self.calls} to {self.func.__name__}')
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):               # On method fetch
        def wrapper(*args, **kwargs):                 # Retain both inst
            return self(instance, *args, **kwargs)    # Runs __call__
        return wrapper

…rest same as Example 39-6…
```

These two descriptor-based tracers work the same as the nested-functions version, so we'll skip their output here. Add `print` statements to their methods to trace their multistep get/call processes if it helps. In either coding, this descriptor-based scheme is also substantially subtler than the nested-function option, and so is probably a second choice here. To be more blunt, if its complexity doesn't send you screaming into the night, its performance costs probably should! Still, this may be a useful coding pattern in other contexts.

Before moving on, it's also worth briefly noting that we might code this descriptor-based decorator more simply as in Example 39-8, but it would then *apply only to methods*, not to simple functions—an intrinsic limitation of attribute descriptors, and just the inverse of the problem we're trying to solve (application to both functions and methods).

Example 39-8. calltracer_desc_fail.py

```
class tracer(object):                          # For methods, but not functions!
    def __init__(self, meth):                  # On @ decorator
        self.calls = 0
        self.meth  = meth
    def __get__(self, instance, owner):        # On method fetch
        def wrapper(*args, **kwargs):          # On method call: proxy with self+inst
            self.calls += 1
            print(f'call {self.calls} to {self.meth.__name__}')
            return self.meth(instance, *args, **kwargs)
        return wrapper

@tracer                                 # OK for methods but FAILS for functions
def hack(a, b, c):                      # hack = tracer(hack)
    …                                   # No attribute fetch occurs on calls!

…rest same as Example 39-6…
```

In the rest of this chapter we're going to be casual about using classes or functions to code our function decorators, as long as they are applied only to functions. Some decorators may not require the instance of the original class, and will still work on both functions and methods if coded as a class—something like Python's own `staticmethod` decorator, for example, wouldn't require an instance of the subject class (indeed, its whole point is to remove the instance from the call).

The simpler moral of this story, though, is that if you want your decorators to work on both simple functions and methods, you're probably better off using the *nested-function* coding pattern instead of a class with call interception.

Timing Function Calls

To better sample what function decorators are capable of, let's turn to a different use case. Our next decorator times *calls* made to a decorated function—both the time for one call and the total time among all calls. As coded in Example 39-9, the decorator is applied to two functions to compare the speeds of list comprehensions and the `map` built-in.

Example 39-9. timerdeco1.py

```
"Caveat: timer won't work on methods as coded (see quiz solution)"
import time, sys

class timer:
    def __init__(self, func):
        self.func    = func
        self.alltime = 0
    def __call__(self, *args, **kargs):
        start   = time.perf_counter()
        result  = self.func(*args, **kargs)
        elapsed = time.perf_counter() - start
        self.alltime += elapsed
```

```
        print(f'{self.func.__name__}: {elapsed:.5f}, {self.alltime:.5f}')
        return result

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer
def mapcall(N):
    return list(map((lambda x: x * 2), range(N)))

if __name__ == '__main__':
    for func in (listcomp, mapcall):
        result = func(5)                        # Time for this call, result
        func(50_000)
        func(500_000)
        func(1_000_000)
        print(result)
        print(f'allTime = {func.alltime}\n')    # Total time for all func calls

    print('**map/comp =', round(mapcall.alltime / listcomp.alltime, 3))
```

When run on a macOS host by CPython 3.12, the output of this file's self-test code is as follows—giving for each function call the function name, time for this call, and time for all calls so far, along with the first call's return value, cumulative time for each function, and the map-to-comprehension time ratio at the end:

```
$ python3 timerdeco1.py
listcomp: 0.00000, 0.00000
listcomp: 0.00366, 0.00366
listcomp: 0.03134, 0.03500
listcomp: 0.05213, 0.08713
[0, 2, 4, 6, 8]
allTime = 0.08712841104716063

mapcall: 0.00001, 0.00001
mapcall: 0.00396, 0.00397
mapcall: 0.04082, 0.04479
mapcall: 0.07789, 0.12268
[0, 2, 4, 6, 8]
allTime = 0.12268476499593817

**map/comp = 1.408
```

Times vary per Python version, test machine, and other variables, of course, and cumulative time is available as a class instance attribute here. As usual, `map` calls are slower than list comprehensions when the latter can avoid a function call (or equivalently, its requirement of function calls may make `map` slower).

For comparison, see Chapter 21 for a *nondecorator* approach to timing iteration alternatives like these. As a review, we saw two per-call timing techniques there, homegrown and library—here deployed to time the 1M list comprehension case of the decorator's test code, though incurring extra admin costs that skew results slightly (add Chapter 21's folder to your `PYTHONPATH` or `sys.path`, or go there to run this):

```
>>> def listcomp(N): [x * 2 for x in range(N)]

>>> import timer                                  # Chapter 21 techniques
>>> timer.total(1, listcomp, 1_000_000)
(0.08150088600814342, None)
>>> timer.bestoftotal(5, 1, listcomp, 1_000_000)
(0.059792334999656305, None)

>>> import timeit
>>> timeit.timeit(number=1, stmt=lambda: listcomp(1_000_000))
0.08125517799635418
>>> min(timeit.repeat(repeat=5, number=1, stmt=lambda: listcomp(1_000_000)))
0.06156357398140244
```

In this specific case, a nondecorator approach would allow the subject functions to be used with or without timing, but it would also complicate the call signature when timing is desired—we'd need to add code at every call instead of once at the `def`. Moreover, in the nondecorator scheme, there would be no direct way to guarantee that all list builder calls in a program are routed through timer logic, short of finding and potentially changing them all. This may make it difficult to collect cumulative data for all calls.

In general, *decorators* may be preferred when functions are already deployed as part of a larger system and may not be easily passed to analysis functions at calls. On the other hand, because decorators charge each call to a function with augmentation logic, a *nondecorator* approach may be better if you wish to augment calls more selectively. As usual, different tools serve different roles.

Adding Decorator Arguments

The timer decorator of the prior section works, but it would be nice if it were more configurable—providing an output label and turning trace messages on and off, for instance, might be useful in a general-purpose tool like this. Decorator *arguments* come in handy here: when they're coded properly, we can use them to specify configuration options that can vary for each decorated function. A label, for instance, might be added as abstractly follows:

```
def timer(label=''):
    def decorator(func):
        def onCall(*args):        # Multilevel state retention:
            …                     # args passed to function
            func(*args)           # func retained in enclosing scope
            print(label, …)       # label retained in enclosing scope
        return onCall
    return decorator              # Returns the actual decorator

@timer('==>')                     # Like listcomp = timer('==>')(listcomp)
def listcomp(N): …                # listcomp is rebound to new onCall

listcomp(…)                       # Really calls onCall
```

This code adds an enclosing scope to retain a decorator argument for use on a later actual call. When the `listcomp` function is defined, Python really invokes `decorator`—the result of `timer`, run before decoration actually occurs—with the `label` value available in its enclosing scope. That is, `timer` *returns* the decorator, which remembers both the decorator argument and the original function, and returns the callable `onCall`, which ultimately invokes the original function on later calls. Because this structure creates new `decorator` and `onCall` functions, their enclosing scopes are per-decoration state retention.

We can put this structure to use in our timer to allow a label and a trace control flag to be passed in at decoration time. Example 39-10 does just that, coded in a module file so it can be imported as a general tool; it uses a class for the second state retention level instead of a nested function, but the net result is similar.

Example 39-10. timerdeco2.py

```
import time

def timer(label='', trace=True):                # On decorator args: retain args
    class Timer:
        def __init__(self, func):               # On @: retain decorated func
            self.func    = func
            self.alltime = 0
        def __call__(self, *args, **kargs):     # On calls: call original
            start   = time.perf_counter()
            result  = self.func(*args, **kargs)
            elapsed = time.perf_counter() - start
            self.alltime += elapsed
            if trace:
                if label: print(label, end=' ')
                print(f'{self.func.__name__}: {elapsed:.5f}, {self.alltime:.5f}')
            return result
    return Timer
```

Mostly, all we've done here is embed the original `Timer` class in an enclosing function in order to create a scope that retains the decorator arguments per deployment. The outer `timer` function is called before decoration occurs, and it simply returns the `Timer` class to serve as the actual decorator. On decoration, an instance of `Timer` is made that remembers the decorated function itself, but also has access to the decorator arguments in the enclosing function scope.

This time, rather than embedding self-test code in this file, we'll run the decorator in a different file. Example 39-11 is a client of our timer decorator, applying it to sequence iteration alternatives again.

Example 39-11. testseqs.py

```
import sys
from timerdeco2 import timer

@timer(label='[CCC]==>')
def listcomp(N):                            # Like listcomp = timer(...)(listcomp)
    return [x * 2 for x in range(N)]        # listcomp(...) triggers Timer.__call__

@timer(trace=True, label='[MMM]==>')
def mapcall(N):
    return list(map((lambda x: x * 2), range(N)))

for func in (listcomp, mapcall):
    result = func(5)                        # Time for this call, return value
    func(50_000)
    func(500_000)
    func(1_000_000)
    print(result)
    print(f'allTime = {func.alltime}\n')    # Total time for all calls

print('**map/comp =', round(mapcall.alltime / listcomp.alltime, 3))
```

When run, this file prints the following—each decorated function now has a label of its own defined by decorator arguments, which may be more useful when we need to find trace displays mixed in with a larger program's output:

```
$ python3 testseqs.py
[CCC]==> listcomp: 0.00000, 0.00000
[CCC]==> listcomp: 0.00379, 0.00379
[CCC]==> listcomp: 0.03142, 0.03521
[CCC]==> listcomp: 0.05188, 0.08709
[0, 2, 4, 6, 8]
allTime = 0.08709081003325991

[MMM]==> mapcall: 0.00001, 0.00001
[MMM]==> mapcall: 0.00401, 0.00402
[MMM]==> mapcall: 0.04025, 0.04427
[MMM]==> mapcall: 0.07776, 0.12203
[0, 2, 4, 6, 8]
allTime = 0.12203056103317067

**map/comp = 1.401
```

Run additional tests on your own to see how the decorator's configuration arguments come into play. As is, this timing function decorator can be used for any function, both in modules and interactively. In other words, it automatically serves as a *general-purpose tool* for timing code in our scripts. Watch for additional examples of decorator arguments ahead when we code decorators to implement attribute privacy and argument range checking.

Timing methods: This section's timer decorator works on any *function*, but a minor rewrite is required to apply it to class-level *methods* too. In short, and per "Class Pitfall: Decorating Methods" on page 1048, it must avoid using a nested class. Because this last mutation is being saved for an end-of-chapter quiz question, though, you'll have to stay tuned for its final code.

Coding Class Decorators

So far, we've been coding function decorators to manage function *calls*, but as we've seen, decorators work on classes too. As described earlier, while similar in concept to function decorators, class decorators are applied to classes instead—they may be used either to manage *classes* themselves or to intercept instance-creation calls in order to manage *instances*. Also like function decorators, class decorators are really just optional syntactic sugar, though they can make a programmer's intent more obvious and minimize erroneous or missed calls.

Singleton Classes

Let's start with something simple. By intercepting instance-creation calls, class decorators can be used to either manage all the instances of a class, or augment the interfaces of those instances. Example 39-12 lists a first class decorator example that does the former—managing all instances of a class. This code implements the classic *singleton* coding pattern, where at most one instance of a class ever exists. Its `singleton` function defines and returns a function for managing instances, and the `@` syntax automatically wraps up a subject class in this function.

Example 39-12. singletons1.py

```
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
```

To use this, decorate the classes for which you want to enforce a single-instance model, as in Example 39-13.

Example 39-13. singletons-test.py

```
from singletons1 import singleton

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

sue = Person('Sue', 50, 20)                     # Really calls onCall
print(sue.name, sue.pay())

bob = Person('Bob', 40, 10)                     # Same, single object
print(bob.name, bob.pay())

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

Now, when the `Person` or `Hack` class is later used to create an instance, the wrapping logic layer provided by the decorator routes instance-creation calls to `onCall`, which in turn ensures a single instance per class, regardless of how many construction calls are made. Here's this code's output when run via command line:

```
$ python3 singletons.py
Sue 1000
Sue 1000
42 42
```

Singleton coding alternatives

Interestingly, you can code a more self-contained solution here with the `nonlocal` statement to change *enclosing-scope* names as described earlier. The following alternative achieves an identical effect, by using one enclosing scope per class, instead of one global table entry per class. It works the same, but it

does not depend on names in the global scope outside the decorator (the `None` check could use `is` instead of == here, but it's a trivial test either way):

```
def singleton(aClass):                                  # On @ decoration
    instance = None
    def onCall(*args, **kwargs):                        # On instance creation
        nonlocal instance
        if instance == None:
            instance = aClass(*args, **kwargs)          # One scope per class
        return instance
    return onCall
```

You can also code a self-contained solution with either function attributes or a class instead. The first of the following codes the former, leveraging the fact that there will be one `onCall` *function* per decoration—the function object's namespace serves the same role as an enclosing scope. The second uses one *instance* per decoration, rather than an enclosing scope, function object, or global table. In fact, the second option relies on the same coding pattern that we will later label a common decorator class pitfall—here we *want* just one instance, but that's not often the case:

```
def singleton(aClass):                                  # On @ decoration
    def onCall(*args, **kwargs):                        # On instance creation
        if onCall.instance == None:
            onCall.instance = aClass(*args, **kwargs)   # One function per class
        return onCall.instance
    onCall.instance = None
    return onCall

class singleton:
    def __init__(self, aClass):                         # On @ decoration
        self.aClass = aClass
        self.instance = None
    def __call__(self, *args, **kwargs):                # On instance creation
        if self.instance == None:
            self.instance = self.aClass(*args, **kwargs) # One instance per class
        return self.instance
```

To make this singleton decorator a fully general-purpose tool, choose one version, store it in an importable module file, and indent the self-test code under a `__name__` check—steps we'll leave as suggested exercise. The final class-based version offers an explicit option with extra structure that may better support later evolution, but OOP might not be warranted in all contexts.

Tracing Object Interfaces

The singleton example of the prior section illustrated using class decorators to manage *all* the instances of a class. Another common use case for class decorators augments the interface of *each* generated instance. Class decorators can essentially install a wrapper or "*proxy*" logic layer atop instances that manages access to their interfaces.

For example, in Chapter 31, the `__getattr__` operator-overloading method was shown as a way to wrap up entire object interfaces of embedded instances in order to implement the *delegation* coding pattern. We saw similar examples in the managed attribute coverage of the prior chapter. Recall that `__getattr__` is run when an undefined attribute name is fetched; we can use this hook to intercept method calls in a controller class and propagate them to an embedded object.

The nondecorator approach

For reference and review, here's the original *nondecorator* delegation example:

```
class Wrapper:
    def __init__(self, object):
        self.wrapped = object                    # Save object
    def __getattr__(self, attrname):
        print('Trace:', attrname)                # Trace fetch
        return getattr(self.wrapped, attrname)   # Delegate fetch

x = Wrapper([1,2,3])       # Wrap a list object
x.append(4)                # Delegate to list method
```

In this code, the `Wrapper` class intercepts access to any of the wrapped object's explicitly named attributes, prints a trace message, and uses the `getattr` built-in to pass off the request to the wrapped object. Specifically, it traces attribute accesses made *outside* the wrapped object's class; accesses inside the wrapped object's methods are not caught and run normally by design. This *whole-interface* model differs from the behavior of function decorators, which wrap up just one specific method.

The class-decorator approach

Class decorators provide an alternative and convenient way to code this `__getattr__` technique and wrap an entire interface. The preceding code, for example, can be coded as a class decorator that triggers wrapped instance creation instead of passing a premade instance into the wrapper's constructor. Example 39-14 codes this mod, also supports keyword arguments with `**kargs`, and counts the number of accesses to illustrate changeable state.

Example 39-14. interfacetracer.py

```
def Tracer(aClass):                                  # On @ decorator
    class Wrapper:
        def __init__(self, *args, **kargs):          # On instance creation
            self.fetches = 0
            self.wrapped = aClass(*args, **kargs)    # Use enclosing-scope name
        def __getattr__(self, attrname):
            print('Trace: ' + attrname)              # Catches all but own attrs
            self.fetches += 1
            return getattr(self.wrapped, attrname)   # Delegate to wrapped obj
    return Wrapper


if __name__ == '__main__':

    @Tracer
    class Hack:                                  # Hack = Tracer(Hack)
        def display(self):                       # Hack is rebound to Wrapper
            print('Hack!' * 3)

    @Tracer
    class Person:                                # Person = Tracer(Person)
        def __init__(self, name, hours, rate):   # Wrapper remembers Person
            self.name = name
            self.hours = hours
            self.rate = rate
        def pay(self):                           # Accesses outside class traced
            return self.hours * self.rate        # In-method accesses not traced
```

```
work = Hack()                                # Triggers Wrapper()
work.display()                               # Triggers __getattr__
print([work.fetches])

print()
bob = Person('Bob', 40, 50)                  # bob is really a Wrapper
print(bob.name)                              # Wrapper embeds a Person
print(bob.pay())

print()
sue = Person('Sue', rate=100, hours=60)      # sue is a different Wrapper
print(sue.name)                              # With a different Person
print(sue.pay())

print()
print(bob.name)                              # bob's state != sue's state
print(bob.pay())
print('calls:', [bob.fetches, sue.fetches])  # Wrapper attrs are not traced
```

It's important to note that this is very different from the tracer decorator we met earlier (despite the name!). In "Coding Function Decorators" on page 1043, we looked at decorators that enabled us to trace and time *calls* to a given function or method. In contrast, by intercepting instance-creation calls, the class decorator here allows us to trace an entire object *interface*—that is, accesses to any of the instance's attributes.

It's also important to note that this decorator's `__getattr__` won't catch the implicit attribute fetches of *built-in operations* per the prior chapter, but we'll defer more on this subject until we code attribute privacy ahead.

The following is the output produced by this code: attribute fetches on instances of both the `Hack` and `Person` classes invoke the `__getattr__` logic in the `Wrapper` class because `work`, and `bob`, and `sue` are really instances of `Wrapper`, thanks to the decorator's redirection of instance-creation calls:

```
$ python3 interfacetracer.py
Trace: display
Hack!Hack!Hack!
[1]

Trace: name
Bob
Trace: pay
2000

Trace: name
Sue
Trace: pay
6000

Trace: name
Bob
Trace: pay
2000
calls: [4, 2]
```

Notice how there is one `Wrapper` class with state retention per decoration, generated by the nested `class` statement in the `Tracer` function, and how each instance gets its own fetches counter by virtue of generating a new `Wrapper` instance. As you'll see ahead, orchestrating this is trickier than you may expect.

Applying class decorators to built-in types

Also notice that the preceding decorates a user-defined class. Just like in the original example in Chapter 31, we can also use the decorator to wrap up a built-in object type such as a list, as long as we either subclass to allow decoration syntax or perform the decoration rebinding manually—decorator syntax requires a `class` statement for the `@` line. In the following, `x` is really a `Wrapper` again due to the indirection of decoration:

```
>>> from interfacetracer import Tracer

>>> @Tracer
... class MyList(list): pass      # MyList = Tracer(MyList)

>>> x = MyList([1, 2, 3])         # Triggers Wrapper()
>>> x.append(4)                   # Triggers __getattr__, append
Trace: append
>>> x.wrapped
[1, 2, 3, 4]

>>> MyList = Tracer(list)         # Or perform decoration manually
>>> x = MyList([4, 5, 6])         # Else subclass statement required
>>> x.append(7)
Trace: append
>>> x.wrapped
[4, 5, 6, 7]
```

The decorator approach allows us to move instance creation into the decorator itself instead of requiring a premade object to be passed in. Although this seems like a minor difference, it lets us retain normal instance-creation syntax and limits augmentation syntax to class definition. Rather than requiring all instance-creation calls to route objects through a wrapper manually, we need only augment class definitions with decorator syntax:

```
@Tracer                                        # Decorator approach
class Person: …
bob = Person('Bob', 40, 50)
sue = Person('Sue', rate=100, hours=60)

class Person: …                                # Nondecorator approach
bob = Wrapper(Person('Bob', 40, 50))
sue = Wrapper(Person('Sue', rate=100, hours=60))
```

Assuming you will make more than one instance of a class and want to apply the augmentation to every instance of a class, decorators will generally be a net win in terms of both code size and code maintenance.

Class Pitfall: Retaining Multiple Instances

Curiously, the decorator function in the preceding example can *almost* be coded as a class instead of a function with the proper operator-overloading protocol. Example 39-15's alternative coding works similarly because its `__init__` is triggered when the `@` decorator is applied to the class, and its `__call__` is triggered when a subject class instance is created. Our objects are really instances of `Tracer` this time, and we essentially just trade an enclosing-scope reference for an instance attribute here.

Example 39-15. interfacetracer-fail.py (start)

```
class Tracer:
    def __init__(self, aClass):               # On @decorator
        self.aClass = aClass                  # Use instance attribute
    def __call__(self, *args):                # On instance creation
        self.wrapped = self.aClass(*args)     # ONE (LAST) INSTANCE PER CLASS!
        return self
    def __getattr__(self, attrname):
        print('Trace:', attrname)
        return getattr(self.wrapped, attrname)

@Tracer                                       # Triggers __init__
class Hack:                                   # Like: Hack = Tracer(Hack)
    def display(self):
        print('Hack!' * 3)

work = Hack()                                 # Triggers __call__
work.display()                                # Triggers __getattr__
```

As we saw in the abstract earlier, though, this class-only alternative handles multiple *classes* as before, but it won't quite work for multiple *instances* of a given class: each instance-creation call triggers `__call__`, which overwrites the prior instance. The net effect is that `Tracer` saves just one instance—the last one created. Example 39-16 extends this file to demo the problem.

Example 39-16. interfacetracer-fail.py (continued)

```
@Tracer
class Person:                                 # Person = Tracer(Person)
    def __init__(self, name):                 # Person rebound to a Tracer
        self.name = name

bob = Person('Bob')                           # bob is really a Tracer
print(bob.name)                               # Tracer embeds a Person
sue = Person('Sue')
print(sue.name)                               # sue overwrites bob
print(bob.name)                               # OOPS: now bob's name is 'Sue'!
```

This code's output follows—because this tracer only has a single shared instance, the second overwrites the first:

```
$ python3 interfacetracer-fail.py
Trace: display
Hack!Hack!Hack!
Trace: name
Bob
Trace: name
Sue
Trace: name
Sue
```

The problem here is bad *state retention*—we make one decorator instance per class but not per class instance, such that only the last instance is retained. The solution, as in our prior class pitfall for decorating methods, lies in abandoning class-based decorators.

The earlier function-based `Tracer` version of Example 39-14, however, *does* work for multiple instances. Because it returns a *class* instead of an *instance* of that class, each instance-creation call makes a

new `Wrapper` instance instead of overwriting the state of a single shared `Tracer` instance. The original nondecorator version handles multiple instances correctly for the same reason. The moral here: decorators are not only arguably magical, they can also be incredibly subtle!

Example: "Private" and "Public" Attributes

The final two sections of this chapter present larger examples of decorator use, which give us a chance to see how concepts come together in more useful code. Both are presented with minimal description, partly to conserve space but mostly because you should already understand decorator basics well enough to be able to study these on your own.

Implementing Private Attributes

First up, the *class decorator* in Example 39-17 implements a `Private` declaration and access checks for class instance attributes—that is, for attributes stored on an instance, or inherited from one of its classes.

This decorator disallows fetch and change access to such attributes from *outside* the decorated class but still allows the class itself to access those names freely within its own methods. It's not quite the same as "private" in C++ or Java—and Python is not about control in general—but this decorator demo provides similar access validations as an option in Python for the rare and atypical cases where this might be useful during development.

We saw an initial and incomplete implementation of instance attribute privacy for *changes* only in Chapter 30. The version here extends this concept to validate attribute *fetches*, too, and it uses delegation instead of inheritance to implement the model. In a sense, this is also just an extension to the attribute-tracer class decorator we met earlier.

Although this example utilizes the syntactic sugar of class decorators to code attribute privacy, its attribute interception is ultimately still based upon the `__getattr__` and `__setattr__` operator-overloading methods we met in prior chapters. When a private attribute access is detected, this version uses the `raise` statement to raise an exception, along with an error message; the exception may be caught in a `try` or allowed to terminate the accessing script.

Example 39-17 lists the decorator's first-cut code, along with a self-test at the bottom of the file. As coded, it catches all explicit attribute fetches, but not the implicit fetches of built-in operations (more on this in a moment).

Example 39-17. access1.py

```
"""
Class decorator with Private attribute declarations.

Privacy for attributes fetched from class instances.
See self-test code at end of file for a usage example.

Rebinding is: Doubler = Private('data', 'size')(Doubler).
Private returns onDecorator, onDecorator returns onInstance,
and each onInstance instance embeds a new Doubler instance.
"""

traceMe = False
def trace(*args):
    if traceMe: print(f'[{' '.join(map(str, args))}]')   # Python 3.12+ f-string
```

```
def Private(*privates):                                # privates in enclosing scope
    def onDecorator(aClass):                           # aClass in enclosing scope
        class onInstance:                              # wrapped in instance attribute
            def __init__(self, *args, **kargs):
                self.wrapped = aClass(*args, **kargs)

            def __getattr__(self, attr):               # My attrs don't call getattr
                trace('get:', attr)                    # Others assumed in wrapped
                if attr in privates:
                    raise TypeError('private attribute fetch, ' + attr)
                else:
                    return getattr(self.wrapped, attr)

            def __setattr__(self, attr, value):            # Outside accesses
                trace('set:', attr, value)                 # Others run normally
                if attr == 'wrapped':                      # Allow my attrs
                    self.__dict__[attr] = value            # Avoid looping
                elif attr in privates:
                    raise TypeError('private attribute change, ' + attr)
                else:
                    setattr(self.wrapped, attr, value)     # Wrapped obj attrs
        return onInstance
    return onDecorator


if __name__ == '__main__':
    traceMe = True

    @Private('data', 'size')                   # Doubler = Private(...)(Doubler)
    class Doubler:
        def __init__(self, label, start):
            self.label = label                 # Accesses inside the subject class
            self.data  = start                 # Not intercepted: run normally
        def size(self):
            return len(self.data)              # Method bodies run with no checking
        def double(self):                      # Because privacy not inherited
            for i in range(self.size()):
                self.data[i] = self.data[i] * 2
        def display(self):
            print(f'{self.label} => {self.data}')

    print('Making instances...')
    X = Doubler('X is', [1, 2, 3])
    Y = Doubler('Y is', [-10, -20, -30])

    # The following all succeed properly

    print('\nExploring X instance...')
    print(X.label)                             # Accesses outside subject class
    X.display(); X.double(); X.display()       # Intercepted: validated, delegated

    print('\nExploring Y instance...')
    print(Y.label)
    Y.display(); Y.double()
    Y.label = 'Hack'
    Y.display()

    # The following all fail properly
    """
```

```
print(X.size())          # Prints "TypeError: private attribute fetch, size"
print(X.data)
X.data = [1, 1, 1]       # Prints "TypeError: private attribute change, data"
X.size = lambda S: 0
print(Y.data)
print(Y.size())
"""
```

When its `traceMe` is `True`, the module file's self-test code produces the following output. Notice how the decorator catches and validates both attribute fetches and assignments run *outside* of the wrapped class but does not catch attribute accesses *inside* the class itself:

```
$ python3 access1.py
Making instances...
[set: wrapped <__main__.Doubler object at 0x1059c7d70>]
[set: wrapped <__main__.Doubler object at 0x1059c7da0>]

Exploring X instance...
[get: label]
X is
[get: display]
X is => [1, 2, 3]
[get: double]
[get: display]
X is => [2, 4, 6]

Exploring Y instance...
[get: label]
Y is
[get: display]
Y is => [-10, -20, -30]
[get: double]
[set: label Hack]
[get: display]
Hack => [-20, -40, -60]
```

Implementation Details I

This code is nontrivial, and you're probably best off tracing through it on your own to see how it works. To help you study, though, here are a few highlights worth mentioning.

Inheritance versus delegation

The initial and limited privacy example shown in Chapter 30 used *inheritance* to mix in a `__setattr__` to catch accesses. Inheritance makes this difficult, however, because differentiating between accesses from inside or outside the class is not straightforward (inside access should be allowed to run normally, and outside access should be restricted). To work around this, the Chapter 30 example requires inheriting classes to use `__dict__` assignments to set attributes—an incomplete solution at best.

The version here uses *delegation* (embedding one object inside another) instead of inheritance; this pattern is better suited to our task as it makes it much easier to distinguish between accesses inside and outside of the subject class. Attribute accesses from outside the subject class are intercepted by the wrapper layer's overloading methods and delegated to the class if valid. Accesses inside the class itself (i.e., through `self` within its methods' code) are not intercepted and are allowed to run normally without checks because privacy is not inherited in this version.

Decorator arguments

The class decorator used here accepts any number of arguments to name private attributes. Again, though, this simply means that the arguments are passed to the `Private` function, and `Private` returns the decorator function to be applied to the subject class. That is, the arguments are used before decoration ever occurs; `Private` returns the decorator, which in turn "remembers" the privates list as an enclosing-scope reference.

State retention and enclosing scopes

Speaking of enclosing scopes, there are actually *three levels* of state retention at work in this code:

- The arguments to `Private` are used before decoration occurs and are retained as an enclosing-scope reference for use in both `onDecorator` and `onInstance`.
- The class argument to `onDecorator` is used at decoration time and is retained as an enclosing-scope reference for use at instance-creation time.
- The wrapped instance object is retained as an instance attribute in the `onInstance` proxy object for use when attributes are later accessed from outside the class.

This all works fairly naturally, given Python's scope and namespace rules.

Using `__dict__` and `__slots__` (and other virtuals)

The `__setattr__` method in this code relies on an instance object's `__dict__` attribute namespace dictionary in order to set `onInstance`'s own `wrapped` attribute. As we learned in the prior chapter, this method cannot assign an attribute directly without looping. However, it uses the `setattr` built-in instead of `__dict__` to set attributes in the *wrapped* object itself. Moreover, `getattr` is used to fetch attributes in the *wrapped* object since they may be stored in the object itself or inherited by it.

Because of that, this code will work for most classes—including those with "virtual" class-level attributes based on *slots*, *properties*, *descriptors*, and even `__getattr__` and its ilk. By assuming a namespace dictionary for itself only and using storage-neutral tools for the wrapped object, the wrapper class avoids limitations imposed by other tools.

For example, you may recall from Chapter 32 that classes with `__slots__` may not store attributes in a `__dict__`, and in fact, may not even have one of these at all. However, because we rely on a `__dict__` only at the `onInstance` level here and not in the wrapped instance, this concern does not apply. Class `onInstance` will have a `__dict__` itself because it does not use slots. In addition, because `setattr` and `getattr` apply to attributes based on both `__dict__` and `__slots__`, our decorator applies to wrapped classes using either storage scheme.

By the same reasoning, the decorator also applies to properties and similar tools: delegated names will be looked up anew in the wrapped instance, irrespective of attributes of the decorator proxy object itself.

Generalizing for Public Declarations

Now that we have a `Private` attribute implementation, it's straightforward to generalize the code to allow for `Public` declarations too—they are essentially the inverse of `Private` declarations, so we need only negate the inner test. The example listed in Example 39-18 allows a class to use decorators to

define a set of either `Private` or `Public` instance attributes—attributes of any kind stored on an instance or inherited from its classes—with the following semantics:

- `Private` declares attributes of a class's instances that *cannot* be fetched or assigned except from within the code of the class's methods. That is, any name declared `Private` cannot be accessed from outside the class, while any name not declared `Private` can be freely fetched or assigned from outside the class.
- `Public` declares attributes of a class's instances that *can* be fetched or assigned from both outside the class and within the class's methods. That is, any name declared `Public` can be freely accessed anywhere, while any name not declared `Public` cannot be accessed from outside the class.

`Private` and `Public` declarations are mutually exclusive: when using `Private`, all undeclared names are considered `Public`, and when using `Public`, all undeclared names are considered `Private`. They are essentially opposites, though undeclared names not created by a class's methods behave slightly differently—new names can be assigned and thus created outside the class under `Private` (all undeclared names are accessible) but not under `Public` (all undeclared names are inaccessible).

Again, study this code on your own to get a feel for how this works. Notice that this scheme adds an additional *fourth level of state retention* at the top, beyond that described in the preceding section: the validation functions used by the `lambda`s are saved in an extra enclosing scope coded separately. This version comes with the same caveat as its predecessor for attributes of built-in operations, noted in the file's docstring and expanded on after the example.

Example 39-18. access2.py

```
"""
Class decorator with Private and Public attribute declarations.

Controls external access to attributes stored on an instance, or
inherited by it from its classes.  Private declares attribute names
that cannot be fetched or assigned outside the decorated class,
and Public declares all the names that can.  Choose either decorator.

Caveat: as is, this works for explicitly-named attributes only.  The
__X__ operator-overloading methods fetched implicitly for built-in
operations do not trigger either __getattr__ or __getattribute__, and
hence won't be delegated to any wrapped objects that define them.  If
needed, add __X__ methods to catch and delegate built-ins (per ahead).
"""

traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)

            def __getattr__(self, attr):
                trace('get:', attr)
                if failIf(attr):
                    raise TypeError('private attribute fetch, ' + attr)
                else:
                    return getattr(self.__wrapped, attr)
```

```
            def __setattr__(self, attr, value):
                trace('set:', attr, value)
                if attr == '_onInstance__wrapped':
                    self.__dict__[attr] = value
                elif failIf(attr):
                    raise TypeError('private attribute change, ' + attr)
                else:
                    setattr(self.__wrapped, attr, value)
        return onInstance
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))

def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))
```

See the prior example's self-test code for a usage example—the effect is the same for `Private`. Here's a quick look at these class decorators in action at the interactive prompt. As advertised, non-`Private` or `Public` names can be fetched and changed from outside the subject class, but `Private` or non-`Public` names cannot:

```
>>> from access2 import Private, Public

>>> @Private('age')                             # Person = Private('age')(Person)
... class Person:                               # Person = onInstance with state
        def __init__(self, name, age):
            self.name = name
            self.age  = age                     # Inside accesses run normally

>>> X = Person('Pat', 40)
>>> X.name                                      # Outside accesses validated
'Pat'
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch, age
>>> X.age = 'Bob'
TypeError: private attribute change, age

>>> @Public('name')
... class Person:
        def __init__(self, name, age):
            self.name = name
            self.age  = age

>>> X = Person('Pat', 40)                       # X is an onInstance
>>> X.name                                      # onInstance embeds Person
'Pat'
>>> X.name = 'Sue'
>>> X.name
'Sue'
>>> X.age
TypeError: private attribute fetch, age
>>> X.age = 'Bob'
TypeError: private attribute change, age
```

Implementation Details II

To help you analyze Example 39-18's code, here are a few final notes on this version. Since this is just a generalization of the preceding section's version, the implementation notes there apply here as well.

Using "__X" pseudoprivate names

Besides generalizing, this version also makes use of Python's *__X* pseudoprivate name mangling feature, which we met in Chapter 31, to localize the `wrapped` attribute to the proxy control class by automatically prefixing it with this class's name. This avoids the prior version's risk for collisions with a `wrapped` attribute that may be used by the real, wrapped class, and it's useful in a general tool like this. It's not quite "privacy," though, because the mangled version of the name can be used freely outside the class. Notice that we also have to use the fully expanded name string—`'_onInstance__wrapped'`—as an admin-name test value in `__setattr__` because that's what Python changes it to.

Breaking privacy

Although this example does implement access controls for attributes of an instance and its classes, it is possible to subvert these controls trivially—for instance, by fetching through the expanded version of the `wrapped` attribute explicitly (`bob.pay` might not work, but the fully mangled `bob._onInstance__wrapped.pay` could!). If you have to try that hard to break them, though, these tools probably suffice for intended roles. Of course, privacy can generally be subverted in other languages too (e.g., `#define private public` may work in some C++ implementations). Although access controls may reduce accidental mods, much of this is up to programmers in any language; whenever source code may be changed, airtight access control will always be a pipe dream. More fundamentally, Python is about *enabling*, not controlling; privacy is a tool best used sparingly (if at all).

Decorator trade-offs

We could again achieve the same results without decorators by using helper functions or coding the name rebinding of decorators manually; the decorator syntax, however, makes this consistent and obvious in code. The chief potential downsides of this and any other wrapper-based approach are that attribute access incurs an extra call, and instances of decorated classes are not really instances of the original decorated class—if you test their type with *X*`.__class__` or `isinstance(`*X*`, `*C*`)`, for example, you'll find that they are instances of the *wrapper* class. Unless you plan to do introspection on objects' types, though, the type issue is irrelevant, and the extra call may apply mostly to development time; as you'll see later, it's possible to remove decorations automatically (via `-O`) if desired.

Delegating Built-In Operations

As is, this section's examples work as planned for methods and other attributes fetched *explicitly* by name. As with most software, though, there is always room for improvement. Most notably, this tool turns in mixed performance on operator-overloading methods if they are used by client classes.

Specifically, the proxy class fails to validate or delegate operator-overloading methods fetched *implicitly* by built-in operations unless such methods are redefined in the proxy. Clients that do not use operator overloading are fully supported, but others may require additional code. It's unclear that operator-overloading methods *should* be validated as private or public, but they are a part of an object's interface and should at least be routed to wrapped objects that define them.

We've encountered this issue a few times already in this book, but let's take a quick look at its impact on the realistic code we've written here and explore workarounds for it. The basic issue is easy to demo—as we've learned, the following is how a class that overloads `print` calls and + expressions normally works:

```
>>> class Tally:
        def __init__(self):
            self.sum = 0
        def __str__(self):
            return f'Tally: {self.sum}'
        def __add__(self, add):
            self.sum += add

>>> X = Tally()
>>> X.sum             # All attributes accessible
0
>>> print(X)          # Same as X.__str__() {sort of}
Tally: 0
>>> X + 5             # Same as X.__add__(5) {ditto}
>>> print(X)
Tally: 5
```

Unfortunately, objects that implement built-in operations like this fail in our proxy classes because built-in operations *skip* instance-level lookup protocols like `__getattr__`, and instead search namespaces of classes:

```
>>> from access2 import Private
>>> @Private('sum', '__add__')
... class Tally:
        …same as before…

>>> X = Tally()
>>> X.sum
TypeError: private attribute fetch, sum

>>> X.__add__(5)
TypeError: private attribute fetch, __add__

>>> print(X)
<access2.accessControl.<locals>.onDecorator.<locals>.onInstance object at 0x…etc…>

>>> X + 5
TypeError: unsupported operand type(s) for +: 'onInstance' and 'int'
```

In this session, the first two *explicit* fetches of `sum` and `__add__` are kicked out as privates as they *should* be. Because the last two *implicit* fetches of `print` and + aren't caught by the proxy, though, they are never delegated to the wrapped `Tally` object. The `print` here only works at all because it runs an `object` default to print the proxy itself. Per the prior chapter, this is an inconsistency in Python; per the following sections, it can also be avoided in full.

Workaround: Coding operator-overloading methods inline

The most straightforward way to support built-ins in delegation proxies is to redefine operator-overloading names that may appear in embedded objects. This creates some code redundancy, but it isn't impossibly onerous; can be automated with tools or superclasses; and can choose to run or skip validations for operator-overloading names declared `Private` or `Public`, depending on redefinitions' routing.

For instance, the partial listing of Example 39-19 sketches an *inline* redefinition approach—it catches and delegates built-ins by adding method definitions to the proxy itself for every operator-overloading method a wrapped object may define. It adds just four operation interceptors to illustrate, but others are similar (in this section, new code is in bold font, and all examples are based on the decorator of *access2.py* in Example 39-18).

Example 39-19. access_builtins_inline_direct.py

```
"Inline methods, skip validations"

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)

            # Intercept and delegate built-in implicit access specifically

            def __add__(self, other):
                return self.__wrapped + other           # Or getattr(), __getattr__()
            def __str__(self):
                return str(self.__wrapped)              # Or self.__wrapped.__str__()
            def __getitem__(self, index):
                return self.__wrapped[index]
            def __call__(self, *args, **kargs):
                return self.__wrapped(*args, **kargs)
            # Plus any others needed

            # Intercept and delegate explicit attribute access generically

            def __getattr__(self, attr): …same…
            def __setattr__(self, attr, value): …same…
        return onInstance
    return onDecorator
```

This works because built-ins will find their requisite methods in the proxy *class* after skipping the proxy instance. As coded, the new interceptor methods trigger the wrapped object's operator-overloading methods *directly* and so bypass the access controls of `__getattr__`, which may or may not be desirable. For alternative codings, let's move on.

Workaround: Coding operator-overloading methods in superclasses

More usefully, the prior section's added methods can be added by a common *superclass*. Given that there are dozens of such methods, an external class may be better suited to the task, especially if it is general enough to be used in any such interface-proxy class.

To demo, the superclass of Example 39-20 catches built-ins and reroutes to the wrapped object *directly* again. It's largely just a repackaging of the prior section's inline scheme, but as a separate class it requires a proxy attribute named `_wrapped`, giving access to the embedded object. The decorator itself must use this name instead of `__wrapped` in `self` references, and sans mangling in `__setattr__`. This may be subpar because it precludes the same name in wrapped objects and creates a subclass dependency, but it's better than using the mangled and subclass-specific `_onInstance__wrapped` and is no worse than a similarly named method.

Example 39-20. access_builtins_mixin_direct.py

```
"Inherit methods, skip validations"

class BuiltinsMixin:
    def __add__(self, other):
        return self._wrapped + other                        # Assume a _wrapped
    def __str__(self):                                      # Bypass __getattr__
        return str(self._wrapped)
    def __getitem__(self, index):
        return self._wrapped[index]
    def __call__(self, *args, **kargs):
        return self._wrapped(*args, **kargs)
    # Plus any others needed

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):
            …rest same, but use unmangled _wrapped instead of __wrapped…
```

Alternatively, the superclass in Example 39-21 catches built-ins and reroutes them down through the subclass `__getattr__` to apply its access controls to the operation's method name. It requires that operator-overloading names be non-`Private` or `Public` per the decorator's arguments if they are to be run, but it treats the implicit fetches of built-in operations the same as explicit-name fetches, and no `_wrapped` is required in subclasses.

Example 39-21. access_builtins_mixin_getattr.py

```
"Inherit methods, run validations"

class BuiltinsMixin:
    def __add__(self, other):
        return self.__getattr__('__add__')(other)           # Route to validator
    def __str__(self):                                      # Finish operations
        return self.__getattr__('__str__')()
    def __getitem__(self, index):
        return self.__getattr__('__getitem__')(index)
    def __call__(self, *args, **kargs):
        return self.__getattr__('__call__')(*args, **kargs)
    # Plus any others needed

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):                    # Inherit methods
            …rest unchanged…
```

Like the inline approach, both of these mix-ins also require one method per built-in operation in general tools that proxy arbitrary objects' interfaces. The next idea does marginally better.

Workaround: Generating operator-overloading descriptors

Finally, all of the inline and mix-in workarounds for built-ins we've seen so far code each operator-overloading method explicitly, and intercept the actual *call* issued for the operation, including its arguments. That makes them responsible for completing the operation, whether by operation syntax or equivalent calls.

With an alternative coding, we could instead intercept only the attribute *fetch* preceding the call by using the class-level *descriptors* of the prior chapter. Moreover, because all such descriptors will run the same, they can be generated automatically from a list of method names. Example 39-22 shows one way to code this scheme. Like Example 39-21, it routes built-in operations through the decorator's validations logic to *apply* private or public checks.

Example 39-22. access_builtins_mixin_desc.py

```
"Inherit descriptors, run validations"

class BuiltinsMixin:
    class ProxyDesc:                                        # Define descriptor
        def __init__(self, attrname):
            self.attrname = attrname
        def __get__(self, instance, owner):
            return instance.__getattr__(self.attrname)      # Run validations

    builtins = ['add', 'str', 'getitem', 'call']            # Plus any others
    for attr in builtins:
        exec(f'__{attr}__ = ProxyDesc("__{attr}__")')       # Make descriptors

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance(BuiltinsMixin):                    # Inherit descriptors
            …rest unchanged…
```

This coding may be the most concise but also the most implicit and complex. Recall that the `exec` built-in by default runs a string of code as if the string was somehow pasted where the `exec` appears. Hence, the loop at the end of this mix-in class is equivalent to the following statements, run in the mix-in class's local scope:

```
__add__ = ProxyDesc("__add__")
__str__ = ProxyDesc("__str__")
…etc…
```

The net effect creates inherited descriptor instances that respond to initial name lookups by fetching from the wrapped object in `__get__` rather than catching the later operation call itself (which happens after this step). If you still find this code confusing (and you probably should), it's equivalent to this stripped-down version, though the `name` fetch occurs implicitly in a built-in operation that skips the instance's protocols:

```
>>> class B:
        class D:
            def __get__(s, i, o): return i.meth()
        name = D()

>>> class A(B):
        def meth(self): return 'hack'

>>> I = A()
>>> I.name
'hack'
```

We could also *skip* the decorator's validations for built-in operations in this scheme by routing attribute fetches directly to the wrapped object—though this requires an accessible `_wrapped` in the decorator just like Example 39-20:

```
    class ProxyDesc:
        …
        def __get__(self, instance, owner):
            return getattr(instance._wrapped, self.attrname)     # Assume a _wrapped
```

In the end, all of these workarounds make classes that overload built-in operations work correctly with our private and public decorators—and other delegation-based decorators like them:

```
>>> from access_builtins_mixin_desc import Private

>>> @Private('sum')
... class Tally:
        def __init__(self):
            self.sum = 0
        def __str__(self):
            return f'Tally: {self.sum}'
        def __add__(self, add):
            self.sum += add

>>> X = Tally()
>>> X.sum                                       # Explicit validated
TypeError: private attribute fetch, sum
>>> X + 10                                      # Built-in delegated
>>> print(X)                                    # Built-in delegated
Tally: 10
```

Public (nonprivate) built-ins are now delegated and work, but private built-ins are validated and canceled:

```
>>> @Private('sum', '__add__')
... class Tally:
        …same as before…

>>> X = Tally()
>>> X.sum                                       # Explicit validated
TypeError: private attribute fetch, sum
>>> X + 10                                      # Built-in canceled: private
TypeError: private attribute fetch, __add__
>>> print(X)                                    # Built-in allowed: public
Tally: 0

>>> @Private('__str__')
... class Tally:
        …same as before…

>>> print(Tally())                              # Built-in canceled: private
TypeError: private attribute fetch, __str__
```

If you care to experiment further with this section's examples, see the book examples package for their complete code, as well as its comprehensive *access_builtins_TEST.py* test script and results. Here, it's time to move on to this chapter's next and final decorators case study.

Example: Validating Function Arguments

As a final example of the utility of decorators, this section develops a *function decorator* that automatically tests whether arguments passed to a function or method are within a valid numeric range. It's designed to be used during either development or production, and it can be used as a template for similar tasks (e.g., argument type testing, if you must). Again, this example is largely self-study content with a limited narrative; read the code for more details.

The Goal

In the object-oriented tutorial of Chapter 28, we wrote a class that gave a pay raise to objects representing fictitious people, based upon a passed-in percentage:

```
class Person:
     …
     def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

There, we noted that if we wanted the code to be robust, it would be a good idea to check the percentage to make sure it's not too large or too small. We could implement such a check with either `if` or `assert` statements in the method itself, using *inline tests*:

```
class Person:
    def giveRaise(self, percent):                # Validate with inline code
        if percent < 0.0 or percent > 1.0:
            raise TypeError, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))

class Person:                                    # Validate with asserts
    def giveRaise(self, percent):
        assert percent >= 0.0 and percent <= 1.0, 'percent invalid'
        self.pay = int(self.pay * (1 + percent))
```

However, this approach clutters the method with inline tests that will probably be useful only during development. For more complex cases, this can become tedious (imagine trying to inline the code needed to implement the attribute privacy provided by the last section's decorator). Perhaps worse, if the validation logic ever needs to change, there may be arbitrarily many inline copies to find and update.

A more useful and interesting alternative would be to develop a general tool that can perform range tests for us automatically for the arguments of any function or method we might code now or in the future. A *decorator* approach makes this explicit and convenient, and easy to disable once development is complete:

```
class Person:
    @rangetest(percent=(0.0, 1.0))               # Use decorator to validate
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

Isolating validation logic in a decorator simplifies both clients and future maintenance.

Notice that our goal here is different than the attribute validations coded in the prior chapter's final example. Here, we mean to validate the values of *function arguments* when passed rather than *attribute values* when accessed. Python's decorator and introspection tools allow us to code this new task just as easily.

A Basic Range-Testing Decorator for Positional Arguments

Let's start with a basic range-test implementation. To keep things simple, we'll begin by coding a decorator that works only for *positional* arguments and assumes they always appear at the same position in every call; they cannot be passed by keyword name because this can invalidate the positions declared in the decorator. Example 39-23 is our first-cut checker.

Example 39-23. rangetest0.py

```
def rangetest(*argchecks):                  # Validate positional arg ranges
    def onDecorator(func):
        if not __debug__:                   # True if "python -O main.py args..."
            return func                     # No-op: call original directly
        else:                               # Else wrapper while debugging
            def onCall(*args):
                for (ix, low, high) in argchecks:
                    if args[ix] < low or args[ix] > high:
                        errmsg = f'Argument {ix} not in {low}..{high}'
                        raise TypeError(errmsg)
                return func(*args)
            return onCall
    return onDecorator
```

As is, this code is mostly a rehash of the coding patterns we explored earlier: we use decorator *arguments*, nested *scopes* for state retention, and so on.

We also use nested `def` statements to ensure that this works for both simple functions and *methods*, as we learned earlier. When used for a class's method, `onCall` receives the subject class's instance in the first item in `*args` and passes this along to `self` in the original method function; explicitly passed argument numbers coded in the `@` decorator line start at 1 in this case, not 0, to accommodate the implicit `self`.

New here, notice this code's use of the `__debug__` built-in variable introduced in Chapter 34. In brief, Python sets this variable to `True` unless the program is being run with the `-O` optimize command-line flag (e.g., `python -O main.py`). As discussed earlier, using options in the `compile` built-in function and `compileall` standard-library module before code is run can have a similar effect.

Either way, when `__debug__` is `False`, the decorator returns the original function unchanged to avoid extra later calls and their associated performance penalty. In other words, the decorator automatically *removes* its augmentation logic when `-O` or similar is used without requiring you to physically remove the decoration lines in your code.

Example 39-24 demos how this first-iteration solution is used.

Example 39-24. rangetest0_test.py

```
from rangetest0 import rangetest
print(f'{__debug__=}')                      # False if "python -O main.py"

@rangetest((1, 0, 120))                     # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):                    # age must be in 0..120
    print(f'{name} is {age} years old')

@rangetest([0, 1, 12], [1, 1, 31], [2, 0, 2024])
def birthday(M, D, Y):
    print(f'birthday = {M}/{D}/{Y}')

class Person:
    def __init__(self, name, job, pay):
        self.job  = job
        self.pay  = pay

    @rangetest([1, 0.0, 1.0])               # giveRaise = rangetest(...)(giveRaise)
    def giveRaise(self, percent):           # Arg 0 is the self instance here
        self.pay = int(self.pay * (1 + percent))
```

```
# Comment lines raise TypeError unless "python -O" used on shell command line

persinfo('Bob Smith', 45)                # Really runs onCall(...) with state
#persinfo('Bob Smith', 200)              # Or persinfo if -O cmd line argument

birthday(8, 31, 2024)
#birthday(8, 32, 2024)

sue = Person('Sue Jones', 'dev', 100_000)
sue.giveRaise(.10)                       # Really runs onCall(self, .10)
print(sue.pay)                           # Or giveRaise(self, .10) if -O
#sue.giveRaise(1.10)
```

When run, valid calls in this code produce the following output:

```
$ python3 rangetest0_test.py
__debug__=True
Bob Smith is 45 years old
birthday = 8/31/2024
110000
```

Uncommenting any of the invalid calls causes a `TypeError` to be raised by the decorator. Here's the result when the last line is allowed to run (as usual, some of the error message text was trimmed here to save space):

```
$ python3 rangetest0_test.py
__debug__=True
Bob Smith is 45 years old
birthday = 8/31/2024
110000
TypeError: Argument 1 not in 0.0..1.0
```

Running Python with its `-O` flag at a system command line will disable range testing but also avoid the performance overhead of the wrapping layer—we wind up calling the original undecorated function directly. Assuming this is a debugging tool only, you can use this flag to optimize your program for production use. Here is the effect with the last line still run and a print added to show `sue`'s fantastical pay raise:

```
$ python3 -O rangetest0_test.py
__debug__=False
Bob Smith is 45 years old
birthday = 8/31/2024
110000
231000
```

Generalizing for Keywords and Defaults

The prior version illustrates the basics we need to employ, but it's fairly limited—it supports validating arguments passed by position only, and it does not validate keyword arguments (in fact, it assumes that no keywords are passed in a way that makes argument position numbers incorrect). Additionally, it does nothing about arguments with defaults that may be omitted in a given call. That's fine if all your arguments are passed by position and never defaulted, but it's less than ideal in a general tool. As we learned in Chapter 18, Python supports much more flexible argument-passing modes, which we're not yet addressing.

The level up of our decorator in Example 39-25 does better. By matching the wrapped function's expected arguments against the actual arguments passed in a call, it supports range validations for

arguments passed by either position or keyword name, and it skips testing for default arguments omitted in the call. Arguments to be validated are specified by keyword arguments to the decorator itself, which later steps through both the call's `*pargs` positionals tuple and its `**kargs` keywords dictionary to validate.

Example 39-25. rangetest.py

```
"""
A function decorator that performs range-test validation for
arguments passed to any function or method.  Usage synopsis:

    @rangetest(percent=(0.0, 1.0), month=(1, 12))
    def func-or-method(..., percent, ..., month=5, ...):
        ...
    func-or-method(..., value, month=8, ...)

Arguments are specified by keyword to the decorator. In the actual
call, arguments may be passed by position or keyword, and defaults
may be omitted.  See rangetest_test.py for example use cases.
"""
trace = True

def rangetest(**argchecks):                 # Validate ranges for both+defaults
    def onDecorator(func):                  # onCall remembers func and argchecks
        if not __debug__:                   # True if "python -O main.py args..."
            return func                     # Wrap if debugging; else use original
        else:
            funcname = func.__name__
            funccode = func.__code__
            funcargs = funccode.co_varnames[:funccode.co_argcount]

            def onCall(*pargs, **kargs):
                # All pargs match first N expected args by position
                # The rest must be in kargs or be omitted defaults
                positionals = funcargs[:len(pargs)]
                errormsg    = lambda *args: '%s argument "%s" not in %s..%s' % args

                for (argname, (low, high)) in argchecks.items():
                    # For all args to be checked
                    if argname in kargs:
                        # Was passed by name
                        if kargs[argname] < low or kargs[argname] > high:
                            raise TypeError(errormsg(funcname, argname, low, high))

                    elif argname in positionals:
                        # Was passed by position
                        position = positionals.index(argname)
                        if pargs[position] < low or pargs[position] > high:
                            raise TypeError(errormsg(funcname, argname, low, high))

                    else:
                        # Assume not passed: default
                        if trace:
                            print(f'-Argument "{argname}" defaulted')

                return func(*pargs, **kargs)    # OK: run original call
            return onCall
    return onDecorator
```

Next, the test script in Example 39-26 shows how the decorator is used—arguments to be validated are given by keyword decorator arguments, and at actual calls, we can pass by name or position and omit arguments with defaults even if they are to be validated otherwise.

Example 39-26. rangetest_test.py

```
"""
Test the rangetest decorator (usage differs from rangetest0).
Comment lines raise TypeError unless "python -O" or similar in compileall.
"""
from rangetest import rangetest
def announce(what): print(what.center(24, '-'))   # str method

# Test functions, positional and keyword
announce('Functions')

@rangetest(age=(0, 120))                  # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):
    print(f'{name} is {age} years old')

@rangetest(M=(1, 12), D=(1, 31), Y=(0, 2024))
def birthday(M, D, Y):
    print(f'birthday = {M}/{D}/{Y}')

persinfo('Pat', 40)
persinfo(age=40, name='Pat')
birthday(8, D=31, Y=2024)
#persinfo('Pat', 150)
#persinfo(age=150, name='Pat')
#birthday(8, Y=2025, D=40)

# Test methods, positional and keyword
announce('Methods')

class Person:
    def __init__(self, name, job, pay):
        self.job  = job
        self.pay  = pay
                                          # giveRaise = rangetest(...)(giveRaise)
    @rangetest(percent=(0.0, 1.0))        # percent passed by name or position
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

sue = Person('Sue Jones', 'dev', 100_000)
bob = Person('Bob Smith', 'dev', 100_000)
sue.giveRaise(percent=.20)
bob.giveRaise(.10)
print(f'sue=>{sue.pay}, bob=>{bob.pay}')
#sue.giveRaise(1.20)
#bob.giveRaise(percent=1.20)

# Test omitted defaults: skipped
announce('Defaults')

@rangetest(a=(1, 10), b=(1, 10), c=(1, 10), d=(1, 10))
def omitargs(a, b=7, c=8, d=9):
    print(a, b, c, d)

omitargs(1, 2, 3, 4)           # Positionals
```

```
omitargs(1, 2, 3)              # Default d
omitargs(1, 2, 3, d=4)         # Keyword d
omitargs(1, d=4)               # Default b and c
omitargs(d=4, a=1)             # Ditto
omitargs(1, b=2, d=4)          # Default c
omitargs(d=8, c=7, a=1)        # Default b

#omitargs(1, 2, 3, 11)         # Bad d
#omitargs(1, 2, 11)            # Bad c
#omitargs(1, 2, 3, d=11)       # Bad d
#omitargs(11, d=4)             # Bad a
#omitargs(d=4, a=11)           # Bad a
#omitargs(1, b=11, d=4)        # Bad b
#omitargs(d=8, c=7, a=11)      # Bad a
```

When this script is run, out-of-range arguments raise an exception as before, but arguments may be passed by either name or position, and omitted defaults are not validated. Trace its output and test this further on your own to experiment; it works like its simpler predecessor, but its scope has been greatly broadened:

```
$ python3 rangetest_test.py
-------Functions--------
Pat is 40 years old
Pat is 40 years old
birthday = 8/31/2024
--------Methods---------
sue=>120000, bob=>110000
--------Defaults--------
1 2 3 4
-Argument "d" defaulted
1 2 3 9
1 2 3 4
-Argument "b" defaulted
-Argument "c" defaulted
1 7 8 4
-Argument "b" defaulted
-Argument "c" defaulted
1 7 8 4
-Argument "c" defaulted
1 2 8 4
-Argument "b" defaulted
1 7 7 8
```

Notice that argument checks are run in the *order* they are listed in the decorator because Python retains insertion order in dictionaries. On validation errors, we get an exception as before unless the `-O` command-line argument is passed to Python to disable the decorator's logic. Here's the scene when one of the method-test lines is uncommented:

```
$ python3 rangetest_test.py
-------Functions--------
Pat is 40 years old
Pat is 40 years old
birthday = 8/31/2024
--------Methods---------
sue=>120000, bob=>110000
TypeError: giveRaise argument "percent" not in 0.0..1.0

$ python3 -O rangetest_test.py
…no error messages or default traces…
```

Implementation Details

This range-tester decorator's code relies on both introspection APIs and subtle constraints of argument passing. To be fully general, we could try to mimic Python's argument-matching logic in its entirety to see which names have been passed in which modes, but that's too much complexity for our tool and is prone to change over time. It would be better if we could somehow match the names of testable arguments given to the decorator against the names of actual arguments expected by the function to determine how the former map to the latter during a given call.

Function introspection

It turns out that the introspection API available on function objects and their associated code objects has exactly the tool we need. This API was briefly introduced in Chapter 19, but we've actually put it to use here. The set of expected *argument names* is simply the first *N* variable names attached to a function's code object:

```
>>> def func(a, b, c, e=True, f=None):       # Args: three required, two defaults
        x = 1                                # Plus two more local variables
        y = 2

>>> code = func.__code__                     # Code object of function object
>>> code.co_nlocals
7
>>> code.co_varnames                         # All local variable names
('a', 'b', 'c', 'e', 'f', 'x', 'y')
>>> code.co_varnames[:code.co_argcount]      # <== First N locals are expected args
('a', 'b', 'c', 'e', 'f')
```

And as usual, *starred-argument* names in the call proxy allow it to collect arbitrarily many arguments to be matched against the expected arguments so obtained from the function's introspection API:

```
>>> def catcher(*pargs, **kargs): print(f'{pargs}, {kargs}')

>>> catcher(1, 2, 3, 4, 5)
(1, 2, 3, 4, 5), {}
>>> catcher(1, 2, c=3, d=4, e=5)             # Arguments at calls
(1, 2), {'d': 4, 'e': 5, 'c': 3}
```

Run a `dir` call on function and code objects for more details.

Argument assumptions

Given the decorated function's set of expected argument names, the solution relies upon two constraints on argument passing *order* imposed by Python and covered in Chapter 18:

- At the call, all positional arguments appear before all keyword arguments.
- In the `def`, all nondefault arguments appear before all default arguments.

That is, a nonkeyword argument cannot generally follow a keyword argument at a *call*, and a nondefault argument cannot follow a default argument at a *definition*. All *`name=value`* syntax must appear after any simple *`name`* in both places. As we've also learned, Python matches argument values passed by position to argument names in function headers from left to right, such that these values always match the *leftmost* names in headers. Keywords match by name instead, and a given argument can receive only one value.

To simplify our work, we can also make the assumption that a call is *valid* in general—that is, that all arguments either will receive values (by name or position) or will be omitted intentionally to pick up defaults. This assumption won't necessarily hold because the function has not yet actually been called when the wrapper logic tests validity—the call may still fail later when invoked by the wrapper layer due to incorrect argument passing. As long as that doesn't cause the wrapper to fail any worse, though, we can ignore the validity of the call. This helps because validating calls before they are actually made would require us to emulate Python's argument-matching algorithm in full.

Matching algorithm

Now, given these constraints and assumptions, we can allow for both keywords and omitted default arguments in the call with this algorithm. When a call is intercepted, we can make the following assumptions and deductions:

1. Let *N* be the number of passed positional arguments, obtained from the length of the `*pargs` tuple.
2. All *N* positional arguments in `*pargs` must match the first *N* expected arguments obtained from the function's code object. This is true per Python's call ordering rules, outlined earlier, since all positionals precede all keywords in a call.
3. To obtain the names of arguments actually passed by position, we can slice the list of all expected arguments up to the length *N* of the `*pargs` passed positionals tuple.
4. Any arguments after the first *N* expected arguments either were passed by keyword or were defaulted by omission at the call.
5. For each argument name to be validated by the decorator:
   a. If the name is in `**kargs`, it was passed by name—indexing `**kargs` gives its passed value.
   b. If the name is in the first *N* expected arguments, it was passed by position—its relative position in the expected list gives its relative position in `*pargs`.
   c. Otherwise, we can assume it was omitted in the call and defaulted and need not be checked.

In other words, we can skip tests for arguments that were omitted in a call by assuming that the first *N* actually passed positional arguments in `*pargs` must match the first *N* argument names in the list of all expected arguments, and that any others must either have been passed by keyword and thus be in `**kargs`, or have been defaulted. Under this scheme, the decorator will simply skip any argument to be checked that was omitted between the rightmost positional argument and the leftmost keyword argument, between keyword arguments, or after the rightmost positional in general. Trace through the decorator and its test script to see how this is realized in code.

Open Issues

Although our range-testing tool works as planned, three caveats remain—it doesn't detect invalid calls, doesn't handle some arbitrary-argument signatures, and doesn't fully support nesting. Improvements may require extension or altogether different approaches. Here's a quick rundown of the issues.

Invalid calls

First, as mentioned earlier, calls to the original function that are *not valid* still fail in our final decorator. The following, for example, both trigger `TypeError` exceptions for a missing positional argument `a`:

```
omitargs()
omitargs(d=8, c=7, b=6)
```

These only fail, though, where we try to *invoke* the original function, at the end of the wrapper. While we could try to imitate Python's argument matching to avoid this, there's not much reason to do so—since the call would fail at this point anyhow, we might as well let Python's own argument-matching logic detect the problem for us.

Arbitrary arguments

Second, although our final version handles positional arguments, keyword arguments, and omitted defaults, it still doesn't do anything explicit about `*pargs` and `**kargs` starred-argument names that may be used in a decorated function `def` that accepts *arbitrarily many* arguments itself. This is probably moot for our purposes, though:

- If an extra *keyword* argument *is* passed, its name will show up in `**kargs` and can be tested normally if mentioned to the decorator.
- If an extra keyword argument is *not* passed, its name won't be in either `**kargs` or the sliced expected positionals list, and it will thus not be checked—it is treated as though it were defaulted, even though it is really an optional extra argument.
- If an extra *positional* argument is passed, there's no way to reference it in the decorator anyhow—its name won't be in either `**kargs` or the sliced expected arguments list, so it will simply be skipped. Because such arguments are not listed in the function's definition, there's no way to map a name given to the decorator back to an expected relative position.

In other words, as it is the code supports testing arbitrary keyword arguments by name, but not arbitrary positionals that are unnamed and hence have no set position in the function's argument signature. In terms of the function object's API, here's the effect of these tools in decorated functions:

```
>>> def func(*pargs, **kargs): pass
>>> code = func.__code__
>>> code.co_nlocals, code.co_varnames
(2, ('pargs', 'kargs'))
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(0, ())

>>> def func(a, b, *pargs, **kargs): pass
>>> code = func.__code__
>>> code.co_argcount, code.co_varnames[:code.co_argcount]
(2, ('a', 'b'))
```

Because starred-argument names show up as locals but *not* as expected arguments, they won't be a factor in our matching algorithm—names preceding them in function headers can be validated as usual, but not any extra positional arguments passed. In principle, we could extend the decorator's interface to support `*pargs` in the decorated function, too, for the rare cases where this might be useful (e.g., a special argument name with a test to apply to all arguments in `*pargs` beyond the length of the expected arguments list), but we'll pass on such an extension here.

Also, bear in mind that this pertains to values in starred *collectors* in `def` headers only; given that starred *unpackings* in *calls* are flattened before they ever reach our decorator, they are irrelevant to its code. To borrow a pathological example from Chapter 18:

```
>>> def f(a, b, c, d, e, f, g, h, i): pass
>>> f.__code__.co_varnames[:f.__code__.co_argcount]
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i')

>>> def f(*p, **k): print(p, k)
>>> f(*[1], 2, *[3], 4, f=6, *[5], **dict(g=7), h=8, **{'i': 9})
(1, 2, 3, 4, 5) {'f': 6, 'g': 7, 'h': 8, 'i': 9}
```

The call's stars here are resolved *before* the function is started. Because our decorator finds values passed to argument names by indexing keywords and mapping expected to actual positionals, it can remain blissfully ignorant of stars in the call and will work normally in this example (though, to be fair, "normally" may be an exaggeration here).

Decorator nesting

Finally, and perhaps most subtly, this code's approach does not fully support the use of *decorator nesting* to combine steps. Because it analyzes arguments using names in function definitions, and the names of the call proxy function returned by a nested decoration won't correspond to argument names in either the original function or decorator arguments, it does not fully support use in nested mode.

Technically, when nested, only the most deeply nested appearance's validations are run in full; all other nesting levels run tests on arguments passed by keyword only. Trace the code to see why; because the `onCall` proxy's call signature expects no named positional arguments, any to-be-validated arguments passed to it by position are treated as if they were omitted and hence defaulted and are thus skipped.

This may be inherent in this tool's approach—proxies change the argument name signatures at their levels, making it impossible to directly map names in decorator arguments to positions in passed argument sequences. When proxies are present, argument *names* ultimately apply to keywords only; by contrast, the first-cut solution's argument *positions* may support proxies better but do not fully support keywords.

In lieu of this nesting capability, we'll generalize this decorator to support *multiple kinds* of validations in a single decoration in an end-of-chapter quiz solution, which also gives examples of the nesting limitation in action. Since we've already neared the space allocation for this example, though, if you care about these or any other further improvements, you've officially crossed over into the realm of suggested exercises.

Decorator Arguments Versus Function Annotations

In closing, Python's annotation feature introduced in Chapter 19 could also provide an alternative to the decorator arguments used by our example to specify range tests. As we learned earlier, annotations allow us to associate expressions with arguments and return values by coding them in the `def` header line itself; Python collects annotations in a dictionary and attaches it to the annotated function.

We could use this in our example to code range limits in the header line instead of in decorator arguments. We would still need a function decorator to wrap the function in order to intercept later calls, but we would essentially trade decorator argument syntax:

```
@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                          # func = rangetest(...)(func)
    print(a + b + c)
```

for annotation syntax like this:

```
@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)
```

That is, the range constraints would be moved into the function itself instead of being coded externally in a decorator line. Example 39-27 illustrates the structure of the resulting decorators under both schemes in incomplete skeleton code for brevity. The decorator-arguments code pattern is that of our complete solution shown earlier; the annotations alternative requires one less level of nesting because it doesn't need to retain decorator arguments as state.

Example 39-27. decoargs-vs-annotation.py

```
# Using decorator arguments

def rangetest(**argchecks):
    def onDecorator(func):
        def onCall(*pargs, **kargs):
            print(argchecks)
            for check in argchecks:
                pass                          # Add validation code here
            return func(*pargs, **kargs)
        return onCall
    return onDecorator

@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                            # func = rangetest(...)(func)
    print(a + b + c)

func(1, 2, c=3)                               # Runs onCall, argchecks in scope

# Using function annotations

def rangetest(func):
    def onCall(*pargs, **kargs):
        argchecks = func.__annotations__
        print(argchecks)
        for check in argchecks:
            pass                              # Add validation code here
        return func(*pargs, **kargs)
    return onCall

@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):          # func = rangetest(func)
    print(a + b + c)

func(1, 2, c=3)                               # Runs onCall, annotations on func
```

When run, both schemes have access to the same validation test information but in different forms—the decorator argument version's information is retained in an argument in an enclosing scope, and the annotation version's information is retained in an attribute of the function itself:

```
$ python3 decoargs-vs-annotation.py
{'a': (1, 5), 'c': (0.0, 1.0)}
6
{'a': (1, 5), 'c': (0.0, 1.0)}
6
```

Fleshing out the rest of the annotation-based version is left as a suggested exercise; its code would be almost identical to that of our earlier solution because range-test information is simply on the function

instead of in an enclosing scope. Really, all this buys us is a different user interface for our tool—it will still need to match argument names against expected argument names to obtain relative positions as before.

In fact, using annotation instead of decorator arguments in this example actually *limits its utility*. By moving the validation specifications into the `def` header, we essentially commit the function to a *single role*—since annotation directly allows us to code only one expression per argument, it can have only one purpose. For instance, we cannot use range-test annotations for any other role (including the optional and unused type hinting of Chapter 6).

By contrast, because decorator arguments are coded outside the function itself, they are both easier to remove and *more general*—the code of the function itself does not imply a single decoration purpose. Crucially, by *nesting* decorators with arguments, we can often apply multiple augmentation steps to the same function; annotation directly supports only one. With decorator arguments, the function itself also retains a simpler, normal appearance.

Still, if you have a single purpose in mind, the choice between annotation and decorator arguments is largely stylistic and subjective. As is so often true in life, one person's decoration or annotation may well be another's syntactic clutter.

Chapter Summary

In this chapter, we explored decorators—both the function and class varieties. As we learned, decorators are a way to insert code to be run automatically when a function or class is defined. When a decorator is used, Python rebinds a function or class name to the callable object that the decorator returns. This hook allows us to manage functions and classes themselves or later calls to them. By adding a layer of wrapper logic to catch later calls, we can augment both function calls and instance interfaces. Decorators provide an explicit and uniform way to achieve such goals.

As we also learned, class decorators can be used to manage classes themselves rather than just their instances. Because this functionality overlaps with *metaclasses*—the topic of the next and final technical chapter—you'll have to read on for the conclusion to this story and that of this book at large.

First, though, let's work through the following quiz. Because this chapter was mostly focused on its examples, the quiz will ask you to modify some of its examples' code in order to review their concepts. Both the examples' code and the quiz's solutions are located in the book's *examples package* (see the Preface for access pointers). Look for the solutions' code there in this chapter's *_QuizAnswers* subfolder. If you're pressed for time, you're welcome to jump right into studying the solutions; programming is often as much about reading code as writing it.

Test Your Knowledge: Quiz

1. *Method decorators*: As mentioned in one of this chapter's notes, the *timerdeco2.py* module's call-timer decorator that we wrote in Example 39-10 of "Adding Decorator Arguments" on page 1055 can be applied only to simple *functions* because it uses a nested class with a `__call__` operator-overloading method to catch calls. This structure does not work for a class's *methods* because the *decorator* instance is passed to `self`, not the subject-class instance. Rewrite this decorator so that it can be applied to *both* simple functions and methods in classes, and test it on both functions and methods. (Hint: see "Class Pitfall: Decorating Methods" on page 1048 for pointers.) Note that you will probably need to use function-object *attributes* to keep track of total time, since you won't have a nested class for state retention and can't access nonlocals from outside the decorator code.

2. *Class decorators*: The `Public/Private` class decorators we wrote in module *access2.py* of Example 39-18 in this chapter's first case study example will add *performance costs* to every attribute fetch in a decorated class. Although we could simply delete the `@` decoration line to gain speed, we could also augment the decorator itself to check the `__debug__` switch and perform no wrapping at all when the `-O` Python flag is passed on the command line—just as we did for the argument range-test decorators. That way, we can speed our program without changing its source via command-line arguments (`python -O main.py`). While we're at it, we could also use one of the mix-in superclass techniques we studied to catch a few *built-in operations* too. Code and test these two extensions.
3. *Generalized argument validations*: The function and method decorator we wrote in *rangetest.py* of Example 39-25 checks that passed arguments are in a valid range, but the same code pattern could apply to similar goals such as argument type testing and possibly more. Generalize the range tester so that its single code base can be used for *multiple kinds* of argument validations. Passed-in validation functions may be the simplest solution given the coding structure here, though subclasses that provide expected methods can often provide similar generalization routes as well. This is substantially challenging, so be sure to see the solution for tips.

Test Your Knowledge: Answers

As noted, coding solutions for this quiz are in this chapter's *_QuizAnswers* subfolder of the book examples package. Each question has its own subfolder there for its files, with a *_Notes.txt* plain-text file giving background info. This edition opted to move these solutions online instead of listing them here because it saves about 10 pages and because this internet thing just might take off after all.