PART I
Functions

1

Decorators

A decorator is a tool for wrapping code around functions or classes. Decorators then explicitly apply that wrapper to functions or classes to cause them to "opt in" to the decorator's functionality. Decorators are extremely useful for addressing common prerequisite cases before a function runs (for example, ensuring authentication), or ensuring cleanup after a function runs (for example, output sanitization or exception handling). They are also useful for taking action on the decorated function or class itself. For example, a decorator might register a function with a signaling system or a URI registry in web applications.

This chapter provides an overview of what decorators are and how they interact with Python functions and classes. It enumerates certain decorators that appear in the Python standard library. Finally, it offers instruction in writing decorators and attaching them to functions and classes.

UNDERSTANDING DECORATORS

At its core, a *decorator* is a callable that *accepts* a callable and *returns* a callable. A decorator is simply a function (or other callable, such as an object with a `__call__` method) that accepts the decorated function as its positional argument. The decorator takes some action using that argument, and then either returns the original argument or some other callable (presumably that interacts with it in some way).

Because functions are first-class objects in Python, they can be passed to another function just as any other object can be. A decorator is just a function that expects another function, and does something with it.

This sounds more confusing than it actually is. Consider the following very simple decorator. It does nothing except append a line to the decorated callable's docstring.

```
def decorated_by(func):
    func.__doc__ += '\nDecorated by decorated_by.'
    return func
```

Now, consider the following trivial function:

```
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

The function's docstring is the string specified in the first line. It is what you will see if you run `help` on that function in the Python shell. Here is the decorator applied to the `add` function:

```
def add(x, y):
    """Return the sum of x and y."""
    return x + y
add = decorated_by(add)
```

Here is what you get if you run `help`:

```
Help on function add in module __main__:

add(x, y)
    Return the sum of x and y.
    Decorated by decorated_by.
(END)
```

What has happened here is that the decorator made the modification to the function's `__doc__` attribute, and then returned the original function object.

DECORATOR SYNTAX

Most times that developers use decorators to decorate a function, they are only interested in the final, decorated function. Keeping a reference to the undecorated function is ultimately superfluous.

Because of this (and also for purposes of clarity), it is undesirable to define a function, assign it to a particular name, and then immediately reassign the decorated function to the same name.

Therefore, Python 2.5 introduced a special syntax for decorators. Decorators are applied by prepending an `@` character to the name of the decorator and adding the line (without the implied decorator's method signature) immediately *above* the decorated function's declaration.

Following is the preferred way to apply a `decorated_by` decorator to the `add` method:

```
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

Note again that no method signature is being provided to `@decorated_by`. The decorator is assumed to take a single, positional argument, which is the method being decorated. (You will see a method signature in some cases, but with other provided arguments. This is discussed later in this chapter.)

This syntax allows the decorator to be applied where the function is declared, which makes it easier to read the code and immediately realize that the decorator is in play. Readability counts.

Order of Decorator Application

When is a decorator applied? When the @ syntax is being used, decorators are applied immediately *after* the decorated callable is created. Therefore, the two examples shown of how to apply decorated_by to add are exactly equivalent. First, the add function is created, and then, immediately after that, it is wrapped with decorated_by.

One important thing to note about this is that it is possible to use multiple decorators on a single callable (just as it is possible to wrap function calls multiple times).

However, note that if you use multiple decorators using the @ syntax, they are applied in order, *from bottom to top*. This may be counterintuitive at first, but it makes sense given what the Python interpreter is actually doing.

Consider the following function with two decorators applied:

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

The first thing that occurs is that the add function is created by the interpreter. Then, the decorated_by decorator is applied. This decorator returns a callable (as all decorators do), which is then sent to also_decorated_by, which does the same; the latter result is assigned to add.

Remember that the application of decorated_by is syntactically equivalent to the following:

```
add = decorated_by(add)
```

The previous two-decorator example is syntactically equivalent to the following:

```
add = also_decorated_by(decorated_by(add))
```

In both cases, the also_decorated_by decorator comes first as a human reads the code. However, the decorators are applied bottom to top for the same reason that the functions are resolved from innermost to outermost. The same principles are at work.

In the case of a traditional function call, the interpreter must first resolve the inner function call in order to have the appropriate object or value to send to the outer call.

```
add = also_decorated_by(decorated_by(add))  # First, get a return value for
                                            # `decorated_by(add)`.
add = also_decorated_by(decorated_by(add))  # Send that return value to
                                            # `also_decorated_by`.
```

With a decorator, first the add function is created normally.

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

Then, the `@decorated_by` decorator is called, being sent the `add` function as its decorated method.

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

The `@decorated_by` function returns its own callable (in this case, a modified version of `add`). *That* value is what is then sent to `@also_decorated_by` in the final step.

```
@also_decorated_by
@decorated_by
def add(x, y):
    """Return the sum of x and y."""
    return x + y
```

When applying decorators, it is important for you to remember that they are applied bottom to top. Many times, order does matter.

WHERE DECORATORS ARE USED

The standard library includes many modules that incorporate decorators, and many common tools and frameworks make use of them for common functionality.

For example, if you want to make a method on a class not require an instance of the class, you use the `@classmethod` or `@staticmethod` decorator, which is part of the standard library. The `mock` module (which is used for unit testing, and which was added to the standard library in Python 3.3) allows the use of `@mock.patch` or `@mock.patch.object` as a decorator.

Common tools also use decorators. Django (which is a common web framework for Python) uses `@login_required` as a decorator to allow developers to specify that a user must be logged in to view a particular page, and uses `@permission_required` for applying more specific permissions. Flask (another common web framework) uses `@app.route` to serve as a registry between specific URIs and the functions that run when the browser hits those URIs.

Celery (a common Python task runner) uses a complex `@task` decorator to identify a function as an asynchronous task. This decorator actually returns an instance of a `Task` class, which illustrates how decorators can be used to make a very convenient API.

WHY YOU SHOULD WRITE DECORATORS

Decorators provide an excellent way to say, "I want this specific, reusable piece of functionality in these specific places." When written well, they are modular and explicit.

The modularity of decorators (you can apply or remove them from functions or classes easily) makes them ideal for avoiding the repetition of boilerplate setup and teardown code. Similarly, because decorators interact with the decorated function itself, they excel at registering functions elsewhere.

Also, decorators are explicit. They are applied, in-place, to all callables where they are needed. This is valuable for readability, and therefore for debugging. It is obvious exactly what is being applied and where.

WHEN YOU SHOULD WRITE DECORATORS

Several very good use cases exist for writing decorators in Python applications and modules.

Additional Functionality

Probably the most common reason to write a decorator is if you want to add additional functionality before or after the decorated method is executed. This could include use cases such as checking authentication or logging the result of a function to a consistent location.

Data Sanitization or Addition

A decorator could also sanitize the values of arguments being passed to the decorated function, to ensure consistency of argument type, or that a value conforms to a specific pattern. For example, a decorator could ensure that the values sent to a function conform to a specific type, or meet some other validation standard. (You will see an example of this shortly, a decorator called `@requires_ints`.)

A decorator can also transform or sanitize data that is *returned from* a function. A valuable use case for this is if you want to have functions that return native Python objects (such as lists or dictionaries), but ultimately receive a serialized format (such as JSON or YAML) on the other end.

Some decorators actually provide additional data to a function, usually in the form of additional arguments. The `@mock.patch` decorator is an example of this, because it (among other things) provides the mock object that it creates as an additional positional argument to the function.

Function Registration

Many times, it is useful to register a function elsewhere—for example, registering a task in a task runner, or a function with a signal handler. Any system in which some external input or routing mechanism decides what function runs is a candidate for function registration.

WRITING DECORATORS

Decorators are simply functions that (usually) accept the decorated callable as their only argument, and that return a callable (such as in the previous trivial example).

It is important to note that the decorator code itself runs when the decorator is *applied* to the decorated function, rather than when the decorated function is called. Understanding this is critical, and will become very clear over the course of the next several examples.

An Initial Example: A Function Registry

Consider the following simple registry of functions:

```
registry = []
def register(decorated):
    registry.append(decorated)
    return decorated
```

The `register` method is a simple decorator. It appends the positional argument, decorated to the registry variable, and then returns the decorated method unchanged. Any method that receives the `register` decorator will have itself appended to `registry`.

```
@register
def foo():
    return 3

@register
def bar():
    return 5
```

If you have access to the registry, you can easily iterate over it and execute the functions inside.

```
answers = []
for func in registry:
    answers.append(func())
```

The `answers` list at this point would now contain `[3, 5]`. This is because the functions are executed in order, and their return values are appended to `answers`.

Several less-trivial uses for function registries exist, such as adding "hooks" into code so that custom functionality can be run before or after critical events. Here is a `Registry` class that can handle just such a case:

```
class Registry(object):
    def __init__(self):
        self._functions = []

    def register(self, decorated):
        self._functions.append(decorated)
        return decorated

    def run_all(self, *args, **kwargs):
        return_values = []
        for func in self._functions:
            return_values.append(func(*args, **kwargs))
        return return_values
```

One thing worth noting about this class is that the `register` method—the decorator—still works the same way as before. It is perfectly fine to have a bound method as a decorator. It receives `self` as the first argument (just as any other bound method), and expects one additional positional argument, which is the decorated method.

By making several different registry instances, you can have entirely separate registries. It is even possible to take the same function and register it with more than one registry, as shown here:

```
a = Registry()
b = Registry()

@a.register
def foo(x=3):
    return x
```

```
@b.register
def bar(x=5):
    return x

@a.register
@b.register
def baz(x=7):
    return x
```

Running the code from either registry's `run_all` method gives the following results:

```
a.run_all()    # [3, 7]
b.run_all()    # [5, 7]
```

Notice that the `run_all` method is able to take arguments, which it then passes to the underlying functions when they are run.

```
a.run_all(x=4)    # [4, 4]
```

Execution-Time Wrapping Code

These decorators are very simple because the decorated function is passed through unmodified. However, sometimes you want additional functionality to run when the decorated method is *executed*. You do this by returning a different callable that adds the appropriate functionality and (usually) calls the decorated method in the course of its execution.

A Simple Type Check

Here is a simple decorator that ensures that every argument the function receives is an integer, and complains otherwise:

```
def requires_ints(decorated):
    def inner(*args, **kwargs):
        # Get any values that may have been sent as keyword arguments.
        kwarg_values = [i for i in kwargs.values()]

        # Iterate over every value sent to the decorated method, and
        # ensure that each one is an integer; raise TypeError if not.
        for arg in list(args) + kwarg_values:
            if not isinstance(arg, int):
                raise TypeError('%s only accepts integers as arguments.' %
                                decorated.__name__)

        # Run the decorated method, and return the result.
        return decorated(*args, **kwargs)
    return inner
```

What is happening here?

The decorator itself is `requires_ints`. It accepts one argument, `decorated`, which is the decorated callable. The only thing that this decorator does is return a new callable, the local function `inner`. This function replaces the decorated method.

You can see this in action by declaring a function and decorating it with `requires_ints`:

```
@requires_ints
def foo(x, y):
    """Return the sum of x and y."""
    return x + y
```

Notice what you get if you run `help(foo)`:

```
Help on function inner in module __main__:

inner(*args, **kwargs)
(END)
```

The `inner` function has been assigned to the name `foo` *instead of* the original, defined function. If you run `foo(3, 5)`, the `inner` function runs with those arguments. The inner function performs the type check, and then it runs the decorated method simply because the inner function calls it using return decorated(*args, **kwargs), returning 8. Absent this call, the decorated method would have been ignored.

Preserving the help

It often is not particularly desirable to have a decorator steamroll your function's docstring or hijack the output of `help`. Because decorators are tools for adding generic and reusable functionality, they are necessarily going to be more vague. And, generally, if someone using a function is trying to run `help` on it, he or she wants information about the guts of the function, not the shell.

The solution to this problem is actually … a decorator. Python implements a decorator called `@functools.wraps` that copies the important introspection elements of one function onto another function.

Here is the same `@requires_ints` decorator, but it adds in the use of `@functools.wraps`:

```
import functools


def requires_ints(decorated):
    @functools.wraps(decorated)
    def inner(*args, **kwargs):
        # Get any values that may have been sent as keyword arguments.
        kwarg_values = [i for i in kwargs.values()]

        # Iterate over every value sent to the decorated method, and
        # ensure that each one is an integer; raise TypeError if not.
        for arg in args + kwarg_values:
            if not isinstance(i, int):
                raise TypeError('%s only accepts integers as arguments.' %
                                decorated.__name__)

        # Run the decorated method, and return the result.
        return decorated(*args, **kwargs)
    return inner
```

The decorator itself is almost entirely unchanged, except for the addition of the second line, which applies the `@functools.wraps` decorator to the `inner` function. You must also import `functools` now (which is in the standard library). You will also notice some additional syntax. This decorator actually takes an argument (more on that later).

Now you apply the decorator to the same function, as shown here:

```
@requires_ints
def foo(x, y):
    """Return the sum of x and y."""
    return x + y
```

Here is what happens when you run `help(foo)` now:

```
Help on function foo in module __main__:

foo(x, y)
    Return the sum of x and y.
(END)
```

You see that the docstring for `foo`, as well as its method signature, is what is read out when you look at `help`. Underneath the hood, however, the `@requires_ints` decorator is still applied, and the `inner` function is still what runs.

Depending on which version of Python you are running, you will get a slightly different result from running `help` on `foo`, specifically regarding the function signature. The previous paste represents the output from Python 3.4. However, in Python 2, the function signature provided will still be that of `inner` (so, `*args` and `**kwargs` rather than `x` and `y`).

User Verification

A common use case for this pattern (that is, performing some kind of sanity check before running the decorated method) is user verification. Consider a method that is expected to take a user as its first argument.

The user should be an instance of this `User` and `AnonymousUser` class, as shown here:

```
class User(object):
    """A representation of a user in our application."""

    def __init__(self, username, email):
        self.username = username
        self.email = email


class AnonymousUser(User):
    """An anonymous user; a stand-in for an actual user that nonetheless
    is not an actual user.
    """
    def __init__(self):
        self.username = None
        self.email = None

    def __nonzero__(self):
        return False
```

A decorator is a powerful tool here for isolating the boilerplate code of user verification. A `@requires_user` decorator can easily verify that you got a `User` object and that it is not an anonymous user.

```
import functools


def requires_user(func):
    @functools.wraps(func)
    def inner(user, *args, **kwargs):
        """Verify that the user is truthy; if so, run the decorated method,
        and if not, raise ValueError.
        """
        # Ensure that user is truthy, and of the correct type.
        # The "truthy" check will fail on anonymous users, since the
        # AnonymousUser subclass has a `__nonzero__` method that
        # returns False.
        if user and isinstance(user, User):
            return func(user, *args, **kwargs)
        else:
            raise ValueError('A valid user is required to run this.')
    return inner
```

This decorator applies a common, boilerplate need—the verification that a user is logged in to the application. When you implement this as a decorator, it is reusable and more easily maintainable, and its application to functions is clear and explicit.

Note that this decorator will only correctly wrap a function or static method, and will fail if wrapping a bound method to a class. This is because the decorator ignores the expectation to send `self` as the first argument to a bound method.

Output Formatting

In addition to sanitizing *input to* a function, another use for decorators can be sanitizing *output from* a function.

When you're working in Python, it is normally desirable to use native Python objects when possible. Often, however, you want a serialized output format (for example, JSON). It is cumbersome to manually convert to JSON at the end of every relevant function, and (and it's not a good idea, either). Ideally, you should be using the Python structures right up until serialization is necessary, and there may be other boilerplate that happens just before serialization (such as or the like).

Decorators provide an excellent, portable solution to this problem. Consider the following decorator that takes Python output and serializes the result to JSON:

```
import functools
import json


def json_output(decorated):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
    """
    @functools.wraps(decorated)
    def inner(*args, **kwargs):
        result = decorated(*args, **kwargs)
        return json.dumps(result)
    return inner
```

Apply the `@json_output` decorator to a trivial function, as shown here:

```
@json_output
def do_nothing():
    return {'status': 'done'}
```

Run the function in the Python shell, and you get the following:

```
>>> do_nothing()
'{"status": "done"}'
```

Notice that you got back a *string* that contains valid JSON. You did *not* get back a dictionary.

The beauty of this decorator is in its simplicity. Apply it to a function, and suddenly a function that did return a Python dictionary, list, or other object now returns its JSON-serialized version.

You might ask, "Why is this valuable?" After all, you are adding a one-line decorator that essentially removes a single line of code—a call to `json.dumps`. However, consider the value of having this decorator as the application's needs expand.

For example, what if certain exceptions should be trapped and output specifically formatted JSON, rather than having the exception bubble up and traceback? Because you have a decorator, that functionality is very easy to add.

```
import functools
import json


class JSONOutputError(Exception):
    def __init__(self, message):
        self._message = message

    def __str__(self):
        return self._message


def json_output(decorated):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
    """
    @functools.wraps(decorated)
    def inner(*args, **kwargs):
        try:
            result = decorated(*args, **kwargs)
        except JSONOutputError as ex:
            result = {
                'status': 'error',
                'message': str(ex),
            }
        return json.dumps(result)
    return inner
```

By augmenting the `@json_output` decorator with this error handling, you have added it to *any function where the decorator was already applied*. This is part of what makes decorators so valuable. They are very useful tools for code portability and reusability.

Now, if a function decorated with `@json_output` raises a `JSONOutputError`, you will get this special error handling. Here is a function that does:

```
@json_output
def error():
    raise JSONOutputError('This function is erratic.')
```

Running the `error` function in the Python interpreter gives you the following:

```
>>> error()
'{"status": "error", "message": "This function is erratic."}'
```

Note that only the `JSONOutputError` exception class (and any subclasses) receives this special handling. Any other exception is passed through normally, and generates a traceback. Consider this function:

```
@json_output
def other_error():
    raise ValueError('The grass is always greener...')
```

When you run it, you will get the traceback you expect, as shown here:

```
>>> other_error()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in inner
  File "<stdin>", line 3, in other_error
ValueError: The grass is always greener...
```

This reusability and maintainability is part of what makes decorators valuable. Because a decorator is being used for a reusable, generally applicable concept throughout the application (in this case, JSON serialization), the decorator becomes the place for housing that functionality as needs arise that are applicable whenever that concept is used.

Essentially, decorators are tools to avoid repeating yourself, and part of their value is in providing hooks for future maintenance.

This can be accomplished without the use of decorators. Consider the example of requiring a logged-in user. It is not difficult to write a function that does this and simply place it near the top of functions that require that functionality. The decorator is primarily syntactic sugar. The syntactic sugar has value, though. Code is read more often than it is written, after all, and it is easy to locate decorators at a glance.

Logging

One final example of execution-time wrapping of code is a general-use logging function. Consider the following decorator that causes the function call, timings, and result to be logged:

```
import functools
import logging
import time


def logged(method):
    """Cause the decorated method to be run and its results logged, along
```

```
        with some other diagnostic information.
        """
        @functools.wraps(method)
        def inner(*args, **kwargs):
            # Record our start time.
            start = time.time()

            # Run the decorated method.
            return_value = method(*args, **kwargs)

            # Record our completion time, and calculate the delta.
            end = time.time()
            delta = end - start

            # Log the method call and the result.
            logger = logging.getLogger('decorator.logged')
            logger.warn('Called method %s at %.02f; execution time %.02f '
                        'seconds; result %r.' %
                        (method.__name__, start, delta, return_value))

            # Return the method's original return value.
            return return_value
        return inner
```

When applied to a function, this decorator runs that function normally, but uses the Python `logging` module to log out information about the function call after it completes. Now, suddenly, you have (extremely rudimentary) logging of any function where this decorator is applied.

```
>>> import time
>>> @logged
... def sleep_and_return(return_value):
...      time.sleep(2)
...      return return_value
...
>>>
>>> sleep_and_return(42)
Called method sleep_and_return at 1424462194.70;
     execution time 2.00 seconds; result 42.
42
```

Unlike the previous examples, this decorator does not alter the function call in an obvious way. No cases exist where you apply this decorator and get a different result from the decorated function than you did from the undecorated function. The previous examples raised exceptions or modified the result if this or that check did not pass. This decorator is more invisible. It does some under-the-hood work, but in no situation should it change the actual result.

Variable Arguments

It is worth noting that the `@json_output` and `@logged` decorators both provide `inner` functions that simply take, and pass on with minimal investigation, variable arguments and keyword arguments.

This is an important pattern. One way that it is particularly important is that many decorators may be used to decorate plain functions as well as methods of classes. Remember that in Python, methods declared in classes receive an additional positional argument, conventionally known as `self`. This *does not change* when decorators are in use. (This is why the `requires_user` decorator shown earlier does not work on bound methods within classes.)

For example, if `@json_result` is used to decorate a method of a class, the `inner` function is called and it receives the instance of the class as the first argument. In fact, this is fine. In this case, that argument is simply `args[0]`, and it is passed to the decorated method unmolested.

Decorator Arguments

One thing that has been consistent about all the decorators enumerated thus far is that the decorators themselves appear not to take any arguments. As discussed, there is an implied argument—the method that is being decorated.

However, sometimes it is useful to have the decorator *itself* take some information that it needs to decorate the method appropriately. The difference between an argument passed to the decorator and an argument passed to the function at call time is precisely that. An argument to the decorator is processed once, when the function is declared and decorated. By contrast, arguments to the function are processed when that function is called.

You have already seen an example of an argument sent to a decorator with the repeated use of `@functools.wraps`. It takes an argument—the method being wrapped, whose help and docstring and the like should be preserved.

However, decorators have implied call signatures. They take one positional argument—the method being decorated. So, how does this work?

The answer is that it is complicated. Recall the basic decorators that have execution-time wrapping of code. They declare an inner method in local scope that they then return. This is the callable returned by the decorator. It is what is assigned to the function name. Decorators that take arguments add one more wrapping layer to this dance. This is because the decorator that takes the argument *is not actually the decorator*. Rather, it is a function that *returns the decorator*, which is a function that takes one argument (the decorated method), which then decorates the function and returns a callable.

That sounds confusing. Consider the following example where a `@json_output` decorator is augmented to ask about indentation and key sorting:

```
import functools
import json


class JSONOutputError(Exception):
    def __init__(self, message):
        self._message = message

    def __str__(self):
        return self._message


def json_output(indent=None, sort_keys=False):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
```

```
        """
        def actual_decorator(decorated):
            @functools.wraps(decorated)
            def inner(*args, **kwargs):
                try:
                    result = decorated(*args, **kwargs)
                except JSONOutputError as ex:
                    result = {
                        'status': 'error',
                        'message': str(ex),
                    }
                return json.dumps(result, indent=indent, sort_keys=sort_keys)
            return inner
        return actual_decorator
```

So, what has happened here, and why does this work?

This is a function, `json_output`, which accepts two arguments (`indent` and `sort_keys`). It returns another function, called `actual_decorator`, which is (as its name suggests) intended to be used as a decorator. That is a classic decorator—a callable that accepts a single callable (`decorated`) as an argument and returns a callable (`inner`).

Note that the `inner` function has changed slightly to accommodate the `indent` and `sort_keys` arguments. These arguments mirror similar arguments accepted by `json.dumps`, so the call to `json.dumps` accepts the values provided to `indent` and `sort_keys` in the decorator's signature and provides them to `json.dumps` in the antepenultimate line.

The `inner` function is what ultimately makes use of the `indent` and `sort_keys` arguments. This is fine, because Python's block scoping rules allow for this. It also is not a problem that this might be called with different values for `inner` and `sort_keys`, because `inner` is a local function (a different copy is returned each time the decorator is used).

Applying the `json_output` function looks like this:

```
@json_output(indent=4)
def do_nothing():
    return {'status': 'done'}
```

And if you run the `do_nothing` function now, you get a JSON block back with indentation and newlines added, as shown here:

```
>>> do_nothing()
'{\n    "status": "done"\n}'
```

How Does This Work?

But wait. If `json_output` is not a decorator, but a function that returns a decorator, why does it look like it is being applied as a decorator? What is the Python interpreter doing here that makes this work?

More explanation is in order. The key here is in the order of operations. Specifically, the function call (`json_output(indent=4)`) precedes the decorator application syntax (`@`). Thus, the *result of* the function call is used to apply the decorator.

The first thing that is happening is that the interpreter is seeing the function call for `json_output` and resolving that call (note that the boldface does *not* include the `@`):

```
@json_output(indent=4)
def do_nothing():
    return {'status': 'done'}
```

All the `json_output` function does is define another function, `actual_decorator`, and return it. As the result of that function, it is then provided to `@`, as shown here:

```
@actual_decorator
def do_nothing():
    return {'status': 'done'}
```

Now, `actual_decorator` is being run. It declares another local function, `inner`, and returns it. As previously discussed, that function is then assigned to the name `do_nothing`, the name of the decorated method. When `do_nothing` is called, the `inner` function is called, runs the decorated method, and JSON dumps the result with the appropriate indentation.

The Call Signature Matters

It is critical to realize that when you introduced your new, altered `json_output` function, you actually introduced a backward-incompatible change.

Why? Because now there is this extra function call that is expected. If you want the old `json_output` behavior, and do not need values for any of the arguments available, you still must call the method.

In other words, you must do the following:

```
@json_output()
def do_nothing():
    return {'status': 'done'}
```

Note the parentheses. They matter, because they indicate that the function is being called (even with no arguments), and then the result is applied to the `@`.

The previous code is *not*—repeat, *not*—equivalent to the following:

```
@json_output
def do_nothing():
    return {'status': 'done'}
```

This presents two problems. It is inherently confusing, because if you are accustomed to seeing decorators applied without a signature, a requirement to supply an empty signature is counterintuitive. Secondly, if the old decorator already exists in your application, you must go back and edit all of its existing calls. You should avoid backward-incompatible changes if possible.

In a perfect world, this decorator would work for three different types of applications:

- `@json_output`
- `@json_output()`
- `@json_output(indent=4)`

As it turns out, this is possible, by having a decorator that modifies its behavior based on the arguments that it receives. Remember, a decorator is just a function and has all the flexibility of any other function to do what it needs to do to respond to the inputs it gets.

Consider this more flexible iteration of `json_output`:

```
import functools
import json


class JSONOutputError(Exception):
    def __init__(self, message):
        self._message = message

    def __str__(self):
        return self._message


def json_output(decorated_=None, indent=None, sort_keys=False):
    """Run the decorated function, serialize the result of that function
    to JSON, and return the JSON string.
    """
    # Did we get both a decorated method and keyword arguments?
    # That should not happen.
    if decorated_ and (indent or sort_keys):
        raise RuntimeError('Unexpected arguments.')

    # Define the actual decorator function.
    def actual_decorator(func):
        @functools.wraps(func)
        def inner(*args, **kwargs):
            try:
                result = func(*args, **kwargs)
            except JSONOutputError as ex:
                result = {
                    'status': 'error',
                    'message': str(ex),
                }
            return json.dumps(result, indent=indent, sort_keys=sort_keys)
        return inner

    # Return either the actual decorator, or the result of applying
    # the actual decorator, depending on what arguments we got.
    if decorated_:
        return actual_decorator(decorated_)
    else:
        return actual_decorator
```

This function is endeavoring to be intelligent about whether or not it is currently being used as a decorator.

First, it makes sure it is not being called in an unexpected way. You never expect to receive both a method to be decorated *and* the keyword arguments, because a decorator is always called with the decorated method as the only argument.

Second, it defines the `actual_decorator` function, which (as its name suggests) is the actual decorator to be either returned or applied. It defines the `inner` function that is the ultimate function to be returned from the decorator.

Finally, it returns the appropriate result based on how it was called:

- If `decorated_` is set, it was called as a plain decorator, without a method signature, and its responsibility is to apply the ultimate decorator and return the `inner` function. Here again, observe how decorators that take arguments are actually working. First, `actual_decorator(decorated_)` is called and resolved, then *its result* (which must be a callable, because this is a decorator) is called with `inner` provided as its only argument.
- If `decorated_` is not set, then this was called with keyword arguments instead, and the function must return an actual decorator, which receives the decorated method and returns `inner`. Therefore, the function returns `actual_decorator` outright. This is then applied by the Python interpreter as the actual decorator (which ultimately returns `inner`).

Why is this technique valuable? It enables you to maintain your decorator's functionality as previously used. This means that you do not have to update each case where the decorator has been applied. But you still get the additional flexibility of being able to add arguments in the cases where you need them.

DECORATING CLASSES

Remember that a decorator is, fundamentally, a callable that accepts a callable and returns a callable. This means that decorators can be used to decorate *classes* as well as functions (classes are callable, after all).

Decorating classes can have a variety of uses. They can be particularly valuable because, like function decorators, class decorators can interact with the attributes of the decorated class. A class decorator can add or augment attributes, or it can alter the API of a class to provide a distinction between how a class is *declared* versus how its instances are *used*.

You might ask, "Isn't the appropriate way to add or augment attributes of a class through sub classing?" Usually, the answer is "yes." However, in some situations an alternative approach may be appropriate. Consider, for example, a generally applicable feature that may apply to many classes in your application that live in distinct places in your class hierarchies.

By way of example, consider a feature of a class such that each instance knows when it was instantiated, and instances are sorted by their creation times. This has general applicability across many different classes, and requires the addition of three attributes—the instantiation timestamp, and the `__gt__` and `__lt__` methods.

You have multiple ways to go about adding this. Here is how you can do it with a class decorator:

```
import functools
import time
```

```
def sortable_by_creation_time(cls):
    """Given a class, augment the class to have its instances be sortable
    by the timestamp at which they were instantiated.
    """
    # Augment the class' original `__init__` method to also store a
    # `_created` attribute on the instance, which corresponds to when it
    # was instantiated.
    original_init = cls.__init__

    @functools.wraps(original_init)
    def new_init(self, *args, **kwargs):
        original_init(self, *args, **kwargs)
        self._created = time.time()
    cls.__init__ = new_init

    # Add `__lt__` and `__gt__` methods that return True or False based on
    # the created values in question.
    cls.__lt__ = lambda self, other: self._created < other._created
    cls.__gt__ = lambda self, other: self._created > other._created

    # Done; return the class object.
    return cls
```

The first thing that is happening in this decorator is that you are saving a copy of the class's original `__init__` method. You do not need to worry about whether the class has one. Because `object` has an `__init__` method, that attribute's presence is guaranteed. Next, you create a new method that will be assigned to `__init__`, and this method first calls the original and then does one piece of extra work, saving the instantiation timestamp to `self._created`.

It is worth noting that this is a very similar pattern to the execution-time wrapping code from previous examples—making a function that wraps another function, whose primary responsibility is to run the wrapped function, but also adds a small piece of other functionality.

It is worth noting that if a class decorated with `@sortable_by_creation_time` defined its own `__lt__` and `__gt__` methods, then this decorator would override them.

The `_created` value by itself does little good if the class does not recognize that it is to be used for sorting. Therefore, the decorator also adds `__lt__` and `__gt__` magic methods. These cause the `<` and `>` operators to return `True` or `False` based on the result of those methods. This also affects the behavior of `sorted` and other similar functions.

This is all that is necessary to make an arbitrary class's instances sortable by their instantiation time. This decorator can be applied to any class, including many classes with unrelated ancestry.

Here is an example of a simple class with instances sortable by when they are created:

```
>>> @sortable_by_creation_time
... class Sortable(object):
...     def __init__(self, identifier):
...         self.identifier = identifier
...     def __repr__(self):
...         return self.identifier
...
>>> first = Sortable('first')
>>> second = Sortable('second')
>>> third = Sortable('third')
>>>
>>> sortables = [second, first, third]
>>> sorted(sortables)
[first, second, third]
```

Bear in mind that simply because a decorator can be used to solve a problem, that does not mean that it is necessarily the appropriate solution.

For instance, when it comes to this example, the same thing could be accomplished by using a "mixin," or a small class that simply defines the appropriate `__init__`, `__lt__`, and `__gt__` methods. A simple approach using a mixin would look like this:

```
import time


class SortableByCreationTime(object):
    def __init__(self):
        self._created = time.time()

    def __lt__(self, other):
        return self._created < other._created

    def __gt__(self, other):
        return self._created > other._created
```

Applying the mixin to a class can be done using Python's multiple inheritance:

```
class MyClass(MySuperclass, SortableByCreationTime):
    pass
```

This approach has different advantages and drawbacks. On the one hand, it will not mercilessly plow over `__lt__` and `__gt__` methods defined by the class or its superclasses (and it may not be obvious when the code is read later that the decorator was clobbering two methods).

On the other hand, it would be very easy to get into a situation where the `__init__` method provided by `SortableByCreationTime` does not run. If `MyClass` *or* `MySuperclass` or any class in `MySuperclass`'s ancestry defines an `__init__` method, it will win out. Reversing the class order does not solve this problem; it simply reverses it.

By contrast, the decorator handles the `__init__` case very well, simply by augmenting the effect of the decorated class's `__init__` method and otherwise leaving it intact.

So, which approach is the correct approach? It depends.

TYPE SWITCHING

Thus far, the discussion in this chapter has only considered cases in which a decorator is expected to decorate a function and provide a function, or when a decorator is expected to decorate a class and provide a class.

There is no reason why this relationship must hold, however. The only requirement for a decorator is that it is a callable that accepts a callable and returns the callable. There is no requirement that it return *the same kind of* callable.

One more advanced use case for decorators is actually when they do not do this. In particular, it can be valuable for a decorator to decorate a function, but return a class. This can be a very useful tool for situations where the amount of boilerplate code grows, or for allowing developers to use a simple function for simple cases, but subclass a class in an application's API for more advanced cases.

An example of this in the wild is a decorator used in a popular task runner in the Python ecosystem: celery. The celery package provides a `@celery.task` decorator that is expected to decorate a function. What the decorator actually does is return a subclass of celery's internal `Task` *class*, with the decorated function being used within the subclass's `run` method.

Consider the following trivial example of a similar approach:

```
class Task(object):
    """A trivial task class. Task classes have a `run` method, which runs
    the task.
    """
    def run(self, *args, **kwargs):
        raise NotImplementedError('Subclasses must implement `run`.')

    def identify(self):
        return 'I am a task.'

def task(decorated):
    """Return a class that runs the given function if its run method is
    called.
    """
    class TaskSubclass(Task):
        def run(self, *args, **kwargs):
            return decorated(*args, **kwargs)
    return TaskSubclass
```

What is happening here? The decorator creates a subclass of `Task` and returns the class. The class is callable calling a class creates an instance of that class and runs its __init __ method

The value of doing this is that it provides a hook for lots of augmentation. The base `Task` class can define much, much more than just the `run` method. For example, a `start` method might run the task asynchronously. The base class might also provide methods to save information about the task's status. Using a decorator that swaps out a function for a class here enables the developer to only consider the actual body of his or her task, and the decorator does the rest of the work.

You can see this in action by taking an instance of the class and running its `identify` method, as shown here:

```
>>> @task
>>> def foo():
>>>     return 2 + 2
>>>
>>> f = foo()
>>> f.run()
4
>>> f.identify()
'I am a task.'
```

A Pitfall

This *exact* approach carries with it some problems. In particular, once a task function is decorated with the `@task_class` decorator, it becomes a class.

Consider the following simple task function decorated in this way:

```
@task
def foo():
    return 2 + 2
```

Now, attempt to run it directly in the interpreter:

```
>>> foo()
<__main__.TaskSubclass object at 0x10c3612d0>
```

That is a bad thing. This decorator alters the function in such a way that if the developer runs it, it does not do what anyone expects. It is usually not acceptable to expect the function to be declared as `foo` and then run using the convoluted `foo().run()` (which is what would be necessary in this case).

Fixing this requires putting a little more thought into how both the decorator and the `Task` class are constructed. Consider the following amended version:

```
class Task(object):
    """A trivial task class. Task classes have a `run` method, which runs
    the task.
    """
    def __call__(self, *args, **kwargs):
        return self.run(*args, **kwargs)

    def run(self, *args, **kwargs):
        raise NotImplementedError('Subclasses must implement `run`.')

    def identify(self):
        return 'I am a task.'


def task(decorated):
    """Return a class that runs the given function if its run method is
    called.
    """
    class TaskSubclass(Task):
        def run(self, *args, **kwargs):
            return decorated(*args, **kwargs)
    return TaskSubclass()
```

A couple of key differences exist here. The first is the addition of the `__call__` method to the base `Task` class. The second difference (which complements the first) is that the `@task_class` decorator now returns *an instance of* the `TaskSubclass`, rather than the class itself.

This is acceptable because the only requirement for the decorator is that it return a callable, and the addition of the `__call__` method to `Task` means that its instances are now callable.

Why is this pattern valuable? Again, the `Task` class is trivial, but it is easy to see how more functionality could be added here that is useful for managing and running tasks.

However, this approach maintains the spirit of the original function if it is invoked directly. Consider the decorated function again:

```
@task
def foo():
    return 2 + 2
```

And now, what do you get if you run it in the interpreter?

```
>>> foo()
4
```

This is what you expect, which makes this a far superior class and decorator design. Under the hood, the decorator has returned a `TaskSubclass` *instance*. When that instance is called in the interpreter, its `__call__` method is invoked, which calls `run`, which calls the original function.

You can see that you still got your instance back, though, by using the `identify` method.

```
>>> foo.identify()
'I am a task.'
```

Now you have an instance that, when called directly, calls exactly like the original function. However, it can include other methods and attributes to provide for other functionality.

This is powerful. It allows a developer to write a function that is easily and explicitly grafted into a class that provides for alternate ways for that function to be invoked or other related functionality. This is a helpful paradigm.

SUMMARY

Decorators are very valuable tools that you can use to write maintainable, readable Python code. A decorator's value is in the fact that it is explicit, as well as the fact that decorators are reusable. They provide an excellent way to use boilerplate code, write it once, and apply it in many different situations.

This useful paradigm is possible because Python's data model provides functions and classes as first-class objects, capable of being passed around and augmented like any other object in the language.

On the other hand, there are also drawbacks to this model. In particular, the decorator syntax, while clean and easy to read, can obscure the fact that a function is being wrapped within another function, which can lead to challenges in debugging. Poorly written decorators may create errors by being careless about the nature of the callables they wrap (for example, by ignoring the distinction between bound methods and unbound functions).

Additionally, bear in mind that, like any function, the interpreter must actually run the code inside the decorator, which has a performance impact. Decorators are not immune to this; be mindful of what you are asking your decorators to do, in the same way that you would be for any other code you write.

Consider using decorators as a way to take leading or trailing functionality and wrap it around unrelated functions. Similarly, decorators are useful tools for function registries, signaling, certain cases of class augmentation, as well as many other things.

Chapter 2 "Context Managers," discusses context managers, which are another way to take bits of functionality that require reuse across an application, and compartmentalize them in an effective and portable way.