Mastering Python Decorators

One of the hallmarks of good Python is the judicious use of decorators to optimize, simplify and add new functionality to existing code. Decorators are usually seen as an advanced topic, but if you read this article and work hard on the exercises I promise that at the end you'll be a decorator wizard.

Before diving in, let's review how functions work. We'll start from the very beginning, so if you're 100% confident you know this stuff, just skip to the section called **Decorators wrap existing functions**.

In Python, functions are defined using the `def` keyword and invoked using parentheses, like so:

```
def foo():
    print('foo!')

foo()
```

Functions in Python are first-class

Unlike languages such as Java, Python treats functions as first-class citizens. That means that the language treats functions and data in the same way. Like data, functions can be assigned to variables, copied, and used as return values. They can also be passed into other functions as parameters. Here's an example:

```
def foo():
    print('foo!')

def bar(fn):
    fn()

bar(foo)   # prints 'foo!'
```

Functions can be defined inside other functions

It is also possible to define functions inside other functions. In the following example, `bar` is defined inside `foo`.

```
def foo()
    def bar():
        print('bar!')
    bar()
```

When `foo` is called, the first thing it does is define `bar`, then it calls the newly defined `bar` function.

An important point to note is that while `foo` is defined in *global scope*, `bar` is defined in the *local scope* of `foo`. What this means from a practical per-

spective is that while `foo` can be called from anywhere in the program, `bar` can only be called from inside `foo`.

```
# Outside foo
foo()    # prints 'bar!'
bar()    # throws an error
```

Now that we've got those preliminaries out of the way, it's time to get started on decorators.

Decorators wrap existing functions

A decorator is a callable that takes a callable as an argument and returns another callable to replace it. In the **decorator execution process** section below, we will go through that definition and unpack exactly what it means, but first let's take a look at an example.

```
def foo(fn):
    def inner():
        print('About to call function.')
        fn()
        print('Finished calling function.')
    return inner

@foo
def bar():
    print('Calling function bar.')
```

There is a lot going on here, but by taking the new elements one by one everything will become clear.

The @ syntax

The first thing to note is the special `@` syntax. In this case, it is used to specify that the function `bar` should be wrapped or *decorated* by `foo`. It may look odd at first, but it is just a form of *syntactic sugar*. The following statement is exactly equivalent to using `@`.

```
bar = foo(bar)
```

Let's think about that for a minute. How exactly would `bar = foo(bar)` work?

The decorator execution process

- A function called `bar` is defined. Its purpose is to print a message.
- A function called `foo` is also defined.
- `foo` takes one parameter, which is itself a function.
- Inside `foo`, another function called `inner` is defined. Because of the way scope works in Python, `inner` has access to the parameters passed into its parent function `foo`.
- `inner` does three things: first, it prints a message saying that it is about to call another function; second, it actually calls the function passed into `foo` as an argument; third, it prints a message saying that it is finished calling the function.
- When `foo` is called with `bar` as an argument, `foo` defines its `inner` function and immediately returns it.

- The return value from `foo` - the `inner` function - is assigned to `bar`, with the result that any time `bar` is called from this point on, it is actually `inner` that is executed.

Here is the result of running the decorated `bar` function:

```
>>> bar()
About to call function.
Calling function bar.
Finished calling function.
```

Function closures

If you're *really* paying attention at this point you might be scratching your head. `foo` returned `inner` and `inner` was assigned to `bar`, but how did `inner` know what function was passed into `foo`? The previous explanation was somewhat simplified: it turns out that `foo` did not really return a function, but a *function closure*. That might sound complicated, but all it means is that before `foo` returned `inner`, it packed up the local variables it had inside it and sent them along too. That is how, when it was called later, the parameter `fn` was available and referred to the function `bar`.

Preserving function metadata

Decorating a function replaces it with something else, so there is the danger that useful metadata about the wrapped function is lost - metadata such as docstrings and the function name. Here is an example:

```
>>> def wrapper(fn):
```

```
        def inner(*args, **kwargs):
            return fn(*args, **kwargs)
        return inner

>>> def hello_world():
        """Prints Hello, world!"""
        print("Hello, world!")

>>> hello_world = wrapper(hello_world)
>>> help(hello_world)
Help on function inner in module __main__:

inner(*args, **kwargs)

>>> hello_world.__name__
'inner'
```

Unfortunately, decorating the `hello_world` function with `wrapper` causes `hello_world` to take on the metadata from `inner`. That makes perfect sense because `hello_world` *does* refer to `inner` after decoration.

It's not really what we want to happen though. To avoid losing the function metadata, `inner` could replace its own metadata with the metadata from the wrapped function. Luckily, there is already a decorator in `functools` to do just that.

```
>>> def wrapper(fn):
        @functools.wraps(fn)
        def inner(*args, **kwargs):
            return fn(*args, **kwargs)
        return inner

>>> hello_world = wrapper(hello_world)
```

```
>>> help(hello_world)
Help on function hello_world in module __main__:

hello_world()
    Prints Hello, world!

>>> hello_world.__name__
'hello_world'
```

From here on, decorator examples will use `functools.wraps` as an example of best practice.

Decorators with parameters

Implementing decorators that accept parameters requires another layer of function nesting. Let's start with a toy example to make it easier to grasp. The decorator below prints the number passed into it before calling the function it is decorating.

```
def print_num(n):
    def decorator(fn):
        @functools.wraps(fn)
        def inner(*args, **kwargs):
            print(n)
            return fn(*args, **kwargs)
        return inner
    return decorator
```

`print_num` is used as follows:

```
@print_num(4)
def foo():
```

```
    # ...
```

In this case, the `@` syntax is equivalent to:

```
foo = print_num(4)(foo)
```

Let's unpack that line of code. First `print_num` is called with the parameter `4`. That returns the `decorator` function from inside `print_num`, which is the real decorator. Then `decorator` is called with the wrapped function as an argument, and it returns its own `inner` function in the normal way. Finally, the inner function is assigned to `foo`.

Decorators with parameters form a big part of the exercises at the end of the chapter, but there's no need to be daunted. We will see plenty of examples before then.

Chaining decorators

The real power of decorators becomes apparent when you start *chaining* them. It turns out that more than one decorator can be applied to a function, like so:

```
@decorator1
@decorator2
def foo():
    print("foo")
```

This can look confusing at first, but, as with single decorators, the key to understanding multiple decorators is to "show your work" and expand the `@` syntax out to basic function calls. Here is what is really going on:

```
foo = decorator1(decorator2(foo))
```

From the inside out, `decorator2` is called with `foo` as an argument. `decorator2` returns its inner function, `inner2`. Then `decorator1` is called with `inner2` as an argument, and it returns its inner function `inner1`. Finally, `inner1` is assigned to `foo`.

At decoration-time, when the decorators are called and return their inner functions, the decorators run *inside out*, i.e. the decorator closest to the function runs first, followed by the second closest, etc. However, when the decorated function is called, the situation is exactly reversed; the inner function of the top decorator runs first, followed by the second from the top, and so on down to the wrapped function. The code below demonstrates these facts about decorator execution order.

```
def decorator1(fn):
    print("decorator1")

    def inner1(*args, **kwargs):
        print("inner1")
        return fn(*args, **kwargs)
    return inner1


def decorator2(fn):
    print("decorator2")

    def inner2(*args, **kwargs):
        print("inner2")
        return fn(*args, **kwargs)
    return inner2
```

```
@decorator1
@decorator2
def foo():
    print("foo")
```

When `foo` is called, it produces the following output:

```
decorator2
decorator1
inner1
inner2
foo
```

Notice how `decorator2` runs before `decorator1` when the function is decorated, but `inner2` runs after `inner1` when the decorated function is called.

Class decorators

Up to this point, all the decorators we have seen have been functions. Sometimes, however, a decorator needs to maintain some state about the different functions it is wrapping. This is useful for, e.g. collection profiling information about a group of functions. In that case, the decorator could refer to some piece of global state outside itself, or it could be implemented as a class. In most cases, the class is the better option.

What follows is a decorator that, when applied to a function, keeps track of the number of times the function is called. It is based on an example from the Python wiki.

```
class countcalls(object):
    __instances = {}
```

```
    def __init__(self, f):
        self.__f = f
        self.__num_calls = 0
        countcalls.__instances[f] = self

    def __call__(self, *args, **kwargs):
        self.__num_calls += 1
        return self.__f(*args, **kwargs)

    def count(self):
        return self.__num_calls

    @staticmethod
    def counts():
        return dict([(f.__name__,
countcalls.__instances[f].__numcalls) for f in
countcalls.__instances])
```

There are two things to note about the class-based constructor above. The first is that the function to be wrapped is passed into `__init__`. The instance keeps a reference to the function it is wrapping and also stores a reference to itself keyed to the wrapped function in a class dictionary called `__instances`. Notice how there is no `return` statement at the end of the constructor, unlike in a function decorator. There is no need to return `self` from `__init__`; constructing an instance of a class automatically returns that instance.

The second thing to note is that the class implements `__call__`, which makes it a `callable`. The `__call__` method in class decorators is equivalent to the `inner` function in function decorators; it is the thing that gets executed

when the wrapped function is called. The code below shows what happens when `countcalls` is used to decorate a function called `foo`.

```
@countcalls
def foo():
    pass

for i in range(10):
    foo()

print(foo.count())           # prints "10"
print(countcalls.counts())  # prints "{'foo': 10}"
```

Calling the `count` method on `foo` returns the number of times `foo` has been called. This works because `foo` no longer refers to the function defined at the top of the code sample, but to an instance of `countcalls`. Calling the class method `counts` returns a dictionary where the keys are the names of each decorated function and the values are the numbers of times each function has been called.

The next section contains examples and explanations of decorators you might find or use in the wild.

Decorator Examples

Timing

A common use of decorators is to time the execution of functions. The following example shows a `timeit` decorator that, when applied to function, prints out the execution time of every invocation.

```
import time

def timeit(fn):
    @functools.wraps(fn)
    def inner(*args, **kwargs):
        start_time = time.time()
        retval = fn(*args, **kwargs)
        duration = time.time() - start_time
        print("%s : %2.2f sec" % (fn.__name__,
duration))
        return retval
    return inner
```

The decorator uses the `time` function in the `time` module from the standard library to store the start time, then calls the wrapped function, then uses the `time` function again to calculate the duration.

Think about how `timeit` could be reimplemented as a class decorator to allow it to collect timing information about multiple functions.

Type checking

As Python is dynamically typed, the types of function parameters are not checked. Often, however, we need to make sure that a particular function only takes certain types of parameters. The traditional way to do this is to just try it and deal with errors as they come up. That can lead to a situation where the function appears to have worked but returns nonsensical data. If further safety is required, pre-conditions can be added to the function implementation.

The decorator below checks the types of the parameters passed into any function and raises an `AssertionError` if they do not match.

```
def accepts(*types):
```

```
    def decorator(fn):
        @functools.wraps(fn)
        def inner(*args, **kwargs):
            for (t, a) in zip(types, args):
                assert isinstance(a, t)
            return fn(*args, **kwargs)
        return inner
    return decorator
```

The `accepts` decorator is used like so:

```
@accepts(int)
def is_prime(n):
    # ...
```

Trying to call `is_prime` with a `float` argument will now result in an error.

```
>>> isprime(3.5)
Traceback (most recent call last):
  File "<pyshell#141>", line 1, in <module>
    isprime(3.5)
  File "<pyshell#136>", line 5, in inner
    assert isinstance(a, t)
AssertionError
```

The implementation above only enforces the types of `*args`. Think about how it could be improved to check `**kwargs` too.

Deprecation

Using the `warnings` module, a deprecation warning can be issued when when a decorated function is called.

```
import warnings
```

```
def deprecated(fn):
    @functools.wraps(fn)
    def inner(*args, **kwargs):
        warnings.warn("Deprecated:
{}".format(fn.__name__),
                      category=DeprecationWarning)
        return fn(*args, **kwargs)

    return inner
```

Think about how the decorator could be modified to only print the deprecation warning for certain Python versions.

Memoization

Memoization is an optimization technique that can be implemented elegantly using decorators. It is worth considering for functions that are both *pure* and *expensive*. Pure means that the function always returns the same output for the same inputs, and expensive means that it takes a while to run. Recursive functions with lots of nested subcalls, such as the naive implementation of the factorial function, are prime candidates for memoization. Here is a memoization decorator that could be applied to that function.

```
def memoize(fn):
    cache = {}
    @functools.wraps(fn)
    def inner(n):
        if n not in cache:
            cache[n] = fn(n)
        return cache[n]
    return inner
```

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

The inner function in `memoize` is very simple. It just looks in the `cache` dictionary to check if a value for `n` has already been computed. If it has not, it calls `fn` and stores the return value in `cache`. The `cache` dictionary is available because it is captured in the closure.

The implementation of the `memoize` decorator above is specific to functions that take a single positional argument. It would be relatively easy to modify the decorator so that the dictionary keys in `cache` are unique representations of a combination of `*args` and `**kwargs`, but it is not necessary. Since Python 3.2, there is a memoization decorator in the `functools` module called `lru_cache` that can be used to either cache the results of the `n` most recent function calls, or the results of an indefinite number of calls, depending on the value of its `maxsize` parameter. The default value of `maxsize` is `128`.

```
import functools

@functools.lru_cache(maxsize=256)
def factorial(n):
    # ...

@functools.lru_cache(maxsize=None)
def factorial(n):
```

```
    # ...
```

The next section contains exercises. It is the most important part of the article because it tests your understanding of the material. The exercises start off easy, but get progressively harder, so don't worry if you get stuck. Just take a break and come back to them later.

Exercises

1. Write a decorator that can be applied to any function and will print out the values of the parameters passed into it.
2. Write a decorator that, when applied to a function, causes an assertion error if the function returns `None`.
3. Write a decorator that caches the `n` most recent return values from the function it is applied to, so that the function does not have to be invoked again.
4. Write a decorator that caches all return values of the function it is applied to for `n` seconds.
5. Write a decorator can be applied to a function that makes retries the function `n` times with delay `m` between retries if the function throws an exception. If, after `n` retries, the function has still not succeeded, re-raise the exception

Solutions

Here are some possible solutions for the exercises in the last section. It's ok if your code isn't exactly the same. As with many things in programming, there's more than one way to do it.

1. Using the `pprint` function from the `pprint` module makes it easy to print the positional and keyword arguments just before calling the decorated function.

```
from pprint import pprint

def print_args(fn):
    @functools.wraps(fn)
    def inner(*args, **kwargs):
        pprint(args)
        pprint(kwargs)
        return fn(*args, **kwargs)
    return inner
```

2. Throwing an assertion error if the decorated function returns `None` is a simple matter of catching the return value of the decorated function and asserting that it is not equal to `None`.

```
def prevent_none(fn):
    @functools.wraps(fn)
    def inner(*args, **kwargs):
        retval = fn(*args, **kwargs)
        assert retval is not None
        return retval
    return inner
```

3. The solution for this exercise is effectively a reimplementation of the `lru_cache` decorator from `functools`, minus some optimizations.

```
import collections

def cache_n(maxsize):
    def decorator(fn):
        recent =
collections.deque(maxlen=maxsize)
        cache = {}

        @functools.wraps(fn)
        def inner(*args, **kwargs):
            key = str(args) + str(kwargs)
            if key not in cache:
                cache[key] = fn(*args,
**kwargs)

            if key in recent:
                recent.remove(key)

            if len(recent) == maxsize:
                del cache[recent.pop()]

            recent.appendleft(key)
            return cache[key]
        return inner
    return decorator
```

4. The solution for this exercise is a modification of the memoization decorator from the examples, except that the cache stores the return value of the function and the time if was calculated.

```
def cache_seconds(n):
    def decorator(fn):
        cache = {}

        @functools.wraps(fn)
        def inner(*args, **kwargs):
            key = str(args)+str(kwargs)
            now = time.time()
            if key in cache and (cache[key][0]
+ n) > now:
                result = cache[key][1]
            else:
                result = fn(*args, **kwargs)
            cache[key] = (now, result)
            return result

        return inner
    return decorator
```

5. Retrying with a delay can be achieved using a loop, a `try except` block, and the `raise` keyword for re-raising caught exceptions.

```
def retry_function(times, delay):
    def decorator(fn):

        @functools.wraps(fn)
        def inner(*args, **kwargs):
            for i in range(times):
                try:
                    return fn(*args, **kwargs)
                except:
                    if i == times-1:
                        raise
```

```
                time.sleep(delay)
        return inner
    return decorator
```

That's it and thanks for reading. This article is extracted from my upcoming book on advanced Python features for intermediate programmers.