7 Methods and decorators

Python provides decorators as a handy way to modify functions. They were first introduced with `classmethod()` and `staticmethod()` in Python 2.2, but were overhauled through PEP 318 into something more flexible and readable. Python provides a few decorators (including the two mentioned above) right out of the box, but it seems that most developers don't understand how they actually work behind the scenes. This chapter aims to change that.

7.1 Creating decorators

A decorator is essentially a function that takes another function as an argument and replaces it with a new, modified function. Odds are good you've already used decorators to make your own wrapper functions. The simplest possible decorator is the identity function, which does nothing except return the original function:

```python
def identity(f):
    return f
```

You can then use your decorator like this:

```python
@identity
def foo():
    return 'bar'
```

Which is the same as:

```
def foo():
    return 'bar'


foo = identity(foo)
```

This decorator is useless, but it works. It just does nothing.

Example 7.1 A registering decorator

```
_functions = {}
def register(f):
    global _functions
    _functions[f.__name__] = f
    return f
```

@register def foo(): return *bar*

In this example, we register and store functions in a dictionary so we can retrieve them by their name later from that dictionary.

In the following sections, I'll explain the standard decorators that Python provides and how (and when) to use them.

The primary use case for decorators is factoring common code that needs to be called before, after, or around multiple function. If you ever wrote Emacs Lisp code you may have used *defadvice* that allows you to define code called around a function. Same things apply for developers having used the fabulous method combinations brought by CLOS [1].

Consider a set of functions that are called and need to check that the user name that they receive as argument:

```
class Store(object):
    def get_food(self, username, food):
```

[1] The Common Lisp Object System

```
        if username != 'admin':
            raise Exception("This user is not allowed to get food")
        return self.storage.get(food)

    def put_food(self, username, food):
        if username != 'admin':
            raise Exception("This user is not allowed to get food")
        self.storage.put(food)
```

The obvious first step here is to factor the checking code:

```
def check_is_admin(username):
    if username != 'admin':
        raise Exception("This user is not allowed to get food")

class Store(object):
    def get_food(self, username, food):
        check_is_admin(username)
        return self.storage.get(food)

    def put_food(self, username, food):
        check_is_admin(username)
        self.storage.put(food)
```

Now our code looks a bit cleaner. But we can do even better if we use a decorator:

```
def check_is_admin(f):
    def wrapper(*args, **kwargs):
        if kwargs.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper
```

```
class Store(object):
    @check_is_admin
    def get_food(self, username, food):
        return self.storage.get(food)

    @check_is_admin
    def put_food(self, username, food):
        self.storage.put(food)
```

Using decorators like this makes it easier to manage common functionality. This is probably old hat to you if you have any serious Python experience, but what you might not realize is that this naive approach to implementing decorators has some major drawbacks.

As mentioned before, a decorator replaces the original function with a new one built on-the-fly. However, this new function lacks many of the attributes of the original function, such as its docstring and its name:

```
>>> def is_admin(f):
...     def wrapper(*args, **kwargs):
...         if kwargs.get('username') != 'admin':
...             raise Exception("This user is not allowed to get food")
...         return f(*args, **kwargs)
...     return wrapper
...
>>> def foobar(username="someone"):
...     """Do crazy stuff."""
...     pass
...
>>> foobar.func_doc
'Do crazy stuff.'
>>> foobar.__name__
```

```
'foobar'
>>> @is_admin
... def foobar(username="someone"):
...     """Do crazy stuff."""
...     pass
...
>>> foobar.__doc__
>>> foobar.__name__
'wrapper'
```

Fortunately, the **functools** module included in Python solves this problem with the `update_wrapper` function, which copies these attributes to the wrapper itself. The source code of `update_wrapper` is self-explanatory:

Example 7.2 Source code of `functools.update_wrapper` in Python 3.3

```
WRAPPER_ASSIGNMENTS = ('__module__', '__name__', '__qualname__', '__doc__',
                       '__annotations__')
WRAPPER_UPDATES = ('__dict__',)
def update_wrapper(wrapper,
                   wrapped,
                   assigned = WRAPPER_ASSIGNMENTS,
                   updated = WRAPPER_UPDATES):
    wrapper.__wrapped__ = wrapped
    for attr in assigned:
        try:
            value = getattr(wrapped, attr)
        except AttributeError:
            pass
        else:
            setattr(wrapper, attr, value)
    for attr in updated:
        getattr(wrapper, attr).update(getattr(wrapped, attr, {}))
    # Return the wrapper so this can be used as a decorator via partial()
    return wrapper
```

If we take our previous example and use this function to update our wrapper, things work much more nicely:

```
>>> def foobar(username="someone"):
...     """Do crazy stuff."""
...     pass
...
>>> foobar = functools.update_wrapper(is_admin, foobar)
>>> foobar.__name__
'foobar'
>>> foobar.__doc__
'Do crazy stuff.'
```

It can get tedious to use `update_wrapper` manually when creating decorators, so **functools** provides a decorator for decorators called `wraps`:

Example 7.3 Using `functools.wraps`

```
import functools


def check_is_admin(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        if kwargs.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper


class Store(object):
    @check_is_admin
    def get_food(self, username, food):
        return self.storage.get(food)
```

In our examples so far, we've always assumed that the decorated function would have a `username` passed to it as a keyword argument, but that might not always be the case. With this in mind, it's a better idea to build a smarter version of our decorator that can look at the decorated function's arguments and pull out what it needs.

To that end, the **inspect** module allows us to retrieve a function's signature and operate on it:

Example 7.4 Retrieving function arguments using **inspect**

```
import functools
import inspect


def check_is_admin(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        func_args = inspect.getcallargs(f, *args, **kwargs)
        if func_args.get('username') != 'admin':
            raise Exception("This user is not allowed to get food")
        return f(*args, **kwargs)
    return wrapper


@check_is_admin
def get_food(username, type='chocolate'):
    return type + " nom nom nom!"
```

The function that does the heavy lifting here is `inspect.getcallargs`, which returns a dictionary containing the names and values of the arguments as key-value pairs.

In our example, this function returns `{'username':'admin', 'type':'chocolate'}`. This means that our decorator doesn't have to check if the `username` parameter is a positional or a keyword argument: all it has to do is look for it in the dictionary.

7.2 How methods work in Python

You've probably written dozens of methods and thought nothing of them before now, but to understand what certain decorators do, you need to know how methods work behind the scenes.

A method is a function that is stored as a class attribute. Let's have a look at what happens when we try to access such an attribute directly:

Example 7.5 A Python 2 method

```
>>> class Pizza(object):
...     def __init__(self, size):
...         self.size = size
...     def get_size(self):
...         return self.size
...
>>> Pizza.get_size
<unbound method Pizza.get_size>
```

Python 2 tells us that the `get_size` attribute of the `Pizza` class is an **unbound** method.

Example 7.6 A Python 3 method

```
>>> class Pizza(object):
...     def __init__(self, size):
...         self.size = size
...     def get_size(self):
...         return self.size
...
```

```
>>> Pizza.get_size
<function Pizza.get_size at 0x7fdbfd1a8b90>
```

In Python 3, the concept of unbound method has been removed entirely, and we're told `get_size` is a function.

The principle is the same in both cases: `get_size` is a function that is not tied to any particular object, and Python will raise an error if we try to call it:

Example 7.7 Calling unbound get_size in Python 2

```
>>> Pizza.get_size()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method get_size() must be called with Pizza instance as  ↩
   first argument (got nothing instead)
```

Example 7.8 Calling unbound get_size in Python 3

```
>>> Pizza.get_size()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: get_size() missing 1 required positional argument: 'self'
```

Python 2 rejects the method call because it's unbound; Python 3 permits the call, but complains that we haven't provided the necessary `self` argument. This makes Python 3 a bit more flexible: not only can we pass an arbitrary instance of the class to the method if we want to, but we can pass *any* object as long as it has the properties that the method expects to find:

```
>>> Pizza.get_size(Pizza(42))
42
```

And it works, just as promised, though it's not very convenient: we have to refer to the class every time we want to call one of its methods.

So Python goes the extra mile for us by binding a class's methods to its instances. In other words, we can access `get_size` from any `Pizza`, and better still, Python will automatically pass the object itself to the method's `self` parameter:

Example 7.9 Calling bound `get_size`

```
>>> Pizza(42).get_size
<bound method Pizza.get_size of <__main__.Pizza object at 0x7f3138827910>>
>>> Pizza(42).get_size()
42
```

As expected, we don't have to provide any argument to `get_size`, since it's a bound method: its `self` argument is automatically set to our `Pizza` instance. Here's a even better example:

```
>>> m = Pizza(42).get_size
>>> m()
42
```

You don't even have to keep a reference to your `Pizza` object as long as you have a reference to the bound method. And if you have a reference to a method but you want to find out which object it's bound to, you can just check the method's `__self__` property:

```
>>> m = Pizza(42).get_size
>>> m.__self__
<__main__.Pizza object at 0x7f3138827910>
>>> m == m.__self__.get_size
True
```

Obviously, we still have a reference to our object, and we can find it back if we want.

7.3 Static methods

Static methods are methods which belong to a class, but don't actually operate on class instances. For example:

Example 7.10 `@staticmethod` usage

```
class Pizza(object):
    @staticmethod
    def mix_ingredients(x, y):
        return x + y

    def cook(self):
        return self.mix_ingredients(self.cheese, self.vegetables)
```

You could write `mix_ingredients` as a non-static method if you wanted to, but it would take a `self` argument that would never actually be used. The `@staticmethod` decorator gives us several things:

- Python doesn't have to instantiate a bound method for each `Pizza` object we create. Bound methods are objects, too, and creating them has a cost. Using a static method lets us avoid that:

```
>>> Pizza().cook is Pizza().cook
False
>>> Pizza().mix_ingredients is Pizza.mix_ingredients
True
>>> Pizza().mix_ingredients is Pizza().mix_ingredients
True
```

- It improves the readability of the code: when we see `@staticmethod`, we know that the method does not depend on the state of the object.

- We can override our static methods in subclasses. If we used a mix_ingredients function defined at the top level of our module, a class inheriting from Pizza wouldn't be able to change the way we mix ingredients for our pizza without overriding the cook method itself.

7.4 Class method

Class methods are methods that are bound directly to a class rather than its instances:

```
>>> class Pizza(object):
...     radius = 42
...     @classmethod
...     def get_radius(cls):
...         return cls.radius
...
>>> Pizza.get_radius
<bound method type.get_radius of <class '__main__.Pizza'>>
>>> Pizza().get_radius
<bound method type.get_radius of <class '__main__.Pizza'>>
>>> Pizza.get_radius is Pizza().get_radius
True
>>> Pizza.get_radius()
42
```

However you choose to access this method, it will be always bound to the class it is attached to, and its first argument will be the class itself (remember, classes are objects too!)

Class methods are mostly useful for creating *factory methods* – methods which instantiate objects in a specific fashion. If we used a @staticmethod instead, we would have to hard-code the Pizza class name in our method, making any class inheriting from Pizza unable to use our factory for its own purposes.

```
class Pizza(object):
    def __init__(self, ingredients):
        self.ingredients = ingredients

    @classmethod
    def from_fridge(cls, fridge):
        return cls(fridge.get_cheese() + fridge.get_vegetables())
```

In this case, we provide a `from_fridge` factory method that we can pass a `Fridge` object to. If we call this method with something like `Pizza.from_fridge(myfridge)`, it will return a brand-new `Pizza` with ingredients taken from what's available in `myfridge`.

7.5 Abstract methods

An abstract method is a method defined in a base class which may or may not actually provide any implementation. The simplest way to write an abstract method in Python is:

```
class Pizza(object):
    @staticmethod
    def get_radius():
        raise NotImplementedError
```

Any class inheriting from `Pizza` should implement and override the `get_radius` method; otherwise, calling the method will raise an exception.

This particular way of implementing abstract methods has a drawback: if you write a class that inherits from `Pizza` and forget to implement `get_radius`, the error will only be raised if you try to use that method at runtime.

Example 7.11 Implementing an abstract method

```
>>> Pizza()
<__main__.Pizza object at 0x7fb747353d90>
>>> Pizza().get_radius()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in get_radius
NotImplementedError
```

If you implement your abstract methods using Python's built-in abc module instead, you'll get an early warning if you try to instantiate an object with abstract methods:

Example 7.12 Implementing an abstract method using `abc`

```
import abc


class BasePizza(object):
    __metaclass__  = abc.ABCMeta


    @abc.abstractmethod
    def get_radius(self):
         """Method that should do something."""
```

When you use `abc` and its special class, if you try to instantiate a `BasePizza` or a class inheriting from it that doesn't override `get_radius`, you'll get a `TypeError`:

```
>>> BasePizza()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class BasePizza with abstract methods ↩
    get_radius
```

Note

The metaclass declaration changed between Python 2 and Python 3. The previous examples only work with Python 2 for this reason.

7.6 Mixing static, class, and abstract methods

Each of these decorators is useful on its own, but the time may come when you'll have to use them together. Here are some tips that will help you with that.

An abstract method's prototype isn't set in stone. When you actually implement the method, there's nothing stopping you from extending the argument list as you see fit:

```
import abc


class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    @abc.abstractmethod
    def get_ingredients(self):
         """Returns the ingredient list."""


class Calzone(BasePizza):
    def get_ingredients(self, with_egg=False):
        egg = Egg() if with_egg else None
        return self.ingredients + [egg]
```

We can define `Calzone`'s methods any way we like, as long as they still support the interface we define in the `BasePizza` class. This includes implementing them as class or static methods:

```python
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    @abc.abstractmethod
    def get_ingredients(self):
         """Returns the ingredient list."""

class DietPizza(BasePizza):
    @staticmethod
    def get_ingredients():
        return None
```

Even though our static `get_ingredients` method doesn't return a result based on the object's state, it still supports our abstract `BasePizza` class's interface, so it's still valid.

Starting with Python 3 (this won't work as expected in Python 2; see issue 5867), it's also possible to use the `@staticmethod` and `@classmethod` decorators on top of `@abstractmethod`:

Example 7.13 Mixing `@classmethod` and `@abstractmethod`

```python
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    ingredients = ['cheese']

    @classmethod
    @abc.abstractmethod
    def get_ingredients(cls):
         """Returns the ingredient list."""
         return cls.ingredients
```

Note that defining `get_ingredients` as a class method in `BasePizza` like this doesn't force its subclasses to define it as a class method as well. The same would apply if we'd defined it as a static method: there's no way to force subclasses to implement abstract methods as a specific kind of method.

But wait – here we have an implementation *in* an abstract method. Can we *do* that? Yep – Python doesn't have a problem with it! Unlike Java, you can put code in your abstract methods and call it using `super()`:

Example 7.14 Using `super()` with abstract methods

```
import abc

class BasePizza(object):
    __metaclass__  = abc.ABCMeta

    default_ingredients = ['cheese']

    @classmethod
    @abc.abstractmethod
    def get_ingredients(cls):
         """Returns the default ingredient list."""
         return cls.default_ingredients

class DietPizza(BasePizza):
    def get_ingredients(self):
        return [Egg()] + super(DietPizza, self).get_ingredients()
```

In this example, every `Pizza` you make that inherits from `BasePizza` will have to override the `get_ingredients` method, but it will have access to the base class's default mechanism for getting the ingredients list.

7.7 The truth about `super`

From the earliest days of Python, developers have been able to use both single and multiple inheritance to extend their classes. However, many developers don't seem to understand how these mechanisms actually work, and the associated `super()` method that is associated with it.

There is pros and cons of single and multiple inheritance, composition or even duck typing would be out of topic for this book, though if you are not familiar with these notions I suggest that you read about them to have a view – and build your own opinion.

Multiple inheritance is still used in many places, and especially in code where the mixin pattern is involved. That's why it's still important to know about it, and because it is part of Python's core.

Note

A mixin is a class that inherits from two or more other classes, combining their features together.

As you should know by now, classes are objects in Python. The construct used to create a class is a special statement that you should be well familiar with: `class classname(expression of inheritance)`.

The part in parentheses is a Python expression that returns the list of class objects to be used as the class's parents. Normally you'd specify them directly, but you could also write something like:

```
>>> def parent():
...     return object
...
>>> class A(parent()):
...     pass
...
>>> A.mro()
[<class '__main__.A'>, <type 'object'>]
```

And it works as expected: class `A` is defined with `object` as its parent class. The class method `mro()` returns the *method resolution order* used to resolve attributes. The current MRO system was first implemented in Python 2.3, and its internal workings are described in the Python 2.3 release notes.

You already know that the canonical way to call a method in a parent class is by using the `super()` function, but what you probably don't know is that `super()` is actually a constructor, and you instantiate a `super` object each time you call it. It takes either one or two arguments: the first argument is a class, and the second argument is either a subclass or an instance of the first argument.

The object returned by the constructor functions as a proxy for the parent classes of the first argument. It has its own `__getattribute__` method that iterates over the classes in the MRO list and returns the first matching attribute it finds:

```
>>> class A(object):
...     bar = 42
...     def foo(self):
...             pass
...
>>> class B(object):
...     bar = 0
...
```

```
>>> class C(A, B):
...     xyz = 'abc'
...
>>> C.mro()
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <type ' ↩
   object'>]
>>> super(C, C()).bar
42
>>> super(C, C()).foo
<bound method C.foo of <__main__.C object at 0x7f0299255a90>>
>>> super(B).__self__
>>> super(B, B()).__self__
<__main__.B object at
```

When requesting an attribute of the super object of an instance of `C`, it walks through the MRO list and return the attribute from the first class having it.

In the previous example, we used a bound `super` object; i.e., we called `super` with two arguments. If we call `super()` with only one argument, it returns an unbound `super` object instead:

```
>>> super(C)
<super: <class 'C'>, NULL>
```

Since this object is unbound, you can't use it to access class attributes:

```
>>> super(C).foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'super' object has no attribute 'foo'
>>> super(C).bar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
AttributeError: 'super' object has no attribute 'bar'
>>> super(C).xyz
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'super' object has no attribute 'xyz'
```

At first glance, it might seem like this kind of `super` object is useless, but the `super` class implements the descriptor protocol (i.e. `__get__`) in a way that makes unbound `super` objects useful as class attributes:

```
>>> class D(C):
...     sup = super(C)
...
>>> D().sup
<super: <class 'C'>, <D object>>
>>> D().sup.foo
<bound method D.foo of <__main__.D object at 0x7f0299255bd0>>
>>> D().sup.bar
42
```

The unbound `super` object's `__get__` method is called using the instance and the attribute name as arguments (`super(C).__get__(D(), 'foo')`), allowing it to find and resolve `foo`.

Note

Even if you've never heard of the descriptor protocol, you've probably used it through the `@property` decorator without knowing it. It's the mechanism in Python that allows an object that's stored as an attribute to return something other than itself. This protocol isn't covered in this book, but you can find out more about it in the Python data model documentation.

There are plenty of situations where using `super` can be tricky, such as handling different method signatures along the inheritance chain. Unfortunately, there's no silver bullet for that, apart from using tricks like having all your methods accept their arguments using `*args, **kwargs`.

In Python 3, `super()` picked up a little bit of magic: it can now be called from within a method without any arguments. When no arguments are passed to `super()`, it automatically searches the stack frame for them:

```
class B(A):
      def foo(self):
          super().foo()
```

`super` is the standard way of accessing parent attributes in subclasses, and you should always use it. It allows cooperative calls of parent methods without any surprises, such as parent methods not being called or being called twice when using multiple inheritance.