

The definitive guide on how to use static, class or abstract methods in Python

Thursday
01 August 2013

Doing code reviews is a great way to discover things that people might struggle to comprehend. While proof-reading OpenStack patches (<http://review.openstack.org>) recently, I spotted that people were not using correctly the various decorators Python provides for methods. So here's my attempt at providing me a link to send them to in my next code reviews. :-)

 **python**
(/blog
/tags/python)



How methods work in Python

A method is a function that is stored as a class attribute. You can declare and access such a function this way:

```
>>> 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>
```

What Python tells you here, is that the attribute `get_size` of the class `Pizza` is a method that is **unbound**. What does this mean? We'll know as soon as we'll try to call it:

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

We can't call it because it's not bound to any instance of `Pizza`. And a method wants an instance as its first argument (in Python 2 it **must** be an instance of that class; in Python 3 it could be anything). Let's try to do that then:




Julien
Danjou


Free Software hacker
([http://en.wikipedia.org/wiki/Hacker_\(programmer_developer_and_consultant\)](http://en.wikipedia.org/wiki/Hacker_(programmer_developer_and_consultant)))
working **freelance (/)**.

Project Technical Leader for **OpenStack Ceilometer** (<http://launchpad.net/ceilometer>), former member of the **OpenStack technical committee** (<http://openstack.org>)

Also a **Debian developer** (<http://debian.org>), the original author of the **awesome window manager** (<http://awesome.naquadah.org>) and a developer of **GNU Emacs**.

 (/#contact)

 (/blog

/index.xml) 

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

It worked! We called the method with an instance as its first argument, so everything's fine. But you will agree with me if I say this is not a very handy way to call methods; we have to refer to the class each time we want to call a method. And if we don't know what class is our object, this is not going to work for very long.

So what Python does for us, is that it binds all the methods from the class *Pizza* to any instance of this class. This means that the attribute *get_size* of an instance of *Pizza* is a bound method: a method for which the first argument will be the instance itself.

```
>>> 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 bound, its *self* argument is automatically set to our *Pizza* instance. Here's a even better proof of that:

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

Indeed, you don't even have to keep a reference to your *Pizza* object. Its method is bound to the object, so the method is sufficient to itself.

But what if you wanted to know which object this bound method is bound to? Here's a little trick:

```
>>> m = Pizza(42).get_size
>>> m.__self__
<__main__.Pizza object at 0x7f3138827910>
>>> # You could guess, Look at this:
...
>>> m == m.__self__.get_size
True
```

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

In Python 3, the functions attached to a class are not considered as *unbound method* anymore, but as simple functions, that are bound to an object if required. So the principle stays the same,

(<https://github.com/jd>)



(<http://twitter.com>

/juldanjou)

(<http://plus.google.com/108793797474967511035?pr>



(<http://linkedin.com>

/in/julienDanjou)

Tweets

Follow



Julien Danjou 20 Mar
@juldanjou

I'm now available to talk about #Python and various stuff on @bbl_fr :)

Expand



Julien Danjou 18 Mar
@juldanjou

Tomorrow is the 7th OpenStack France meetup, join us to talk about SDN meetup.com/OpenStack-France
Show Summary



Julien Danjou 17 Mar

Tweet to @juldanjou

the model is just simplified.

```
>>> 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 0x7f307f984dd0>
```

Static methods

Static methods are a special case of methods. Sometimes, you'll write code that belongs to a class, but that doesn't use the object itself at all. For example:

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

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

In such a case, writing *mix_ingredients* as a non-static method would work too, but it would provide it a *self* argument that would not be used. Here, the decorator *@staticmethod* buys us several things:

- Python doesn't have to instantiate a bound-method for each *Pizza* object we instantiate. Bound methods are objects too, and creating them has a cost. Having a static method avoids that:

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

- It eases the readability of the code: seeing *@staticmethod*, we know that the method does not depend on the state of object itself;
- It allows us to override the *mix_ingredients* method in a subclass. If we used a function *mix_ingredients* 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 *cook* itself.

Class methods

Having said that, what are class methods? Class methods are methods that are not bound to an object, but to... a class!

```
>>> 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
```

Whatever the way you use to access this method, it will be always bound to the class it is attached too, and its first argument will be the class itself (remember that classes are objects too).

When to use this kind of methods? Well class methods are mostly useful for two types of methods:

- Factory methods, that are used to create an instance for a class using for example some sort of pre-processing. If we use a *@staticmethod* instead, we would have to hardcode the *Pizza* class name in our function, making any class inheriting from *Pizza* unable to use our factory for its own use.

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

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

- Static methods calling static methods: if you split a static methods in several static methods, you shouldn't hard-code the class name but use class methods. Using this way to declare our method, the *Pizza* name is never directly referenced and inheritance and method overriding will work flawlessly

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

    @staticmethod
    def compute_circumference(radius):
        return math.pi * (radius ** 2)

    @classmethod
    def compute_volume(cls, height, radius):
        return height * cls.compute_circumference(radius)

    def get_volume(self):
        return self.compute_volume(self.height, self.radius)
```

Abstract methods

An abstract method is a method defined in a base class, but that may not provide any implementation. In Java, it would describe the methods of an interface.

So the simplest way to write an abstract method in Python is:

```
class Pizza(object):
    def get_radius(self):
        raise NotImplementedError
```

Any class inheriting from *Pizza* should implement and override the *get_radius* method, otherwise an exception would be raised.

This particular way of implementing abstract method has a drawback. If you write a class that inherits from *Pizza* and forget to implement *get_radius*, the error will only be raised when you'll try to use that 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
```

There's a way to trigger this way earlier, when the object is being instantiated, using the *abc* (<http://docs.python.org/2/library/abc.html>) module that's provided with Python.

```
import abc

class BasePizza(object):
    __metaclass__ = abc.ABCMeta

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

Using *abc* and its special class, as soon as you'll try to instantiate *BasePizza* or any class inheriting from it, 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
```

Mixing static, class and abstract methods

When building classes and inheritances, the time will come where you will have to mix all these methods decorators. So here's some tips about it.

Keep in mind that declaring a class as being abstract, doesn't freeze the prototype of that method. That means that it must be implemented, but it can be implemented with any argument list.

```
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
```

This is valid, since *Calzone* fulfil the interface requirement we defined for *BasePizza* objects. That means that we could also implement it as being a class or a static method, for example:

```
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
```

This is also correct and fulfil the contract we have with our abstract *BasePizza* class. The fact that the *get_ingredients* method don't need to know about the object to return result is an implementation detail, not a criteria to have our contract fulfilled.

Therefore, you can't force an implementation of your abstract method to be a regular, class or static method, and arguably you shouldn't. Starting with Python 3 (this won't work as you would expect in Python 2, see [issue5867](http://bugs.python.org/issue5867) (<http://bugs.python.org/issue5867>)), it's now possible to use the *@staticmethod* and *@classmethod* decorators on top of *@abstractmethod*:

```
import abc

class BasePizza(object):
    __metaclass__ = abc.ABCMeta

    ingredient = ['cheese']

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

Don't misread this: if you think this going to force your subclasses to implement *get_ingredients* as a class method, you are wrong. This simply implies that your implementation of *get_ingredients* in the *BasePizza* class is a class method.

An implementation in an abstract method? Yes! In Python, contrary to methods in Java interfaces, you can have code in your abstract methods and call it via *super()*:

```
import abc

class BasePizza(object):
    __metaclass__ = abc.ABCMeta

    default_ingredients = ['cheese']

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

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

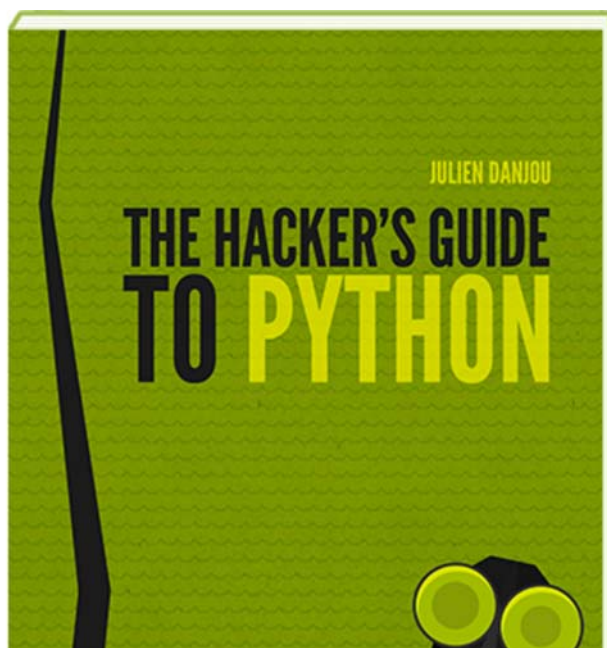
In such a case, every pizza you will build by inheriting from *BasePizza* will have to override the *get_ingredients* method, but will be able to use the default mechanism to get the ingredient list by using *super()*.

The Hacker's Guide to Python (/books/the-hacker-guide-to-python)

A book I'm writing that will be launched soon, talking about designing Python applications, state of the art, and various Python tips.

If you want to be the first to hear about the launch, subscribe now.

<input type="checkbox"/>	<input type="text"/>	<input type="button" value="Keep me updated"/>
--------------------------	----------------------	------------------------------------------------





(/books/the-hacker-guide-to-python)

Enjoyed this article?

Share it:

Tweet 27 127 Share 1

Recommend 23 Receive next articles by email:

← **Previous post:**

OpenStack Ceilometer Havana-2 milestone released (/blog/2013/openstack-ceilometer-havana-2-milestone-released)

Next post: Announcing The

Hacker's Guide to Python →
(/blog/2013/announcing-the-hacker-guide-to-python)

Created by Julien Danjou (<http://julien.danjou.info>) using Hyde (<https://github.com/hyde/hyde>). Sources (<http://git.naquadah.org/?p=~jd/julien.danjou.info.git;a=summary>).

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License (http://creativecommons.org/licenses/by-sa/3.0/deed.en_US).



(http://creativecommons.org/licenses/by-sa/3.0/deed.en_US)