Difference between staticmethod and classmethod

https://stackoverflow.com/a/1669524

Chris B.Chris B. 2,37411 gold badge1414 silver badges99 bronze badges Tue Aug, 03 14:52

What is the difference between @staticmethod and @classmethod in Python?

You may have seen Python code like this pseudocode, which demonstrates the signatures of the various method types and provides a docstring to explain each:

```
class Foo(object):

    def a_normal_instance_method(self, arg_1, kwarg_2=None):
        '''
        Return a value that is a function of the instance with its
        attributes, and other arguments such as arg_1 and kwarg2
        '''

    @staticmethod
    def a_static_method(arg_0):
        '''
        Return a value that is a function of arg_0. It does not know the
        instance or class it is called from.
        '''

    @classmethod
    def a_class_method(cls, arg1):
        '''
        Return a value that is a function of the class and other arguments.
        respects subclassing, it is called with the class it is called from.
        '''
```

The Normal Instance Method

First I'll explain `a_normal_instance_method`. This is precisely called an '**instance method**'. When an instance method is used, it is used as a partial function (as opposed to a total function, defined for all values when viewed in source code) that is, when used, the first of the arguments is predefined as the instance of the object, with all of its given attributes. It has the instance of the object bound to it, and it must be called from an instance of the object. Typically, it will access various attributes of the instance.

For example, this is an instance of a string:

```
', '
```

if we use the instance method, `join` on this string, to join another iterable, it quite obviously is a function of the instance, in addition to being a function of the iterable list, `['a', 'b', 'c']`:

```
>>> ', '.join(['a', 'b', 'c'])
'a, b, c'
```

Bound methods

Instance methods can be bound via a dotted lookup for use later.

For example, this binds the `str.join` method to the `':'` instance:

```
>>> join_with_colons = ':'.join
```

And later we can use this as a function that already has the first argument bound to it. In this way, it works like a partial function on the instance:

```
>>> join_with_colons('abcde')
'a:b:c:d:e'
>>> join_with_colons(['FF', 'FF', 'FF', 'FF', 'FF', 'FF'])
'FF:FF:FF:FF:FF:FF'
```

Static Method

The static method does *not* take the instance as an argument.

It is very similar to a module level function.

However, a module level function must live in the module and be specially imported to other places where it is used.

If it is attached to the object, however, it will follow the object conveniently through importing and inheritance as well.

An example of a static method is `str.maketrans`, moved from the `string` module in Python 3. It makes a translation table suitable for consumption by `str.translate`. It does seem rather silly when used from an instance of a string, as demonstrated below, but importing the function from the `string` module is rather clumsy, and it's nice to be able to call it from the class, as in `str.maketrans`

```
# demonstrate same function whether called from instance or not:
>>> ', '.maketrans('ABC', 'abc')
{65: 97, 66: 98, 67: 99}
>>> str.maketrans('ABC', 'abc')
{65: 97, 66: 98, 67: 99}
```

In python 2, you have to import this function from the increasingly less useful string module:

```
>>> import string
>>> 'ABCDEFG'.translate(string.maketrans('ABC', 'abc'))
'abcDEFG'
```

Class Method

A class method is a similar to an instance method in that it takes an implicit first argument, but instead of taking the instance, it takes the class. Frequently these are used as alternative constructors for better semantic usage and it will support inheritance.

The most canonical example of a builtin classmethod is `dict.fromkeys`. It is used as an alternative constructor of dict, (well suited for when you know what your keys are and want a default value for them.)

```
>>> dict.fromkeys(['a', 'b', 'c'])
{'c': None, 'b': None, 'a': None}
```

When we subclass dict, we can use the same constructor, which creates an instance of the subclass.

```
>>> class MyDict(dict): 'A dict subclass, use to demo classmethods'
>>> md = MyDict.fromkeys(['a', 'b', 'c'])
>>> md
{'a': None, 'c': None, 'b': None}
>>> type(md)
<class '__main__.MyDict'>
```

See the pandas source code for other similar examples of alternative constructors, and see also the official Python documentation on `classmethod` and `staticmethod`.