`staticmethod` vs `classmethod`

Descriptors

- Descriptors let objects customize attribute lookup, storage, and deletion.
- `@property`, `@staticmethod`, `@classmethod` are pre-defined decorators in Python and are also called as `descriptors`.

NOTE

- The core idea is not to repeat yourself.
- We will do comparison of 3 methods at the end
  - normal instance method
  - static method
  - class method

`Staticmethod`

- When a method inside a class is decorated with `@staticmethod`, the function becomes identically accessible from either an object or a class.
- `@staticmethod` do NOT refer to `self` variable
- Function that is defined as `@staticmethod` does not have `self` in the parameter list.
- Neither `self`, nor `cls` is implicit argument. BUT, you can invoke function using instance of class, class name or also, by `self`.

```
class ABC:
    @staticmethod
    def stat_foo:
        ...
        ...

# all 3 below are valid -

# Invocation of static method outside the class
>>> ABC.stat_foo()
>>> ABC().stat_foo()

# Invokation of static method inside the class (under instance method)
self.stat_foo()
```

Why do use `@staticmethod`?

- `staticmethod` are utility methods.
- `staticmethod` have a very clear use case.
- When we need some functionality not w.r.t an Object but w.r.t the complete class, we make a method as `staticmethod`.
- Functions that are conceptually related to the class but do not depend on data bundled in the class.
- Computational, transformational methods that do not have dependency on class data.
- For instance, `square(num)` is handy conversion routine that comes up in statistical work but does not directly depend on a particular dataset.

`classmethod`

- `staticmethod` shouldn't be confused with `classmethod`.
- `classmethod` methods are bound to a class rather than an object.
- Class methods can be called by both class and object.

- `classmethod` must have a reference to a class object as the first parameter (denoted as `cls`).
- These methods can be called with a class or with an object.

Why do use `@classmethod`?

- Classes may have class attributes and instance attributes.
- When a instance method is invoked, as it has instance as an implicit attribute (`self`).
- It may uselessly refer to class attribute which has no business with it being an instance.
- Typical use cases are factory methods.
- **Factory methods**, that are used to create an instance for a class using for example some sort of pre-processing.
- **Example -**

```
"""

Let's assume that we want to create a lot of Date class instances
having date information coming from an outer source encoded as a string
with format 'dd-mm-yyyy'.

Suppose we have to do this in different places in the source code of our project.
"""

class MyDate(object):
    def __init__(self, d=0, m=0, y=0):
        self.day = d
        self.month = m
        self.year = y

    @classmethod
    def from_string(cls, date_as_string):
        dd, mm, yy = map(int, date_as_string.split("-"))
        date = cls(dd, mm, yy)
        return date


string_date = "11-09-2012"
day, month, year = map(int, string_date.split("-"))
```

```
date1 = MyDate(day, month, year)
date2 = MyDate.from_string("11-09-2012")

print(date1)
print(date2)

"""
Basically we create a model class (OR a data-class) and
Then to fit certain data entity into class struct.
You would want to do certain manipulations on data entity.

In the scenario such as this, you would define a `@classmethod` inside a class.
And would invoke it by passing data entity to it.
And your manipulation logic sits within `@classmethod`.


-------------
CONCLUSION -
-------------
Basically, we try to do little  pre-processing for creating an instance of class.

"""
```

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Impact with counter inside constructor

```
class Robot:
    __counter = 0
    def __init__(self):
        type(self).__counter += 1
```

```
    @classmethod
    def robot_instances(cls):
        return cls, Robot.__counter


if __name__ == "__main__":

    # calling from class name
    print(Robot.robot_instances())

    # calling using object name
    x = Robot()
    print(x.robot_instances())

    # creating another object and checking impact
    y = Robot()
    print(x.robot_instances())
    print(Robot.robot_instances())
```

Comparison and notes

```
class Apple:

    _counter = 0

    @staticmethod
    def about_apple():
        print('Apple is good for you.')

        # note you can still access other member of the class
        # but you have to use the class instance
        # which is not very nice, because you have repeat yourself
        #
        # For example:
        # @staticmethod
        #    print('Number of apples have been juiced: %s' % Apple._counter)
        #
        # @classmethod
        #    print('Number of apples have been juiced: %s' % cls._counter)
        #
        # @classmethod is especially useful when you move
        # your function to another class,
        # You don't have to rename the referenced class

    @classmethod
    def make_apple_juice(cls, number_of_apples):
        print('Making juice:')
        for i in range(number_of_apples):
```

```
            cls._juice_this(i)

    @classmethod
    def _juice_this(cls, apple):
        print('Juicing apple %d...' % apple)
        cls._counter += 1
```

Another perspective

```
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.
        '''
```

Class method vs Static Method

- A class method takes `cls` as the first parameter while a static method needs no specific parameters.
- A class method can access or modify the class state while a static method can’t access or modify it.
- In general, static methods know nothing about the class state. They are utility-type methods that take some parameters and work upon those parameters. On the other hand class methods must have class as a parameter.
- We use `@classmethod` decorator in python to create a class method and we use `@staticmethod` decorator to create a static method in python.

💡 Classes may have class attributes and instance attributes.

When a instance method is invoked, as it has instance as an implicit attribute (`self`). It may uselessly refer to class attribute which has no business with it being an instance. Also, the fact that you can not access class attribute by using static method outside of class without creating an instance of that class. Thereby, wasting memory by creating an instance only to access class attribute.

```
class Calculator:

    # create addNumbers static method
    @staticmethod
    def addNumbers(x, y):
        return x + y

print('Product:', Calculator.addNumbers(15, 110))
```

Another example -

```
from datetime import date as dt


class Employee:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @staticmethod
    def isAdult(age):
        if age > 18:
            return True
        else:
            return False

    @classmethod
    def emp_from_year(emp_class, name, year):
        return emp_class(name, dt.today().year - year)
```

```
    def __str__(self):
        return "Employee Name: {} and Age: {}".format(self.name, self.age)


e1 = Employee("Dhiman", 25)
print(e1)
e2 = Employee.emp_from_year("Subhas", 1987)
print(e2)
print(Employee.isAdult(25))
print(Employee.isAdult(16))
```

...