**Classmethod and staticmethod python**

Though classmethod and staticmethod are quite similar, there's a slight difference in usage for both entities: classmethod must have a reference to a class object as the first parameter, whereas staticmethod can have no parameters at all.

Let's look at all that was said in real examples.

**Boilerplate**

Let's assume an example of a class, dealing with date information (this is what will be our boilerplate to cook on):

class Date(object):

    day = 0

    month = 0

    year = 0

    def \_\_init\_\_(self, day=0, month=0, year=0):

        self.day = day

        self.month = month

        self.year = year

This class obviously could be used to store information about certain dates (without timezone information; let's assume all dates are presented in UTC).

Here we have \_\_init\_\_, a typical initializer of Python class instances, which receives arguments as a typical instancemethod, having the first non-optional argument (self) that holds reference to a newly created instance.

**Class Method**

We have some tasks that can be nicely done using classmethods.

*Let's assume that we want to create a lot of Date class instances having date information coming from outer source encoded as a string of next format ('dd-mm-yyyy'). We have to do that in different places of our source code in project.*

So what we must do here is:

1. Parse a string to receive day, month and year as three integer variables or a 3-item tuple consisting of that variable.
2. Instantiate Date by passing those values to initialization call.

This will look like:

day, month, year = map(int, string\_date.split('-'))

date1 = Date(day, month, year)

For this purpose, C++ has such feature as overloading, but Python lacks that feature- so here's when classmethod applies. Lets create another "*constructor*".

    @classmethod

    def from\_string(cls, date\_as\_string):

        day, month, year = map(int, date\_as\_string.split('-'))

        date1 = cls(day, month, year)

        return date1

date2 = Date.from\_string('11-09-2012')

Let's look more carefully at the above implementation, and review what advantages we have here:

1. We've implemented date string parsing in one place and it's reusable now.
2. Encapsulation works fine here (if you think that you could implement string parsing as a single function elsewhere, this solution fits OOP paradigm far better).
3. cls is an object that holds **class itself**, not an instance of the class. It's pretty cool because if we inherit our Date class, all children will have from\_string defined also.

**Static method**

What about staticmethod? It's pretty similar to classmethod but doesn't take any obligatory parameters (like a class method or instance method does).

Let's look at the next use case.

*We have a date string that we want to validate somehow. This task is also logically bound to Date class we've used so far, but still doesn't require instantiation of it.*

Here is where staticmethod can be useful. Let's look at the next piece of code:

    @staticmethod

    def is\_date\_valid(date\_as\_string):

        day, month, year = map(int, date\_as\_string.split('-'))

        return day <= 31 and month <= 12 and year <= 3999

    # usage:

    is\_date = Date.is\_date\_valid('11-09-2012')

So, as we can see from usage of staticmethod, we don't have any access to what the class is- it's basically just a function, called syntactically like a method, but without access to the object and it's internals (fields and another methods), while classmethod does.

Another example

class Date:

  def \_\_init\_\_(self, month, day, year):

    self.month = month

    self.day   = day

    self.year  = year

  def display(self):

    return "{0}-{1}-{2}".format(self.month, self.day, self.year)

  @staticmethod

  def millenium(month, day):

    return Date(month, day, 2000)

new\_year = Date(1, 1, 2013)               # Creates a new Date object

millenium\_new\_year = Date.millenium(1, 1) # also creates a Date object.

# Proof:

new\_year.display()           # "1-1-2013"

millenium\_new\_year.display() # "1-1-2000"

isinstance(new\_year, Date) # True

isinstance(millenium\_new\_year, Date) # True

Thus both new\_year and millenium\_new\_year are instances of Date class.

But, if you observe closely, the Factory process is hard-coded to create Date objects no matter what. What this means is that even if the Date class is subclassed, the subclasses will still create plain Date object (without any property of the subclass). See that in the example below:

class DateTime(Date):

  def display(self):

      return "{0}-{1}-{2} - 00:00:00PM".format(self.month, self.day, self.year)

datetime1 = DateTime(10, 10, 1990)

datetime2 = DateTime.millenium(10, 10)

isinstance(datetime1, DateTime) # True

isinstance(datetime2, DateTime) # False

datetime1.display() # returns "10-10-1990 - 00:00:00PM"

datetime2.display() # returns "10-10-2000" because it's not a DateTime object but a Date object. Check the implementation of the millenium method on the Date class

datetime2 is not an instance of DateTime? WTF? Well that's because of the @staticmethod decorator used.

In most cases, this is undesired. If what you want is a Factory method that is aware of the class that called it, then @classmethod is what you need.

Rewriting the Date.millenium as (that's the only part of the above code that changes)

@classmethod

def millenium(cls, month, day):

    return cls(month, day, 2000)

ensures that the class is not hard-coded but rather learnt. cls can be any subclass. The resulting object will rightly be an instance of cls. Let's test that out.

datetime1 = DateTime(10, 10, 1990)

datetime2 = DateTime.millenium(10, 10)

isinstance(datetime1, DateTime) # True

isinstance(datetime2, DateTime) # True

datetime1.display() # "10-10-1990 - 00:00:00PM"

datetime2.display() # "10-10-2000 - 00:00:00PM"