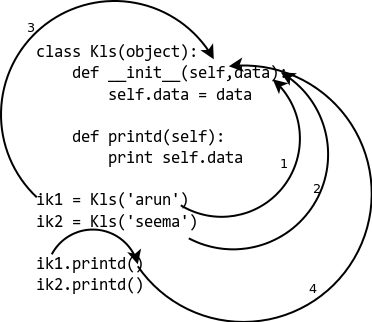
For example, a basic instance method would be as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Kls(object):      def \_\_init\_\_(self, data):          self.data = data        def printd(self):          print(self.data)    ik1 = Kls('arun')  ik2 = Kls('seema')    ik1.printd()  ik2.printd() |

This gives us the following output:

|  |  |
| --- | --- |
| 1  2 | Arun  seema |

[](http://www.pythoncentral.io/wp-content/uploads/2013/02/instancemethod.png)

After looking at the code sample and diagram:

* In 1 and 2, the arguments are passed to the method.
* On 3, the self argument refers to the instance.
* At 4, we do not need to provide the instance to the method, as it is handled by the interpretor itself.

Now what if the method we want to write interacts with classes only and not instances? We can code a simple function out of the class to do so but that will spread the code related to class, to out of the class. This can cause a future code maintenance problem, as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | def get\_no\_of\_instances(cls\_obj):      return cls\_obj.no\_inst    class Kls(object):      no\_inst = 0        def \_\_init\_\_(self):          Kls.no\_inst = Kls.no\_inst + 1    ik1 = Kls()  ik2 = Kls()    print(get\_no\_of\_instances(Kls)) |

Gives us the following output:

|  |  |
| --- | --- |
| 1 | 2 |

The Python @classmethod

What we want to do now is create a function in a class, which gets the class object to work on instead of the instance. If we want to get the no of instances, all we have to do is something like below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def iget\_no\_of\_instance(ins\_obj):      return ins\_obj.\_\_class\_\_.no\_inst    class Kls(object):      no\_inst = 0        def \_\_init\_\_(self):      Kls.no\_inst = Kls.no\_inst + 1    ik1 = Kls()  ik2 = Kls()  print iget\_no\_of\_instance(ik1) |

|  |  |
| --- | --- |
| 1 | 2 |

Using features introduced after Python 2.2, we can create a method in a class, using @classmethod.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class Kls(object):      no\_inst = 0        def \_\_init\_\_(self):          Kls.no\_inst = Kls.no\_inst + 1        @classmethod      def get\_no\_of\_instance(cls\_obj):          return cls\_obj.no\_inst    ik1 = Kls()  ik2 = Kls()    print ik1.get\_no\_of\_instance()  print Kls.get\_no\_of\_instance() |

We get the following output:

|  |  |
| --- | --- |
| 1  2 | 2  2 |

The benefit of this is: whether we call the method from the instance or the class, it passes the class as first argument.

The Python @staticmethod

Often there is some functionality that relates to the class, but does not need the class or any instance(s) to do some work. Perhaps something like setting environmental variables, changing an attribute in another class, etc. In these situation we can also use a function, however doing so also spreads the interrelated code which can cause maintenance issues later.

This is a sample case:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | IND = 'ON'    def checkind():      return (IND == 'ON')    class Kls(object):       def \_\_init\_\_(self,data):          self.data = data    def do\_reset(self):      if checkind():          print('Reset done for:', self.data)    def set\_db(self):      if checkind():          self.db = 'new db connection'          print('DB connection made for:',self.data)    ik1 = Kls(12)  ik1.do\_reset()  ik1.set\_db() |

Which gives us the following output:

|  |  |
| --- | --- |
| 1  2 | Reset done for: 12  DB connection made for: 12 |

Here if we use a @staticmethod, we can place all code in the relevant place.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | IND = 'ON'    class Kls(object):      def \_\_init\_\_(self, data):          self.data = data        @staticmethod      def checkind():          return (IND == 'ON')        def do\_reset(self):          if self.checkind():              print('Reset done for:', self.data)        def set\_db(self):          if self.checkind():              self.db = 'New db connection'          print('DB connection made for: ', self.data)    ik1 = Kls(12)  ik1.do\_reset()  ik1.set\_db() |

Which gives us the following output:

|  |  |
| --- | --- |
| 1  2 | Reset done for: 12  DB connection made for: 12 |

Here is a more comprehensive code example, with a diagram to show you

How @staticmethod and @classmethod are different.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Kls(object):      def \_\_init\_\_(self, data):          self.data = data        def printd(self):          print(self.data)        @staticmethod          def smethod(\*arg):              print('Static:', arg)        @classmethod          def cmethod(\*arg):              print('Class:', arg) |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | >>> ik = Kls(23)  >>> ik.printd()  23  ()>>> ik.smethod  Static: ()  >>> ik.cmethod()  Class: (<class '\_\_main\_\_.Kls'>,)  >>> Kls.printd()  TypeError: unbound method printd() must be called with Kls instance as first argument (got nothing instead)  >>> Kls.smethod()  Static: ()  >>> Kls.cmethod()  Class: (<class '\_\_main\_\_.Kls'>,) |

Here's a diagram to explain what's going on:

