Page author

This page was written by **Bernd Klein**.

Bernd is an experienced computer scientist with a history of working in the education management industry and is skilled in Python, Perl, Computer Science, and C++. He has a Dipl.-Informatiker / Master Degree focused in Computer Science from Saarland University.

- Bernd Klein on Facebook
- Bernd Klein on LinkedIn
- python-course on Facebook

In this Object Oriented Programming chapter

Intro to Object Oriented Programming
1. Object Oriented Programming
2. Class vs. Instance Attributes
3. Properties vs. Getters and Setters
4. Implementing a Custom Property Class
5. Introduction to Descriptors
6. Inheritance
7. Multiple Inheritance
8. Multiple Inheritance: Example
9. Magic Methods
10. Callable Instances of Classes
11. Inheritance Example
12. Slots: Avoiding Dynamically Created Attributes
13. Polynomial Class
14. Dynamically Creating Classes with type
15. Road to Metaclasses
16. Metaclasses
17. Count Function calls with the help of a Metaclass
18. The 'ABC' of Abstract Base Classes

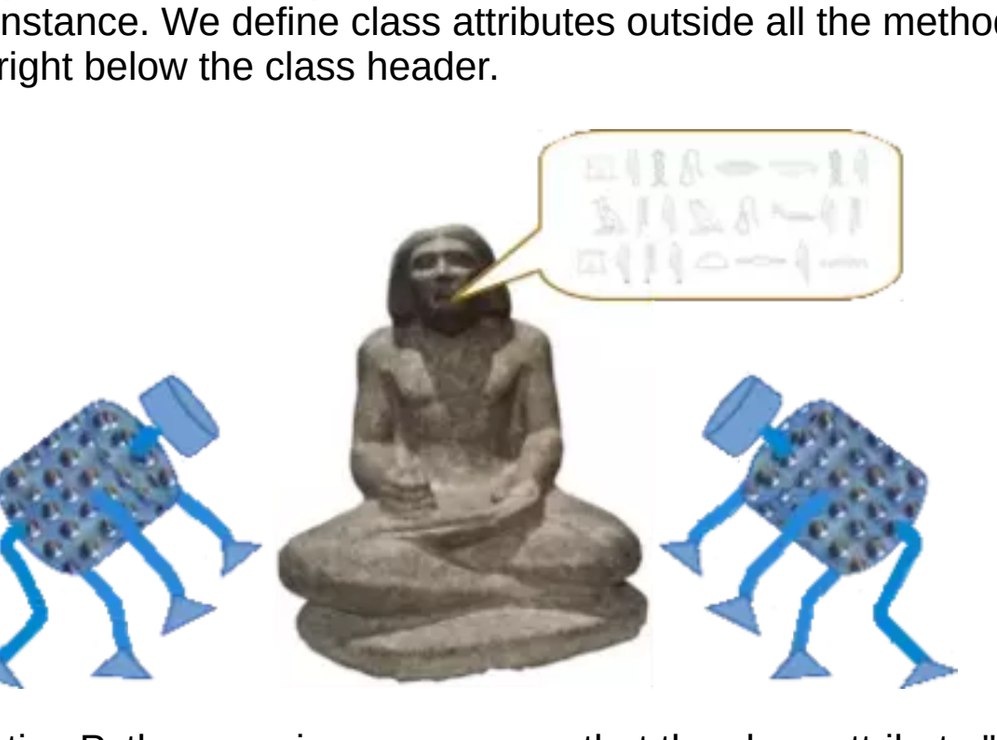
2. Class vs. Instance Attributes

By **Bernd Klein**. Last modified: 01 Feb 2022.

Class Attributes

Instance attributes are owned by the specific instances of a class. That is, for two different instances, the instance attributes are usually different. You should by now be familiar with this concept which we introduced in our previous chapter.

We can also define attributes at the class level. Class attributes are attributes which are owned by the class itself. They will be shared by all the instances of the class. Therefore they have the same value for every instance. We define class attributes outside all the methods, usually they are placed at the top, right below the class header.



In the following interactive Python session, we can see that the class attribute "a" is the same for all instances, in our examples "x" and "y". Besides this, we see that we can access a class attribute via an instance or via the class name:

```
class A:
    a = "I am a class attribute!"
x = A()
y = A()
x.a
```

OUTPUT:

```
'I am a class attribute!'
```

```
y.a
```

OUTPUT:

```
'I am a class attribute!'
```

```
A.a
```

OUTPUT:

```
'I am a class attribute!'
```

But be careful, if you want to change a class attribute, you have to do it with the notation ClassName.AttributeName. Otherwise, you will create a new instance variable. We demonstrate this in the following example:

```
class A:
    a = "I am a class attribute!"
x = A()
y = A()
x.a = "This creates a new instance attribute for x!"
```

```
y.a
```

OUTPUT:

```
'I am a class attribute!'
```

```
A.a
```

OUTPUT:

```
'I am a class attribute!'
```

```
A.a = "This is changing the class attribute 'a'!"
```

```
A.a
```

OUTPUT:

```
"This is changing the class attribute 'a'!"
```

```
y.a
```

OUTPUT:

```
"This is changing the class attribute 'a'!"
```

```
x.a
# but x.a is still the previously created instance variable
```

OUTPUT:

```
'This creates a new instance attribute for x!'
```

Python's class attributes and object attributes are stored in separate dictionaries, as we can see here:

```
x.__dict__
```

OUTPUT:

```
{'a': 'This creates a new instance attribute for x!'}
```

```
y.__dict__
```

OUTPUT:

```
{}
```

```
A.__dict__
```

OUTPUT:

```
mappingproxy({'__module__': '__main__',
              'a': "This is changing the class attribute 'a'!",
              '__dict__': <attribute '__dict__' of 'A' objects>,
              '__weakref__': <attribute '__weakref__' of 'A' objects>,
              '__doc__': None})
```

```
x.__class__.__dict__
```

OUTPUT:

```
mappingproxy({'__module__': '__main__',
              'a': "This is changing the class attribute 'a'!",
              '__dict__': <attribute '__dict__' of 'A' objects>,
              '__weakref__': <attribute '__weakref__' of 'A' objects>,
              '__doc__': None})
```


Example with Class Attributes

Isaac Asimov devised and introduced the so-called "Three Laws of Robotics" in 1942. The appeared in his story "Runaround". His three laws have been picked up by many science fiction writers. As we have started manufacturing robots in Python, it's high time to make sure that they obey Asimovs three laws. As they are the same for every instance, i.e. robot, we will create a class attribute Three_Laws. This attribute is a tuple with the three laws.

```
class Robot:
    Three_Laws = (
"""A robot may not injure a human being or, through inaction, allow a human being to come to harm.""",
"""A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.,""",
"""A robot must protect its own existence as long as such protection does not conflict with the First or Second Law."""
)
    def __init__(self, name, build_year):
        self.name = name
        self.build_year = build_year
    # other methods as usual
```

As we mentioned before, we can access a class attribute via instance or via the class name. You can see in the following that we don't need an instance:

```
from robot_asimov import Robot
for number, text in enumerate(Robot.Three_Laws):
    print(str(number+1) + ":\n" + text)
```

OUTPUT:

```
1:
A robot may not injure a human being or, through inaction, allow a
human being to come to harm.
2:
A robot must obey the orders given to it by human beings, except where
such orders would conflict with the First Law.,
3:
A robot must protect its own existence as long as such protection does
not conflict with the First or Second Law.
```

In the following example, we demonstrate, how you can count instance with class attributes. All we have to do is

- to create a class attribute, which we call "counter" in our example
- to increment this attribute by 1 every time a new instance is created
- to decrement the attribute by 1 every time an instance is destroyed

```
class C:
    counter = 0
    def __init__(self):
        type(self).counter += 1
    def __del__(self):
        type(self).counter -= 1
if __name__ == "__main__":
    x = C()
    print("Number of instances: : " + str(C.counter))
    y = C()
    print("Number of instances: : " + str(C.counter))
    del x
    print("Number of instances: : " + str(C.counter))
    del y
    print("Number of instances: : " + str(C.counter))
```

OUTPUT:

```
Number of instances: : 1
Number of instances: : 2
Number of instances: : 1
Number of instances: : 0
```

Principially, we could have written C.counter instead of type(self).counter, because type(self) will be evaluated to "C" anyway. However we will understand later, that type(self) makes sense, if we use such a class as a superclass.

Static Methods

We used class attributes as public attributes in the previous section. Of course, we can make public attributes private as well. We can do this by adding the double underscore again. If we do so, we need a possibility to access and change these private class attributes. We could use instance methods for this purpose:

```
class Robot:
    __counter = 0
    def __init__(self):
        type(self).__counter += 1
    def RobotInstances(self):
        return Robot.__counter
if __name__ == "__main__":
    x = Robot()
    print(x.RobotInstances())
    y = Robot()
    print(x.RobotInstances())
```

OUTPUT:

```
1
2
```

This is not a good idea for two reasons: First of all, because the number of robots has nothing to do with a single robot instance and secondly because we can't inquire the number of robots before we create an instance. If we try to invoke the method with the class name Robot.RobotInstances(), we get an error message, because it needs an instance as an argument:

```
Robot.RobotInstances()
```

OUTPUT:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-f53600e3296e> in <module>
----> 1Robot.RobotInstances()
TypeError: RobotInstances() missing 1 required positional argument: 'self'
```

The next idea, which still doesn't solve our problem, is omitting the parameter "self":

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

Now it's possible to access the method via the class name, but we can't call it via an instance:

```
from static_methods2 import Robot
Robot.RobotInstances()
```

OUTPUT:

```
0
```

```
x = Robot()
x.RobotInstances()
```

OUTPUT:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-4d5e11c3474a> in <module>
      1 x = Robot()
----> 2x.RobotInstances()
TypeError: RobotInstances() takes 0 positional arguments but 1 was given
```

The call "x.RobotInstances()" is treated as an instance method call and an instance method needs a reference to the instance as the first parameter.

So, what do we want? We want a method, which we can call via the class name or via the instance name without the necessity of passing a reference to an instance to it.

The solution lies in static methods, which don't need a reference to an instance. It's easy to turn a method into a static method. All we have to do is to add a line with "@staticmethod" directly in front of the method header. It's the decorator syntax.

You can see in the following example that we can now use our method RobotInstances the way we want:

```
class Robot:
    __counter = 0
    def __init__(self):
        type(self).__counter += 1
    @staticmethod
    def RobotInstances():
        return Robot.__counter
if __name__ == "__main__":
    print(Robot.RobotInstances())
    x = Robot()
    print(x.RobotInstances())
    y = Robot()
    print(x.RobotInstances())
    print(Robot.RobotInstances())
```

OUTPUT:

```
0
1
2
2
```


Class Methods

Static methods shouldn't be confused with class methods. Like static methods class methods are not bound to instances, but unlike static methods class methods are bound to a class. The first parameter of a class method is a reference to a class, i.e. a class object. They can be called via an instance or the class name.

```
class Robot:
    __counter = 0
    def __init__(self):
        type(self).__counter += 1
    @classmethod
    def RobotInstances(cls):
        return cls, Robot.__counter
if __name__ == "__main__":
    print(Robot.RobotInstances())
    x = Robot()
    print(x.RobotInstances())
    y = Robot()
    print(x.RobotInstances())
    print(Robot.RobotInstances())
```

OUTPUT:

```
(<class '__main__.Robot'>, 0)
(<class '__main__.Robot'>, 1)
(<class '__main__.Robot'>, 2)
(<class '__main__.Robot'>, 2)
```

The use cases of class methods:

- They are used in the definition of the so-called factory methods, which we will not cover here.
- They are often used, where we have static methods, which have to call other static methods. To do this, we would have to hard code the class name, if we had to use static methods. This is a problem, if we are in a use case, where we have inherited classes.

The following program contains a fraction class, which is still not complete. If you work with fractions, you need to be capable of reducing fractions, e.g. the fraction 8/24 can be reduced to 1/3. We can reduce a fraction to lowest terms by dividing both the numerator and denominator by the Greatest Common Divisor (GCD).

We have defined a static gcd function to calculate the greatest common divisor of two numbers. the greatest common divisor (gcd) of two or more integers (at least one of which is not zero), is the largest positive integer that divides the numbers without a remainder. For example, the 'GCD of 8 and 24 is 8. The static method "gcd" is called by our class method "reduce" with "cls.gcd(n1, n2)". "CLS" is a reference to "fraction".

```
class fraction(object):
    def __init__(self, n, d):
        self.numerator, self.denominator = fraction.reduce(n, d)
    @staticmethod
    def gcd(a,b):
        while b != 0:
            a, b = b, a%b
        return a
    @classmethod
    def reduce(cls, n1, n2):
        g = cls.gcd(n1, n2)
        return (n1 // g, n2 // g)
    def __str__(self):
        return str(self.numerator)+'/'+str(self.denominator)
```

Using this class:

```
from fraction1 import fraction
x = fraction(8,24)
print(x)
```

OUTPUT:

```
1/3
```

Class Methods vs. Static Methods and Instance Methods

Our last example will demonstrate the usefulness of class methods in inheritance. We define a class `Pet` with a method `about`. This method should give some general class information. The class `Cat` will be inherited both in the subclass `Dog` and `Cat`. The method `about` will be inherited as well. We will demonstrate that we will encounter problems, if we define the method `about` as a normal instance method or as a static method. We will start by defining `about` as an instance method:

```
class Pet:
    _class_info = "pet animals"
    def about(self):
        print("This class is about " + self._class_info + "!")
class Dog(Pet):
    _class_info = "man's best friends"
class Cat(Pet):
    _class_info = "all kinds of cats"
p = Pet()
p.about()
d = Dog()
d.about()
c = Cat()
c.about()
```

OUTPUT:

```
This class is about pet animals!
This class is about man's best friends!
This class is about all kinds of cats!
```

This may look alright at first at first glance. On second thought we recognize the awful design. We had to create instances of the `Pet`, `Dog` and `Cat` classes to be able to ask what the class is about. It would be a lot better, if we could just write `Pet.about()`, `Dog.about()` and `Cat.about()` to get the previous result. We cannot do this. We will have to write `Pet.about(p)`, `Dog.about(d)` and `Cat.about(c)` instead.

Now, we will define the method `about` as a "staticmethod" to show the disadvantage of this approach. As we have learned previously in our tutorial, a staticmethod does not have a first parameter with a reference to an object. So about will have no parameters at all. Due to this, we are now capable of calling "about" without the necessity of passing an instance as a parameter, i.e. `Pet.about()`, `Dog.about()` and `Cat.about()`. Yet, a problem lurks in the definition of `about`. The only way to access the class info `_class_info` is putting a class name in front. We arbitrarily put in `Pet`. We could have put there `Cat` or `Dog` as well. No matter what we do, the solution will not be what we want:

```
class Pet:
    _class_info = "pet animals"
    @staticmethod
    def about():
        print("This class is about " + Pet._class_info + "!")
class Dog(Pet):
    _class_info = "man's best friends"
class Cat(Pet):
    _class_info = "all kinds of cats"
Pet.about()
Dog.about()
Cat.about()
```

OUTPUT:

```
This class is about pet animals!
This class is about pet animals!
This class is about pet animals!
```

In other words, we have no way of differenciating between the class `Pet` and its subclasses `Dog` and `Cat`. The problem is that the method `about` does not know that it has been called via the `Pet` the `Dog` or the `Cat` class.

A classmethod is the solution to all our problems. We will decorate `about` with a classmethod decorator instead of a staticmethod decorator:

```
class Pet:
    _class_info = "pet animals"
    @classmethod
    def about(cls):
        print("This class is about " + cls._class_info + "!")
class Dog(Pet):
    _class_info = "man's best friends"
class Cat(Pet):
    _class_info = "all kinds of cats"
Pet.about()
Dog.about()
Cat.about()
```

OUTPUT:

```
This class is about pet animals!
This class is about man's best friends!
This class is about all kinds of cats!
```


« PREVIOUS: 1. Object Oriented Programming

NEXT: 3. Properties vs. Getters and Setters »