CLASS AND INSTANCE ATTRIBUTES

CLASS ATTRIBUTES

Instance attributes are owned by the specific instances of a class. This means for two different instances the instance attributes are usually different. You should by now be familiar with this concept which we introduced the 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 of all the methods, usually they are placed at the top, right below the class header.

We can see in the following interactive Python session that the class attribute "a" is the same for all instances, in our example "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
'I am a class attribute!'
>>> y.a
'I am a class attribute!'
>>> A.a
'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
'I am a class attribute!'
>>> A.a
'I am a class attribute!'
>>> A.a = "This is changing the class attribute 'a'!"
>>> A.a
"This is changing the class attribute 'a'!"
>>> y.a
"This is changing the class attribute 'a'!"
>>> # but x.a is still the previously created instance variable:
...
>>> x.a
'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__
{'a': 'This creates a new instance attribute for x!'}
>>> y.__dict__
{}
>>> A.__dict__
dict_proxy({'a': "This is changing the class attribute 'a'!",
'__dict__': <attribute '__dict__' of 'A' objects>, '__module__':
'__main__', '__weakref__': <attribute '__weakref__' of 'A'
objects>, '__doc__': None})
>>> x.__class__.__dict__
dict_proxy({'a': "This is changing the class attribute 'a'!",
'__dict__': <attribute '__dict__' of 'A' objects>, '__module__':
'__main__', '__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 need no instance:

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

We demonstrate in the following example, 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 will be create
- to decrement the attribute by 1 every time an instance will be 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))
```

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

If we start the previous program, we will get the following results:

```
$ python3 counting_instances.py
Number of instances: : 1
Number of instances: : 2
Number of instances: : 1
Number of instances: : 0
```

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())
```

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 haven't created 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()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: RobotInstances() takes exactly 1 argument (0 given)
```

The next idea, which still doesn't solve our problem, consists in 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()
0
>>> x = Robot()
>>> x.RobotInstances()
Traceback (most recent call last):

  File "<stdin>", line 1, in <module>
```

```
TypeError: RobotInstances() takes no arguments (1 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 consists 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 wanted:

```
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())
```

We will get the following 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())
```

The output looks like this:

```
$ python3 static_methods4.py
<class '__main__.Robot'>, 0)
<class '__main__.Robot'>, 1)
<class '__main__.Robot'>, 2)
<class '__main__.Robot'>, 2)
```

The use cases of class methods:

- the 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)
1/3
>>>
```

We will demonstrate in our last example the usefulness of class methods in inheritance. We define a class "Pets" with a method "about". This class will be inherited in a subclass "Dogs" and "Cats". The method "about" will be inherited as well. We will define the method "about" as a "staticmethod" in our first implementation to show the disadvantage of this approach:

```
class Pets:
    name = "pet animals"
```

```
    @staticmethod
    def about():
        print("This class is about {}!".format(Pets.name))


class Dogs(Pets):
    name = "'man's best friends' (Frederick II)"

class Cats(Pets):
    name = "cats"

p = Pets()
p.about()
d = Dogs()
d.about()
c = Cats()
c.about()
```

We get the following output:

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

Especially, in the case of c.about() and d.about(), we would have preferred a more specific phrase! The "problem" is that the method "about" doesn't know that it has been called by an instance of the Dogs or Cats class.
We decorate it now with a classmethod decorator instead of a staticmethod decorator:

```
class Pets:
    name = "pet animals"

    @classmethod
    def about(cls):
        print("This class is about {}!".format(cls.name))

class Dogs(Pets):
    name = "'man's best friends' (Frederick II)"

class Cats(Pets):
    name = "cats"

p = Pets()
p.about()
```

```
d = Dogs()
d.about()

c = Cats()
c.about()
```

The output is now like we wanted it to be:

```
This class is about pet animals!
This class is about 'man's best friends' (Frederick II)!
This class is about cats!
```