Table of contents (../toc.ipynb)

Classes

- A class is ideal if you want to combine code and data and classes give clean structure.
- Object orientated programming is very simple and powerful in Python.
- Basically, an object is something that has some properties and can do some things.
- Let's just start with an *mechanical object* that we code as Python class.

In [9]:
```python
class Engine:
    """A class that defines an engine with some methods."""
    conversion = 3.6

    def __init__(self, hp, consumption):
        self.power = hp
        self.cons = consumption

    def mps_in_kph(self, mps):
        """Here a method which converts meter per second
        in kilometer per hour."""
        return self.conversion * mps

    def get_power(self):
        """This method prints the power of the engine."""
        print(self.power)

    def consumption(self, distance):
        return self.cons * distance
```

Class syntax

Some explanation of the code you have seen:

- The class is defined with the `class` keyword.
- The variable `conversion` is a class attribute and shared everywhere in the class.
- The `def __init__(self, args)` line of code is a special method, which is executed once the object is initialized.
- The methods of the class are defined like functions with `def`.
- The `self.xxx` syntax is a reference to variables or methods of the class.

Class attributes

- Most attributes of classes are public (read- and writeable).
- However, you might want to protect some attributes sometimes. This can be done with:
    - Protected attributes. Here, the variable name is prefixed with one underscore `_varname`. You can actually change this attribute, but the developer gives you the hint to ignore it.
    - Private attributes, which are prefixed with two underscores `__varname`. These attributes are not visible from outside.

Now let's play with the class.

First, throw two objects from it.

In [10]:
```
small_eng = Engine(hp=80, consumption=5.9)

large_eng = Engine(hp=160, consumption=9.2)
```

We can print the doc string of the object with

```
In [11]: small_eng.__doc__
```

```
Out[11]: 'A class that defines an engine with some methods.'
```

We can access attributes of the object either with

```
In [12]: getattr(small_eng, 'conversion')

Out[12]: 3.6
```

Now let's use one method of the class.

```
In [13]: small_eng.get_power()
```

```
80
```

Or another one with an argument.

```
In [14]: small_eng.consumption(distance=5.8)
```

```
Out[14]: 34.22
```

```
In [15]: large_eng.consumption(distance=5.8)
```

```
Out[15]: 53.35999999999999
```

We can change a property of one of the engines easily.

```
In [16]: large_eng.cons = 11.9
```

And this changes of course it's behavior.

```
In [17]: large_eng.consumption(distance=5.8)

Out[17]: 69.02
```

Inheritance

Class inheritance is useful if you want to define classes as children of parent classes.

Let us take a look at one example.

In [18]:
```
class Parent:
    income = "large"


class Child(Parent):
    def education_level(self):
        if Parent.income == "large":
            print("High education level")
        else:
            print("Minor education level")
```

In [19]:
```
peter = Child()
peter.education_level()
```

```
High education level
```

- Here, the `Child` class "knows" the attribute of the `Parent` class by the definition `class Child(Parent)`.
- This way of organizing classes allow to structure your code very well.

- These were just some basics in Python classes.
- Please consult the [Python class documentation (https://docs.python.org/3/tutorial/classes.html)](https://docs.python.org/3/tutorial/classes.html) for more details.