

Objektorientierte Programmierung

Ein Computerprogramm besteht aus **Funktionen** (Programmabschnitten, die etwas tun) und **Variablen** (Speicherplätzen für Informationen). Werden Funktionen aktiviert, verändern sie diese Variablen.

Manchmal gehören Variable und Werte 'zusammen' und bilden eine logische Einheit. Ein Buntstift etwa hat eine bestimmte Farbe, eine bestimmte Länge und kann zum Schreiben und Zeichnen verwendet werden.

Das **Objekt Buntstift** hat

- Eigenschaften (*Farbe, Länge*) – Variable
- Fähigkeiten (*schreiben und zeichnen*) – Funktionen

Versuchen wir, eine Lampe zu beschreiben:

Das **Objekt Lampe** hat

- eine Eigenschaft: nämlich den Zustand, ob sie *brennt* oder nicht
- Fähigkeiten: man kann sie *einschalten* und *ausschalten*

wie könnte man das umsetzen:

Objekt Lampe:

```
def einschalten():
    brennt = 1
def ausschalten():
    brennt = 0
```

Hier gibt's aber ein Problem. Wie wir wissen, sind Variable prinzipiell lokal und nur innerhalb der Funktion gültig. Somit ist das `brennt` in `einschalten()` ein anderer Speicherplatz als das `brennt` in `ausschalten()`. Wir müssen den Funktionen sagen, dass es eine spezielle Variable `brennt` überall im Objekte Lampe gibt, also ihr 'eigenes' `brennt`. Um das zu ermöglichen, wird einer Funktion in einem Objekt immer automatisch der Name des Objektes selbst (englisch: `self`) übergeben. Und diesen dürfen wir verwenden. Jetzt klappt es so:

Objekt Lampe:

```
def einschalten(self):
    self.brennt = 1
def ausschalten(self):
    self.brennt = 0
```

So ein Aufwand für eine einzelne Lampe sieht etwas übertrieben aus. Tatsächlich können Objekte in guten Programmiersprachen mehr. Man programmiert sich nicht das Objekt alleine, sondern allgemeiner ein Muster (eine **Klasse**) dieses Objektes. Aus so einer Vorlage kann dann das **Objekt** konkret **erzeugt** werden – und nicht nur das, man kann dann beliebig viele Objekte erzeugen. Wir folgen dabei einer Python-Übereinkunft: Klassennamen schreiben wir mit großem Anfangsbuchstaben

```
class Lampe:
    def ein(self):
        self.brennt = 1
    def aus(self):
        self.brennt = 0
```

Und wie bedient man nun eine solche Lampe?

| | |
|--------------------|--------------------------------|
| lichterl = Lampe() | Wir erzeugen ein Lampen-Objekt |
| lichterl.ein() | einschalten |
| lichterl.aus() | ausschalten |

Die Schreibweise `lichterl.ein()` mit dem Punkt zwischen Objektname und Funktion ist typisch in der objektorientierten Programmierung und leichter lesbar als `'ein(lichterl)'`.

Noch ein Fachausdruck: unser `lichterl` ist ein Objekt, das nach dem Muster der Klasse `Lampe` erzeugt wird. Anders herum: `lichterl` ist ein konkretes Auftreten eines Exemplars vom Typ `Lampe` bzw. ein konkretes Beispiel für einen Vertreter der Klasse `Lampe`. Und da auf Englisch 'zum Beispiel' 'for instance' heißt, sagt man ganz gelehrt:

Das **Objekt** `lichterl` ist eine **Instanz** der **Klasse** `Lampe`. Die Variable `brennt` bezeichnet man als **Instanzvariable** oder **Attribut**, da sie eine Eigenschaft dieses Objektes ist. Und statt 'Funktion eines Objektes' spricht man gern von einer **Methode** des Objekts.

Jetzt wollen wir unsere Lampen aber verbessern. Es wäre doch schön, könnten wir feststellen, ob sie auf- oder abgedreht sind. Wir verpassen der Lampe eine Methode namens `zustand`.

```
class Lampe:
    def ein(self):
        self.brennt = 1
    def aus(self):
        self.brennt = 0
    def zustand(self):
        if self.brennt == 1:
            print 'Ich leuchte!'
        else:
            print 'Ich bin ausgeschaltet.'
```

Wir testen:

```
>>> funserl = Lampe()
>>> funserl.ein()
>>> funserl.zustand()
Ich leuchte!
>>> funserl.aus()
>>> funserl.zustand()
Ich bin ausgeschaltet.
```

Sieht gut aus! Ist es aber noch nicht....

```
>>> funserl = Lampe()
>>> funserl.zustand()
....
AttributeError: Lampe instance has no attribute 'brennt'
```

Beim Erzeugen des Objektes hat `brennt` noch keinen Wert. Sogar schlimmer – `brennt` existiert überhaupt noch nicht! Und das erklärt die Fehlermeldung.

Da es häufig so ist, dass Klassenvariable einen Anfangswert benötigen, gibt es dafür eine spezielle Methode in Python. Sie heißt `__init__`, wobei die doppelten Unterstreichungszeichen vorn und hinten auch eine Besonderheit dieser Funktion signalisieren sollen: sie wird nämlich vollautomatisch ausgeführt, wenn das Objekt erzeugt wird!

Wenn wir die Lampe anfangs ausgeschaltet sein lassen wollen, klappt es so:

```
class Lampe:
    def __init__(self):
        self.brennt = 0
    def ein(self):
        self.brennt = 1
    def aus(self):
        self.brennt = 0
    def zustand(self):
        if self.brennt == 1:
            print 'Ich leuchte!'
        else:
            print 'Ich bin ausgeschaltet.'
```

Und nun ein Experiment mit mehreren Lampen, die alle aus der selben Klasse erzeugt werden. Beachte, dass jede ihre eigene private Eigenschaft `brennt` besitzt!

```
>>> funserl = Lampe()
>>> leuchte = Lampe()
>>> funserl.ein()
>>> x = Lampe()
>>> x.ein()
>>> funserl.zustand()
Ich leuchte!
>>> leuchte.zustand()
Ich bin ausgeschaltet.
>>> x.aus()
>>> x.zustand()
Ich bin ausgeschaltet.
```

Randbemerkung:

In C++ und Java schreibt man üblicherweise **this** statt **self**. Wir könnten dies auch in Python so tun (oder ein ganz anderes Wort wählen), doch hat sich 'self' eingebürgert und alle schreiben das so. Wieso man das auch tun sollte? Um sich daran zu gewöhnen. Und wenn man einmal fremde Programme liest, weiß man sofort, dass 'self' die Referenz auf das Objekt selbst ist und muss nicht umdenken.

Das Überschreiben von Rechenoperationen wie `+` oder `/` ist ebenfalls möglich und wird als **überladen** bezeichnet. (In Python mit speziellen Methodennamen wie `__add__`, ...)

Vererbung

Nun ein umfangreicheres Beispiel. Wir wollen ein Sparbuch modellieren.

Welche Eigenschaften hat es: die Höhe des Guthabens

Was kann man mit ihm tun: einzahlen, abheben (aber nur, wenn der Betrag auch vorhanden ist) und den Guthabensstand abfragen.

```
class Sparbuch:
    def __init__(self):
        self.guthaben = 0
    def stand(self):
        print self.guthaben
    def einzahlen(self,betrag):
        self.guthaben += betrag
    def abheben(self,betrag):
        if betrag<=self.guthaben:
            guthaben -= betrag
        else:
            print 'So viel Geld hast Du nicht!'
```

```
>>> s = Sparbuch()
>>> s.einzahlen(50)
>>> s.stand()
50
>>> s.einzahlen(90)
>>> s.abheben(100)
>>> s.stand()
40
>>> s.abheben(60)
So viel Geld hast Du nicht!
```

Übung:

- wir wollen mitzählen, wie viele Einzahlungen und Abhebungen stattgefunden haben. Die Ausgabe des Guthabenstandes soll z.B. 'Guthaben 315 Euro nach 7 Einzahlungen und 3 Abhebungen' lauten.
- ein Mangel unserer Klasse: man kann die Einzahlung durch Angabe eines negativen Geldbetrages austricksen. Blockiere derartige Fehleingaben bei Einzahlung und Abheben.
- Die *print*-Ausgabe im Fehlerfall ist nicht sehr flexibel. Besser wäre es, den Fehler mittels *raise* als **Ausnahme** zu behandeln, die man mittels *try – except* vom aufrufenden Programm aus erkennen kann.

Ein Bankkonto

Eine besondere Art von Sparbuch ist ein Bankkonto. Es hat zwei Besonderheiten: man kann mehr abheben, als man besitzt (der Überziehungsrahmen), und man kann Geld auf ein anderes Konto überweisen. Die Attribute und Methoden des Sparbuchs sind aber immer noch sinnvoll. Deshalb programmieren wir das Konto als 'Erweiterung' des Sparbuches. Wir benutzen den Mechanismus der **Vererbung** um von der Grundklasse Sparbuch eine neue Klasse Konto **abzuleiten**. Dann erbt unsere neue Klasse alle Methoden und Attribute der alten Klasse (solange wir diese nicht umdefinieren)!

Die Methode abheben müssen wir allerdings verändern, um eine Kontoüberziehung zuzulassen. Wir **überschreiben** die Methode abheben der Klasse Sparbuch, das heißt wir ersetzen sie durch eine andere. (Sparbuch-Objekte sind von dieser Änderung natürlich nicht betroffen.)

| | |
|--|--|
| <pre>class Konto(Sparbuch): def __init__(self): Sparbuch.__init__(self) self.rahmen = 400 def abheben(self,betrag): if betrag<=self.guthaben+self.rahmen: self.guthaben -= betrag else: print 'So viel Geld hast Du nicht!' def ueberweisen(self,zielkonto,betrag): if betrag<=self.guthaben+self.rahmen: self.abheben(betrag) zielkonto.einzahlen(betrag) else: print 'So viel Geld hast Du nicht!'</pre> | <p>neue Klasse ableiten</p> <p>init der alten Klasse rufen (guthaben=0) Überziehungsrahmen festlegen</p> |
|--|--|

| | |
|--|--|
| <pre>>>> fritz = Konto() >>> franz = Konto() >>> fritz.einzahlen(500) >>> fritz.stand() 500 >>> franz.stand() 0 >>> fritz.ueberweisen(franz,300) >>> fritz.stand() 200 >>> franz.stand() 300 >>> franz.abheben(400) >>> franz.stand() -100 >>> fritz.abheben(900) So viel Geld hast Du nicht!</pre> | |
|--|--|