Klassen

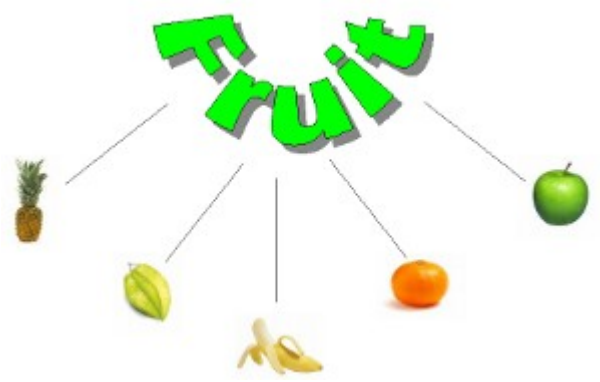
Vorbemerkung

Dieses Kapitel haben wir für Python3 komplett überarbeitet und erweitert. Wir empfehlen Ihnen dort die folgenden Kapitel durchzuarbeiten:

- Allgemeine Einführung in die Objektorientierte Programmierung (OOP)
- Klassen- und Instanzattribute
- Properties
- Vererbung
- Mehrfachvererbung
- Magische Methoden und Operator-Überladung

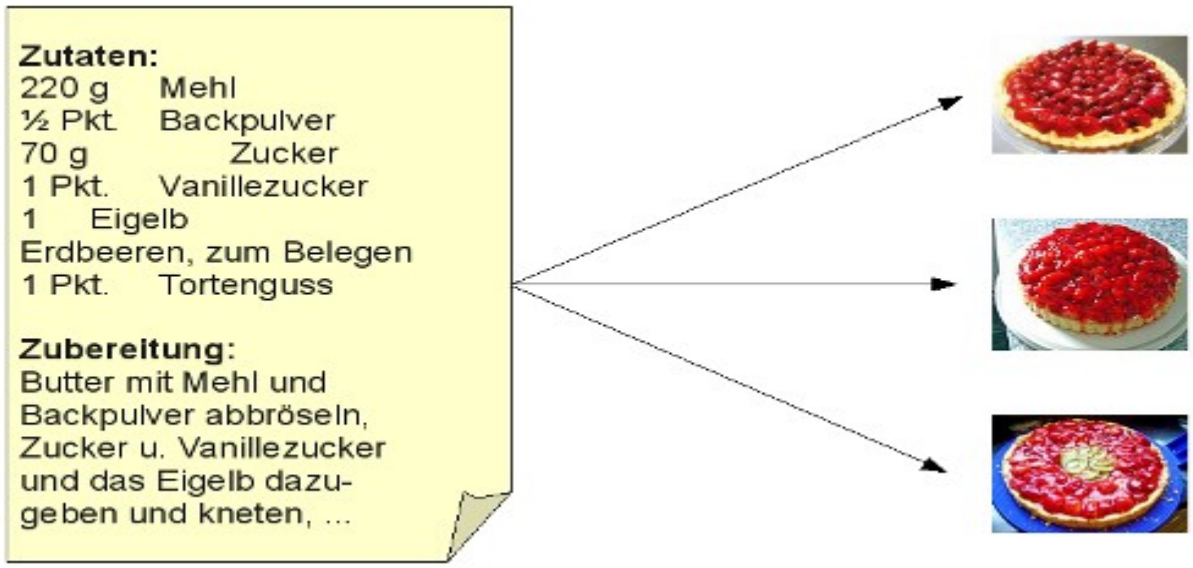
Da die Pflege und Erweiterung von vier verschiedene Python-Tutorials, - d.h. Python2 in Deutsch und Englisch und auch Python3 in beiden Sprachen, - einen enormen Arbeitsaufwand bedeutet, haben wir beschlossen in Zukunft uns hauptsächlich auf die deutschen und englischen Tutorials für Python3 zu konzentrieren. Wir hoffen auf Ihr Verständnis!

Objektorientierte Programmierung (OOP)



Die objektorientierte Programmierung (kurz: OOP) erfreut sich seit ihrer "Einführung" oder "Erfindung" mit "Simula 67" durch Ole-Johan Dahl und Kristen Nygard größter Beliebtheit. Dennoch ist sie nicht unumstritten. So bescheinigte beispielsweise der russische Informatiker Alexander Stepanow der OOP nur eine eingeschränkte mathematische Sichtweise und sagte, dass die OOP beinahe ein so großer Schwindel wie die künstliche Intelligenz sei.[1] Alexander Stepanow hat wesentlich an der Entwicklung der "C++ Standard Template Library" mitgewirkt, also sollte er bestens über OOP und ihrer Probleme in der Praxis Bescheid wissen.
Das Grundkonzept der objektorientierten Programmierung besteht darin, Daten und deren Funktionen (Methoden), - d.h. Funktionen, die auf diese Daten angewendet werden können - in einem Objekt zusammenzufassen und nach außen zu kapseln, so dass Methoden fremder Objekte diese Daten nicht direkt manipulieren können.
Objekte werden über Klassen definiert.
Eine Klasse ist eine formale Beschreibung, wie ein Objekt beschaffen ist, d.h. welche Attribute und welche Methoden sie hat.
Eine Klasse darf nicht mit einem Objekt verwechselt werden. Statt Objekt spricht man auch von einer Instanz einer Klasse.

Analogie: Kuchenklasse



Bei Einführungen in die objektorientierte Programmierung wird häufig und gerne auf Beispiele aus dem Alltag zurückgegriffen. Dabei handelt es sich meistens um Beispiele, die zwar helfen, objektorientierte Konzepte zu verdeutlichen, aber diese Beispiele lassen sich dann nicht in Programmcode wandeln. So auch in diesem Beispiel einer Kuchenklasse.
Betrachten wir das Rezept eines Erdbeerkuchens. Dann kann man dieses Rezept als eine Klasse betrachten. Das heißt, das Rezept bestimmt, wie eine Instanz der Klasse beschaffen sein muss. Backt jemand einen Kuchen nach dieser Klasse, dann schafft er eine Instanz oder ein Objekt dieser Klasse. Es gibt dann verschiedene Methoden, diesen Kuchen zu verarbeiten oder zu verändern. Ein nette Methode stellt übrigens in diesem Beispiel "aufessen" dar.
Ein Erdbeerkuchen gehört einer übergeordnete Klasse "Kuchen" an, die ihre Eigenschaften, z.B. dass ein Kuchen sich als Nachtisch nutzen lässt, an Unterklassen wie Erdbeerkuchen, Rührkuchen, Torten und so weiter vererbt.

Objekte

Der zentrale Begriff in der Objektorientierten Programmierung ist der des Objektes. Ein Objekt bezeichnet in der OOP die Abbildung eines realen Gegenstandes mit seinen Eigenschaften und Verhaltensweisen (Methoden) in ein Programm. Anders ausgedrückt: Ein Objekt kann immer durch zwei Dinge beschrieben werden:

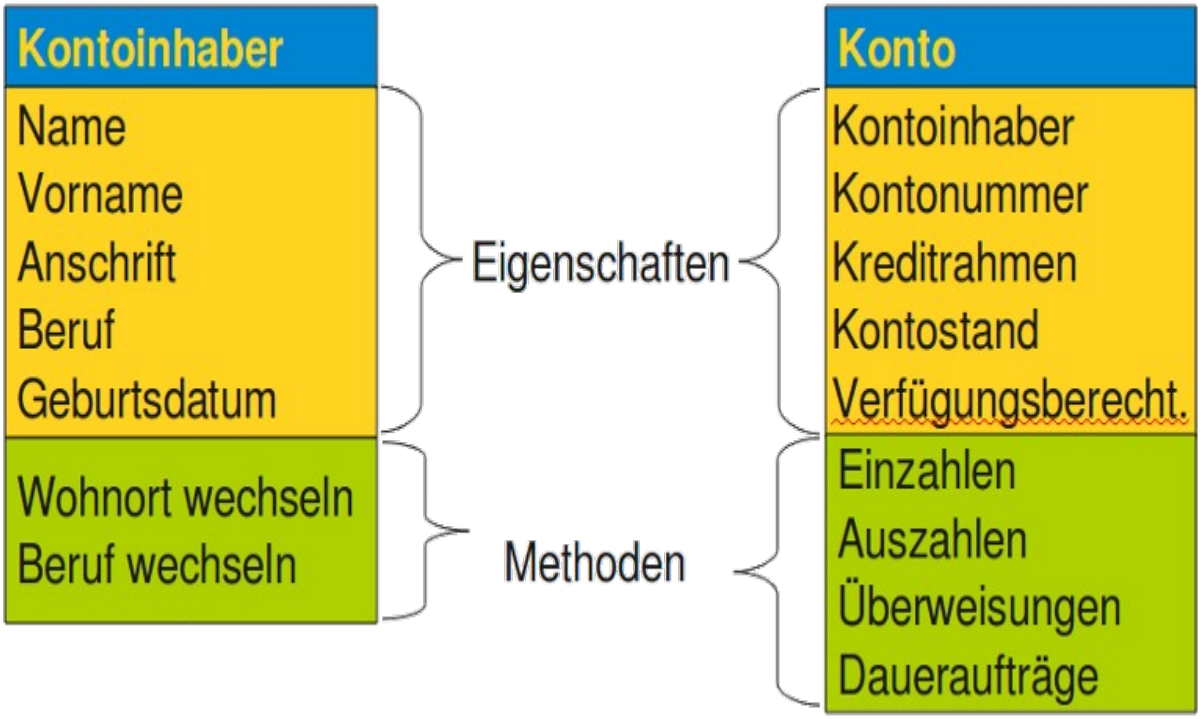
- was es tun kann oder was wir in einem Programm mit ihm tun können
- was wir über es wissen

Objekte sind Instanzen oder Exemplare einer Klasse. Die Begriffe Objekt und Instanz werden meist synonym gebraucht und bezeichnen den gleichen "Gegenstand". Objekte oder Instanzen werden mittels Konstruktoren erzeugt. Konstruktoren sind spezielle Methoden zur Erzeugung von Instanzen einer Klasse. Zum Entfernen oder Löschen von Instanzen gibt es die Destruktor-Methode.

Klasse

Eine Klasse ist ein abstrakter Oberbegriff für die Beschreibung der gemeinsamen Struktur und des gemeinsamen Verhaltens von realen Objekten (Klassifizierung).
Reale Objekte werden auf die für die Software wichtigen Merkmale abstrahiert.

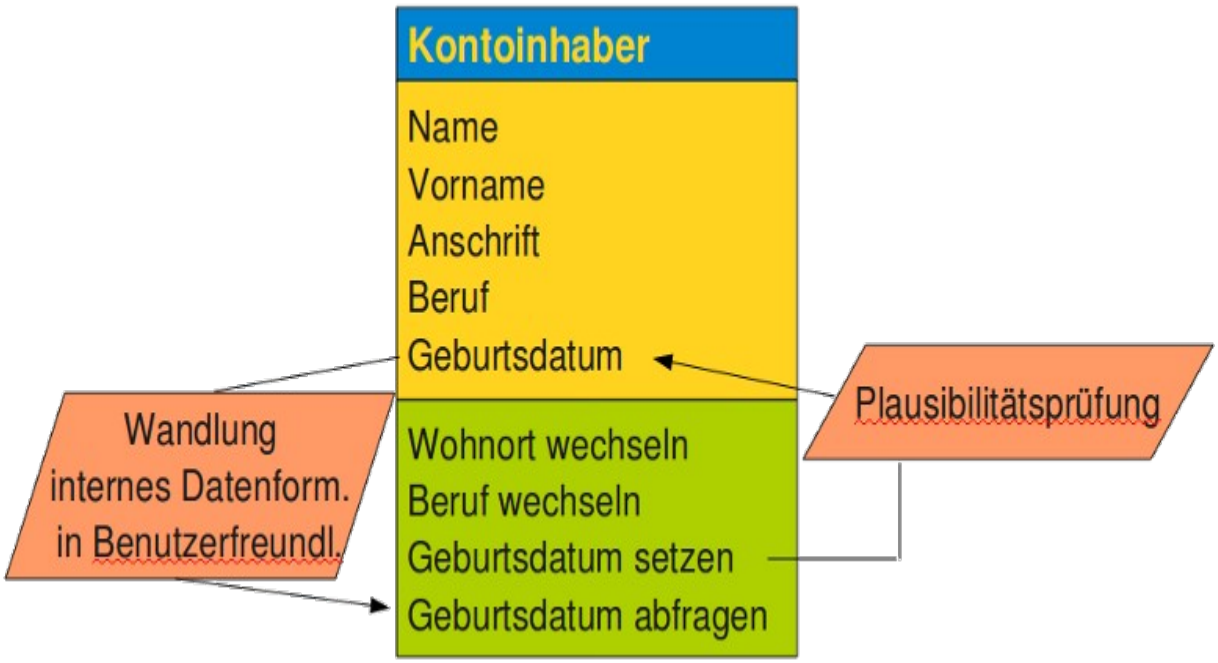
Die Klasse dient als Bauplan zur Abbildung von realen Objekten in Softwareobjekte, die sogenannten Instanzen. Die Klasse fasst hierfür notwendige Eigenschaften (Attribute) und zur Manipulation der Eigenschaften notwendige Methoden zusammen.
Klassen stehen häufig in Beziehung zueinander. Man hat zum Beispiel eine Oberklasse (Kuchen) und aus dieser leitet sich eine andere Klasse ab (Erdbeerkuchen). Diese abgeleitete Klasse erbt bestimmte Eigenschaften und Methoden der Oberklasse.
Methoden und Eigenschaften am Beispiel der Klassen "Kontoinhaber" und "Konto":



Kapselung von Daten

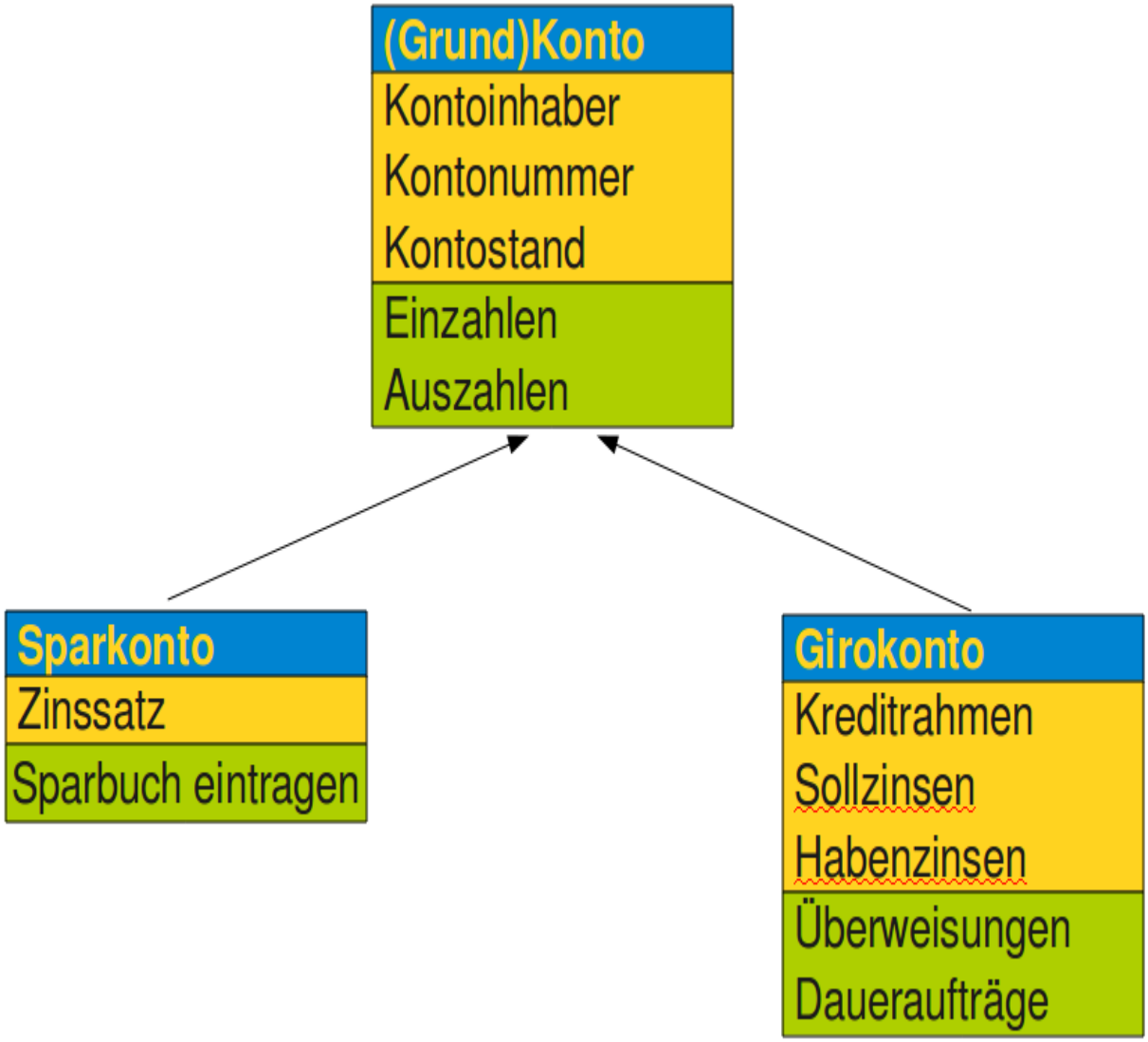
Ein weiterer wesentlicher Vorteil der OOP besteht in der Kapselung von Daten.
Zugriff auf Eigenschaften darf nur über Zugriffsmethoden erfolgen. Diese Methoden können Plausibilitätstests enthalten und sie (oder "nur" sie) besitzen "Informationen" über die eigentliche Implementierung.
So kann z.B. eine Methode zum Setzen des Geburtsdatums prüfen, ob das Datum korrekt ist und sich innerhalb eines bestimment Rahmen bewegt, z.B. Girokonto für Kinder unter 14 nicht möglich

oder Kunden über 100 Jahre unwahrscheinlich.



Vererbung

In unserem Beispiel erkennt man leicht, dass eine Klasse "Konto" einer realen Bank nicht genügen kann.
Es gibt verschiedene Arten von Konten: Girokonto, Sparkonto, usw.
Aber allen verschiedenen Konten sind bestimmte Eigenschaften und Methoden gemeinsam.
Beispielsweise wird jedes Konto eine Kontonummer, einen Kontoinhaber und einen Kontostand aufweisen. Gemeinsame Methoden: Einzahlen und Auszahlen
Es gibt also so etwas, wie ein Grundkonto, von dem alle anderen Konten "erben".



Die Vererbung dient also dazu, unter Zugrundelegung von existierenden Klassen neue zu schaffen. Eine neue Klasse kann dabei sowohl als eine Erweiterung als auch als eine Einschränkung der ursprünglichen Klasse entstehen.

Die einfachste Klasse

Die Definition einer neuen Klasse in Python wird mit dem Schlüsselwort class begonnen.

```
class Konto(object):
        pass
```

Die obige Klasse hat weder Attribute noch Methoden. Das "pass" ist übrigens eine Anweisung, die dem Interpreter sagt, dass man die eigentlichen Anweisungen erst später "nachliefert".
Ein Objekt oder eine Instanz der obigen (leeren) Klasse erzeugt man wie folgt:

```
>>> Konto()
<__main__.Konto object at 0x7f5feca55750>
```

Definition von Methoden

Eine Methode unterscheidet sich äußerlich nur in zwei Aspekten von von einer Funktion:

- Sie ist eine Funktion, die innerhalb einer class-Definition definiert ist.

- Der erste Parameter einer Methode ist immer eine Referenz self auf die Instanz, von der sie aufgerufen wird.

Der Parameter self erscheint nur bei der Definition einer Methode. Beim Aufruf wird er nicht angegeben.
Beispiel mit Methode:

```
class Konto(object):
    def ueberweisen(self, ziel, betrag):
        pass
    def einzahlen(self, betrag):
        pass
    def auszahlen(self, betrag):
        pass
    def kontostand(self):
        pass
```

Konstruktor

Genaugenommen gibt es in Python keine expliziten Konstruktoren oder Destruktoren. Häufig wird die __init__-Methode als Konstruktor bezeichnet. Wäre sie wirklich ein Konstruktor, würde sie wahrscheinlich __constr__ oder __constructor__ heißen. Sie heißt stattdessen __init__, weil mit dieser Methode ein Objekt, welcher vorher automatisch erzeugt ("konstruiert") worden ist, initialisiert wird. Diese Methode wird also unmittelbar nach der Konstruktion eines Objektes aufgerufen. Es wirkt also so, als würde das Objekt durch __init__ erzeugt. Dies erklärt den häufig gemachten Fehler in der Bezeichnungsweise.

Wir benutzen nun die __init__-Methode, um die Objekte unserer Kontoklasse zu initialisieren.
__init__
Konstruktoren werden wie andere Methoden definiert:

```
def __init__(self, inhaber, kontonummer,
                                        kontostand, kontokorrent=0):
        self.Inhaber = inhaber
        self.Kontonummer = kontonummer
        self.Kontostand = kontostand
        self.Kontokorrent = kontokorrent
```

Destruktor

Hier gilt analog das bereits unter dem Absatz Konstrukoren Gesagte: Es gibt keine expliziten Konstruktoren. Für eine Klasse kann man eine Methode __del__ definieren. Wenn man eine Instanz einer Klasse mit del löscht, wird die Methode __del__ aufgerufen. Allerdings nur, falls es keine weitere Referenz auf diese Instanz gibt. In C++ werden Destruktoren prinzipiell benötigt, da in ihnen die Speicherbereinigung vorgenommen wird. Da man sich in Python nicht um die Speicherbereinigung kümmern muss, wird die Methode __del__ relativ selten benutzt.
Im Folgenden sehen wir ein Beispiel mit __init__ ("Konstruktor") und __del__ (Destruktor):

```
class Greeting:
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print "Destruktor gestartet"
    def SayHello(self):
```

```
        print "Guten Tag", self.name
```

Diese Klasse wird nun interaktiv benutzt:

```
>>> execfile("hello_class.py")
Guten Tag Guido
>>> x1 = Greeting("Guido")
>>> x2 = x1
>>> del x1
>>> del x2
Destruktor gestartet
```

Man sollte aber vorsichtig mit der Benutzung der __del__-Methode sein. "del x" startet erst dann die __del--Methode, wenn es keine weiteren Referenzen auf das Objekt gibt, d.h. wenn der Refernzzähler auf 0 gesetzt wird. Probleme gibt es beispielsweise, wenn es zirkuläre Referenzen (Zirkelbezüge), wie beispielsweise bei doppelt verketteten Listen gibt.

Vollständiges Beispiel der Kontoklasse

```
class Konto(object):

    def __init__(self, inhaber, kontonummer,
                 kontostand,
                 kontokorrent=0):
        self.Inhaber = inhaber
        self.Kontonummer = kontonummer
        self.Kontostand = kontostand
        self.Kontokorrent = kontokorrent

    def ueberweisen(self, ziel, betrag):
        if(self.Kontostand - betrag < -self.Kontokorrent):
            # Deckung nicht genuegend
            return False
        else:
            self.Kontostand -= betrag
            ziel.Kontostand += betrag
            return True

    def einzahlen(self, betrag):
       self.Kontostand += betrag

    def auszahlen(self, betrag):
       self.Kontostand -= betrag

    def kontostand(self):
        return self.Kontostand
```

Hat man dieses Beispiel unter konto.py abgespeichert, kann man in einer python-Shell wie folgt damit arbeiten:

```
>>> from konto import Konto
>>> K1 = Konto("Jens",70711,2022.17)
>>> K2 = Konto("Uta",70813,879.09)
>>> K1.kontostand()
2022.17
>>> K1.ueberweisen(K2,998.32)
True
>>> K1.kontostand()
1023.85
```

```
>>> K2.kontostand()
1877.41
```

"Öffentlicher" Schönheitsfehler

Einen kleinen Schönheitsfehler hat die Klasse Konto() noch. Man kann von außen direkt auf die Attribute zugreifen:

```
>>> K2.Kontostand
1877.4100000000001
>>> K2.Kontonummer
70813
>>> K2.Kontokorrent
0
>>> K2.Kontostand = 1000000
>>> K2.Kontostand
1000000
```

Datenkapselung

Normalerweise sind alle Attribute einer Klasseninstanz öffentlich, d.h. von außen zugänglich. Python bietet einen Mechanismus um dies zu verhindern. Die Steuerung erfolgt nicht über irgendwelchen speziellen Schlüsselworte sondern über die Namen, d.h. einfacher dem eigentlichen Namen vorgestellter Unterstrich für den protected und zweifacher vorgestellter Unterstrich für private, wie man der folgenden Tabelle entnehmen kann:

Namen	Bezeichnung	Bedeutung
name	Public	Attribute ohne führende Unterstriche sind sowohl innerhalb einer Klasse als auch von außen les- und schreibbar.
_name	Protected	Man kann zwar auch von außen lesend und schreibend zugreifen, aber der Entwickler macht damit klar, dass man diese Member nicht benutzen sollte.
__name	Private	Sind von außen nicht sichtbar und nicht benutzbar.

Unsere Konto-Beispielklasse sieht mit private-Membern wie folgt aus:

```
class Konto(object):

    def __init__(self, inhaber, kontonummer,
                 kontostand,
                 kontokorrent=0):
        self.__Inhaber = inhaber
        self.__Kontonummer = kontonummer
        self.__Kontostand = kontostand
        self.__Kontokorrent = kontokorrent

    def ueberweisen(self, ziel, betrag):
        if(self.__Kontostand - betrag < -self.__Kontokorrent):
            # Deckung nicht genuegend
            return False
        else:
            self.__Kontostand -= betrag
            ziel.__Kontostand += betrag
            return True

    def einzahlen(self, betrag):
       self.__Kontostand += betrag

    def auszahlen(self, betrag):
       self.__Kontostand -= betrag
```

```
    def kontostand(self):
        return self.__Kontostand
```

Statische Member

Bisher hatte jedes Objekt einer Klasse seine eigenen Attribute und Methoden, die sich von denen anderer Objekte unterschieden.
Man bezeichnet dies als "nicht-statisch" oder dynamisch, da sie für jedes Objekt einer Klasse dynamisch erstellt werden.
Wie kann man aber z.B. die Anzahl der verschiedenen Instanzen/Objekte einer Klasse zählen? In unserer Konto()-Klasse also die Anzahl der verschiedenen Konten.
Statische Attribute werden außerhalb des Konstruktors direkt im class-Block definiert. Es ist Usus die statischen Member direkt unterhalb der class-Anweisung zu positionieren.

In unserem Beispiel der Kontenklasse lässt sich zum Beispiel die Anzahl der Konten innerhalb des Programms nur statisch zählen:

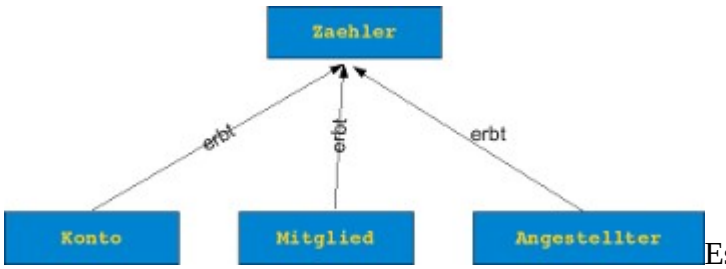
```
class Konto(object):
    objekt_zaehler = 0

    def __init__(self, inhaber, kontonummer,
                 kontostand,  kontokorrent=0):
        self.__Inhaber = inhaber
        self.__Kontonummer = kontonummer
        self.__Kontostand = kontostand
        self.__Kontokorrent = kontokorrent
        Konto.objekt_zaehler += 1

    def __del__(self):
        Konto.objekt_zaehler -= 1
```

In der folgenden interaktiven Sitzung können wir verfolgen, wie dieses Zählen vor sich geht:

```
>>> execfile("konto.py")
>>> k1 = Konto("Homer Simpson", 2893002, 2325.21)
>>> Konto.objekt_zaehler
1
>>> k2 = Konto("Fred Flintstone", 2894117, 755.32)
>>> k3 = Konto("Bill Gates", 2895007, 5234.32)
>>> Konto.objekt_zaehler
3
>>> k2.objekt_zaehler
3
>>> del k3
>>> Konto.objekt_zaehler
2
```

Vererbung

So wie man in der Klasse Konto() die Instanzen zählt, so könnte es auch in anderen Klassen notwendig oder sinnvoll sein. Man möchte aber nicht in jeder Klasse den Code fürs Hochzählen in den Konstruktor und den fürs Herunterzählen in den Destruktor übernehmen.



Es gibt die Möglichkeit die Fähigkeit des Instanzen-Zählens an andere Klassen zu vererben.
Dazu definiert man eine "Ober"klasse Zähler, die ihre Eigenschaften an andere, wie z.B. Konto überträgt. Im nebenstehenden Diagram gehen wir beispielsweise davon aus, dass die Klassen "Konto", "Mitglied" und Angestellter" die Basisklasse "Zähler" benötigen.
Im Folgenden zeigen wir den vollständigen Code einer solchen Zaehler-Klasse:

```
class Zaehler(object):
    Anzahl = 0

    def __init__(self):
        type(self).Anzahl += 1

    def __del__(self):
        type(self).Anzahl -= 1

class Konto(Zaehler):
    def __init__(self, inhaber, kontonummer,
                 kontostand,
                 kontokorrent=0):
        Zaehler.__init__(self)
```

Mehrfachvererbung

Eine Klasse kann auch Subklasse von mehreren Basisklassen sein, d.h. sie kann von mehreren Klassen erben. Das Erben von mehreren bezeichnet man als **Mehrfachvererbung**
Syntaktisch gesehen ist dies denkbar einfach: Statt nur eine Klasse innerhalb der KLammer hinter dem Klassennamen gibt man eine durch Komma getrennte Liste aller Basisklassen an, von denen geerbt werden soll.

```
class NN (Klasse1, Klasse2, Klasse3, ...):
```

Die obige Klasse NN erbt also von den Klassen "Klasse1", "KLasse2", "Klasse3" und so weiter.

---

[1] in einem Interview mit Graziano Lo Russo

Voriges Kapitel: Ausnahme-Behandlung
Nächstes Kapitel: Slots