Teil I.

OOP mit Python - erster Teil vgl. anderes PDF

Teil II.

OOP mit Python 2.7 - weitere Konzepte

1. Vererbung in Python

Eine Vererbung in Python wird grundsätzlich dadurch realisiert, indem nach dem *import* die Elternklasse (Oberklasse) in Klammern hinter die Kindklasse (Unterklasse) geschrieben wird. Bei einer Mehrfachvererbung werden die Elternklassen durch Komma getrennt hintereinander in die Klammer geschrieben.

HINWEIS zum import:[1]

from *modul* import *name*

Bei diesem Aufruf wird aus dem Modul mit dem Namen *modul* die Klasse, Funktion oder Variable *name* importiert. Danach ist es als *name* zugreifbar. Auch komplexere Aufrufe mit umbenennen sind möglich, wie

from *math* import *sqrt* as *quadratwurzel*

Dadurch ist die Funktion *sqrt()* aus dem math-Modul nun als *quadratwurzel()* bekannt. Dabei muss *name* auch nicht unbedingt eine Funktion sein, es kann auch eine Variable, eine Klasse oder auch eine Instanz sein.

from *modul* import *

Die letzte Option ist ein Stern-Import. Der Stern dient als Platzhalter und signalisiert, dass "alles" aus dem betreffenden Modul importiert werden soll. Dies mag zwar bequem erscheinen, so dass man nicht mehr den Modulnamen vor der Funktion schreiben muss, hat aber den Nachteil, dass bestehende Objekte (Funktionen, Klassen etc.) unbemerkt überschrieben werden können!

Es passiert oft, dass dann bestehende Importe durch nachfolgende Importe bei gleichen Namen überschreiben werden. Das führt sehr oft zu schwer zu verstehenden Fehlern. Eine saubere Lösung ist es, Module generell nie mit Stern einzubinden, außer man weiß genau was man tut.

HINWEIS: Es sollte immer nur das importiert werden, was gebraucht wird, möglichst keinen Stern * verwenden!

```
from Mitarbeiter import Mitarbeiter
class Angestellter(Mitarbeiter):
```

[1] vgl. http://wiki.python-forum.de/Import

Angestellter2.py

```
1 from Mitarbeiter import Mitarbeiter
2 class Angestellter2(Mitarbeiter):
3    'Angestellter2 erbt von Mitarbeiter'
4
5    def zeigeAngestellter(self):
6       print "AngestellterAusgabe:   Name : ", self.name,  ", Gehalt: ", self.gehalt
7
8 # ENDE der Klassendefinition
9
```

Dann können alle Attribute und Methoden von der Elternklasse (hier Mitarbeiter) genutzt werden:

Mitarbeiter.py

```
1  class Mitarbeiter:
2     'Optionaler Klassen-Dokumentations-Text, z.B. Gemeinsame Basis-Klasse für alle Angestellten'
3     mitarbAnzahl = 0
4     def __init__(self, name, gehalt):
5        self.name = name
6        self.gehalt = gehalt
7        Mitarbeiter.mitarbAnzahl += 1
8
9     def zeigeAnzahl(self):
10      print "Gesamt Mitarbeiter %d" % Mitarbeiter.mitarbAnzahl
11
12    def zeigeMitarbeiter(self):
13       print "Name : ", self.name,  ", Gehalt: ", self.gehalt
14
15    def setGehalt(self,gehalt):  # beachte die Angabe von self in der Parameterliste!
16       self.gehalt = gehalt
17
18 # ENDE der Klassendefinition
19
```

Die Methode *zeigeAngestellter()* gehört nur zur Klasse Angestellter2. Dies ist ein Beispiel wie eine Klasse trotz Vererbung eine eigene Ausgabe bekommen kann:

Main3.py

```
1  from Angestellter2 import Angestellter2
2  # ANFANG Hauptprogramm (main)
3
4  # Hier wird ein erstes Objekt der Angestellter-Klasse erschaffen
5  angest1 = Angestellter2("Zara", 2000)
6  # Hier wird ein zweites Objekt der Angestellter-Klasse erschaffen
7  angest2 = Angestellter2("Manni", 5000)
8
9  angest1.zeigeAngestellter()    # beachte: kein self in der Klammer beim Aufruf!
10 angest2.zeigeAngestellter()
11
12 angest2.setGehalt(4430)
13 angest2.zeigeAngestellter()
14 angest2.zeigeMitarbeiter()
15
16 print "Gesamt Mitarbeiter %d" % Mitarbeiter.mitarbAnzahl
```

```
AngestellterAusgabe:   Name :  Zara , Gehalt:  2000
AngestellterAusgabe:   Name :  Manni , Gehalt:  5000
AngestellterAusgabe:   Name :  Manni , Gehalt:  4430
Name :  Manni , Gehalt:  4430
Gesamt Mitarbeiter 2
```

ACHTUNG!

Bei der Ausführung der o.a. main3.py gibt es einen Fehler in Zeile 16:

```
12 angest2.setGehalt(4430)
13 angest2.zeigeAngestellter()
14 angest2.zeigeMitarbeiter()
15
16 print "Gesamt Mitarbeiter %d" % Mitarbeiter.mitarbAnzahl
```

```
Der Name 'Mitarbeiter' ist nicht definiert oder falsch geschrieben.
Es existiert also keine Variable oder Funktion mit diesem Namen.
-----
Traceback (most recent call last):
  File                                                    'Main3.py", line 16, in <module>
    print "Gesamt Mitarbeiter %d" % Mitarbeiter.mitarbAnzahl
NameError: name 'Mitarbeiter' is not defined
```

Was ist passiert? Haben wir nicht alles richtig gemacht? Die Klasse Mitarbeiter sollte doch in Zeile 1 importiert worden sein, da Angestellter2 von Mitarbeiter erbt?[2]

1.1. Aufruf des übergeordneten Konstruktors

Normalerweise wird bei der Instanziierung, also dem Aufruf des Konstruktors, automatisch der Konstruktor der Elternklasse benutzt. Das geschieht „automatisch“, ohne dass dieser Konstruktor in der Angestellten-Klasse nochmals definiert bzw., hingeschrieben werden muss. Allerdings

[2] Hier lauert ein Fallstrick bei der Benutzung der import-Funktion. Zwar sollte man auf das * beim Import verzichten und so genau wie möglich die zu importierenden Module bzw. Klassen angeben. Allerdings „kennt“ hier main3.py die Klasse Mitarbeiter in diesem Falle nicht, da tatsächlich *nur* die Klasse Angestellter2 bekannt gemacht wird! Wenn wir also eine Ebene tiefer importieren wollen, müssen wir die Klasse explizit angeben, indem wir die Zeile 1 um die Klasse Mitarbeiter ergänzen:
from Angestellter2 import Angestellter2, Mitarbeiter
Folgerung: Beim Import von vererbten Klassen ist auf den Import der Elternklasse zu achten. Oder doch, als schlechtere Lösung, sicherheitshalber das * verwenden.

gibt es genügend Fälle, in denen die Angestellten-Klasse eigene, zusätzliche, Attribute zur Elternklasse bekommt. Dann ist folgendes Vorgehen in Python notwendig (vgl. *Angestellter5.py*):

```
1  from Mitarbeiter import Mitarbeiter
2  class Angestellter5(Mitarbeiter):
3     'Angestellter5 erbt von Mitarbeiter'
4     def __init__(self, name, gehalt, projektnummer):
5      Mitarbeiter.__init__(self, name, gehalt) # Aufruf des übergeordneten Konstruktors der Elternklasse mit den aktuellen Parametern
6      # alternative Aufruf des übergeordneten Konstruktors ohne Namensangabe aber mit dem Schlüsselwort super
7      # super().__init__(name, gehalt)
8      self.projNummer = projektnummer # neues Attribut nur für die Angestellten-Klasse
9
10    def zeigeMitarbeiter(self):
11       print
12       print "Angestellter-Ausgabe:"
13       print "Name : ", self.name,  ", Gehalt: ", self.gehalt
14       print "Projekt:", self.projNummer
15
16 # ENDE der Klassendefinition
17
```

hier die zugehörige *Main4b.py* mit dem Aufruf des erweiterten Konstruktors und der Ausgabe:

```
1  from Angestellter5 import   *
2  # ANFANG Hauptprogramm (main)
3
4  # Hier wird ein erstes Objekt der Angestellter-Klasse erschaffen
5  angest1 = Angestellter5("Zara", 2000,12)
6  # Hier wird ein zweites Objekt der Angestellter-Klasse erschaffen
7  angest2 = Angestellter5("Manni", 5000, 54)
8
9  angest1.zeigeMitarbeiter()    # beachte: kein self in der Klammer beim Aufruf!
10 angest2.zeigeMitarbeiter()
11
12 angest2.setGehalt(4430)
13 angest2.zeigeMitarbeiter()
14
15 print "Gesamt Mitarbeiter %d" % Mitarbeiter.mitarbAnzahl
```

```
Angestellter-Ausgabe:
Name :  Zara , Gehalt:  2000
Projekt: 12

Angestellter-Ausgabe:
Name :  Manni , Gehalt:  5000
Projekt: 54

Angestellter-Ausgabe:
Name :  Manni , Gehalt:  4430
Projekt: 54
Gesamt Mitarbeiter 2
```

Dabei gilt grundsätzlich folgende Regel: Neue (eigene) Attribute werden immer im Konstruktor-Bereich geschrieben (definiert). *Sie müssen aber nicht zwangsläufig beim Konstruktor-Aufruf als Parameter erscheinen!*

Aber: wird der Konstruktor in der Kindklasse geschrieben, so muss ein Aufruf des Konstruktors der Elternklasse erfolgen, um die grundlegenden Parameter des Elternkonstruktors durchzureichen. Dies kann wie oben gezeigt entweder über den Namen der Elternklasse oder mit Hilfe der Methode *super()* erfolgen.

1.2. Überladen von Methoden

Bei der Vererbung können Methoden, die schon in der Elternklasse vorhanden sind, in der Kindklasse *überschrieben* werden. Diesen Vorgang nennt man *Überladen von Methoden*. Dazu wird in der Unterklasse (Kindklasse) eine Methode mit gleichem Namen wie in der Oberklasse (Elternklasse) definiert, allerdings mit anderer Implementierung.

Angenommen wir wollen die Methode *zeigeAnzahl*, die schon in der Klasse Mitarbeiter (vgl. unten) definiert ist, für die Klasse Angestellter „neu“ implementieren. Dann könnte dies beispielsweise so aussehen:

```
from Mitarbeiter import *
class Angestellter5(Mitarbeiter):
   'Angestellter5 erbt von Mitarbeiter'
   angestAnzahl = 0
   def __init__(self, name, gehalt, projektnummer):
    Mitarbeiter.__init__(self, name, gehalt) # Aufruf des übergeordneten Konstruktors der Elternklasse mit den aktuellen Parametern
    # alternative Aufruf des übergeordneten Konstruktors ohne Namensangabe aber mit dem Schlüsselwort super
    # super().__init__(name, gehalt)
    self.projNummer = projektnummer # neues Attribut nur für die Angestellten-Klasse
    Angestellter5.angestAnzahl += 1

   def zeigeAnzahl():
     pass

   def zeigeMitarbeiter(self):
      print
      print "Angestellter-Ausgabe:"
      print "Name : ", self.name,  ", Gehalt: ", self.gehalt
      print "Projekt:", self.projNummer

# ENDE der Klassendefinition

```

In der Klasse Mitarbeiter ist die übergeordnete Methode etwas anders definiert:

```
class Mitarbeiter:
   'Optionaler Klassen-Dokumentations-Text, z.B. Gemeinsame Basis-Klasse für alle Angestellten'
   mitarbAnzahl = 0
   def __init__(self, name, gehalt):
      self.name = name
      self.gehalt = gehalt
      Mitarbeiter.mitarbAnzahl += 1

   def zeigeAnzahl(self):
     print "Gesamt Mitarbeiter %d" % Mitarbeiter.mitarbAnzahl

   def zeigeMitarbeiter(self):
      print "Name : ", self.name,  ", Gehalt: ", self.gehalt

   def setGehalt(self,gehalt):  # beachte die Angabe von self in der Parameterliste!
      self.gehalt = gehalt
```

Beim Ausführen der Methoden wird die zum zugehörigen Objekt gehörige Methode aufgerufen. Diese gibt eine der unterschiedlichen Implementierung entsprechende Ausgabe in Main5b.py:

```
1  from Angestellter5b import *
2  # ANFANG Hauptprogramm (main)
3
4  # Hier wird ein erstes Objekt der Angestellter-Klasse erschaffen
5  angest1 = Angestellter5b("Zara", 2000,12)
6  # Hier wird ein zweites Objekt der Angestellter-Klasse erschaffen
7  angest2 = Angestellter5b("Manni", 5000, 54)
8
9  angest1.zeigeMitarbeiter()    # beachte: kein self in der Klammer beim Aufruf!
10 angest2.zeigeMitarbeiter()
11
12 angest2.setGehalt(4430)
13 angest2.zeigeMitarbeiter()
14 angest2.zeigeAnzahl()
15
16 mitarb = Mitarbeiter("Herr Mitarbeit",10000)
17 mitarb.zeigeAnzahl()
18
```

```
Angestellter-Ausgabe:
Name :  Zara , Gehalt:  2000
Projekt: 12

Angestellter-Ausgabe:
Name :  Manni , Gehalt:  5000
Projekt: 54

Angestellter-Ausgabe:
Name :  Manni , Gehalt:  4430
Projekt: 54
Anzahl Angestellter:  2
Gesamt Mitarbeiter 3
```

2. Assoziationen in Python

Um Beziehungen von Klassen untereinander zu realisieren benutzt man in der OOM, insbesondere in der UML-Darstellung, die Assoziation (vgl. dort). Eine Assoziation ist eine „hat“-Beziehung von zwei Klassen untereinander. Man sagt auch, die assoziierten Klassen kennen sich. Beispielsweise kann eine Klasse *Abteilungsleiter*, die ebenfalls von Mitarbeiter erbt, in Beziehung zu einer einer Klasse *Buero* stehen. Klassen müssen untereinander nicht zwangsweise in Beziehung stehen. Sollte aber in der Realität eine Beziehung wie „Ein Abteilungsleiter hat ein Büro“ oder „Zu einem Abteilungsleiter gehört ein Büro“ oder „Ein Büro wird von einem Abteilungsleiter besetzt“ im OOM modelliert werden, so wird mit einer Assoziation (Beziehung) gearbeitet.

Auf der OOP-Ebene wird diese „hat“ Beziehung (Assoziation) folgendermassen ausgedrückt (Beispiel5, hier Main5.py abgebildet):

```
from Abteilungsleiter5 import *
from Buero5 import *
# ANFANG Hauptprogramm (main)

# Zuerst brauchen wir ein Büro, da wir dies später verwenden wollen
buero1 = Buero5("0055302",13) # Raumnummer 0055302 mit 13qm Größe

# Hier wird ein erstes Objekt der Abteilungsleiter-Klasse erschaffen
leiter1 = Abteilungsleiter5("Sarah", 12000, buero1)
# beide sitzen im gleichen Büro
leiter2 = Abteilungsleiter5("Manfred", 15000, buero1)

leiter1.zeigeAbteilungsleiter()    # beachte: kein self in der Klammer beim Aufruf!
leiter2.zeigeAbteilungsleiter()

leiter2.umsatz = 98000
leiter2.setzeProvisionanteil(1.6)
leiter2.zeigeProvisionbetrag()

leiter2.setGehalt(14430)
leiter2.zeigeMitarbeiter()

print "Gesamt Mitarbeiter %d" % Mitarbeiter.mitarbAnzahl
```

```
Abteilungsleiter-Ausgabe
Name :  Sarah , Gehalt:  12000
Buero:  Bueronummer 0055302  mit Groesse 13 qm
None
Abteilungsleiter-Ausgabe
Name :  Manfred , Gehalt:  15000
Buero:  Bueronummer 0055302  mit Groesse 13 qm
None
Name :  Manfred , Gehalt:  14430
Gesamt Mitarbeiter 2
```

Zuerst wird ein Objekt aus der Buero5-Klasse erzeugt. Dann wird dieses erzeugte Objekt bei der Erzeugung einer Instanz des Abteilungsleiter im Konstruktor mit aufgeführt („hat ein Büro“).

Beide Abteilungsleiter haben das gleiche Büro. Die Abteilungsleiter-Ausgabe wurde erweitert (vgl. Ausgabe in der Abbildung oben).

In der Klasse Abteilungsleiter5 wurde das Büro mit in den eigenen Konstruktor aufgenommen und alle weiteren Parameter werden an die Oberklasse durchgereicht. Das eigene Attribut *buero* wird mit der übergebenen Büro-Referenz belegt. Die zwei Methoden *setzeProvisionanteil* und *zeigeProvisionbetrag* wurden zwar definiert aber nicht ausprogrammiert. Es gibt Fälle, wo eine Methode zwar definiert ist, aber (erstmal) nichts bewirken soll. Dies erreicht man entweder mit einem sofortigen *return* (bei einer Methode mit Rückgabewert) oder mit der leeren Anweisung *pass*. Deshalb auch die Ausgabe „None“ im Ausgabebereich.

Die Assoziation zwischen Abteilungsleiter und Büro erfolgt über die Aufnahme des Büros in den Konstruktor der Klasse Abteilungsleiter5. Diese Form bedeutet, dass ein Abteilungsleiter zwingend ein Büro übergeben bekommen muss, sonst kann keine Objekt aus der Klasse Abteilungsleiter5 erzeugt werden. Diese Art von Assoziation (Beziehung) wird Komposition genannt: Die Existenz eines Objekts, das Teil eines Ganzen ist, ist von der Existenz des Ganzen abhängig.

```
from Mitarbeiter import *

class Abteilungsleiter5 (Mitarbeiter):
    'Abteilungsleiter erbt von Mitarbeiter und hat ein Büro'
    def __init__(self, name, gehalt, bueroraum):
     Mitarbeiter.__init__(self, name, gehalt) # Aufruf des übergeordneten Konstruktors der Elternklasse mit den aktuellen Parametern
     self.buero = bueroraum
     self.umsatz = 0

    def zeigeAbteilungsleiter(self):
        print "Abteilungsleiter-Ausgabe"
        print "Name : ", self.name,  ", Gehalt: ", self.gehalt
        print "Buero: ", self.buero.ausgabe()

    def setzeProvisionanteil(self, provision):
     pass

    def zeigeProvisionbetrag(self):
     return


# ENDE der Klassendefinition

```

Aufgabe

Der Provisionsbetrag ist der prozentuale Provisionsanteil vom Umsatz.

> Beispiel: beträgt der Provisionsanteil 1,6 (Prozent), dann ergibt sich bei einem Umsatz von 98.000 (Euro) ein Provisionsbetrag von 1.568 (Euro).

Ergänzen Sie die Klasse Abteilungsleiter5 um die notwendigen Attribute und programmieren Sie die beiden schon definierten Methoden in ihrer Funktionalität. Falls notwendig, ergänzen Sie den Konstruktor.

2.1. Kategorien von Assoziationen

Weitere Beispiel für Assoziationen zwischen Klassen:[3]

Assoziationskategorie	Beispiele
A ist physische Komponente von B	Beamer (A) - Raum (B)
A ist logische Komponente von B	Teilnehmer – Studiengang
A ist ein Mitglied von B	Mitarbeiter – Unternehmen
A ist eine organisatorische Einheit von B	Abteilung – Unternehmen
A benutzt/verwaltet B	Organisationsassistent – Anmeldung
A steht in Beziehung zu einem Vorgang	Teilnehmer – Anmeldung
A besitzt B	Teilnehmer – Schulabschuss

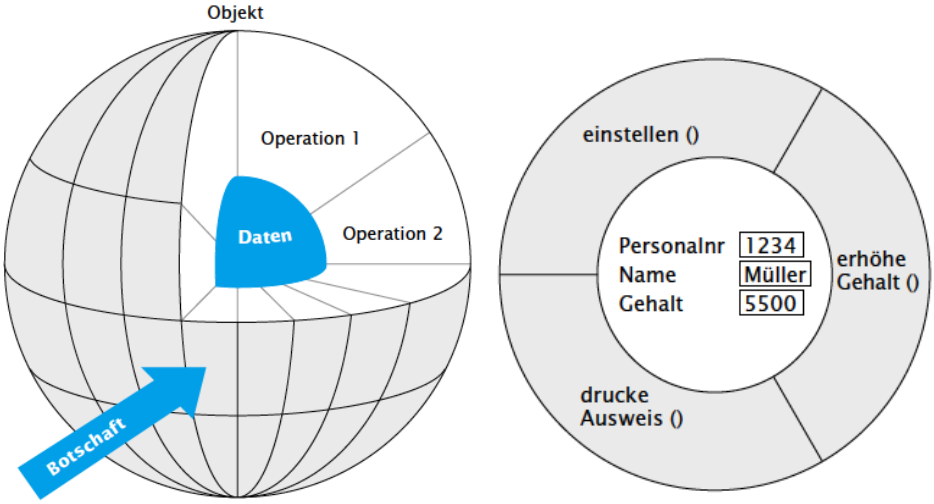
3. Sichtbarkeit von Attributen und Methoden (Geheimnisprinzip und Kapselung von Objekten) - gekürzte Wiederholung

Ein wesentlicher Vorteil der OOP besteht in der Kapselung von Daten. Zugriff auf Eigenschaften darf normalerweise nur über Zugriffsmethoden (*getter*- bzw *setter*-Methoden) erfolgen. Diese Methoden können Plausibilitätstests enthalten. So kann z.B. eine Methode zum Setzen des Geburtsdatums prüfen, ob das Datum korrekt ist und sich innerhalb eines bestimmten Rahmen bewegt, z.B. Girokonto für Kinder unter 14 nicht möglich oder Kunden über 150 Jahre unwahrscheinlich.

Die Verkapselung (encapsulation) sagt aus, dass zusammengehörende Attribute und Operationen, in einer Einheit – der Klasse – verkapselt sind. Die Attribute und die Realisierung der

[3] vgl. Karl-Heinz Rau: Objektorientierte Systementwicklung. Wiesbaden. 2007. S.76 ff.

Operationen kann durchaus nach aussen sichtbar sein. Beispielsweise erlaubt C++ mit seinen verschiedenen Sichtbarkeiten public, protected und private die Verkapselung ohne und mit Einhaltung des Geheimnisprinzips



In der Abbildung links realisiert das Objekt das *Geheimnisprinzip*. Zustand und Verhalten eines Objekts bilden eine Einheit. Wir sagen auch: ein Objekt kapselt *Zustand* (Daten) und *Verhalten* (Operationen). Die Daten eines Objekts können nur mittels der Operationen gelesen und geändert werden (Botschaft). Das bedeutet, dass die Repräsentation dieser Daten nach aussen verborgen sein soll. Wir sagen: ein Objekt realisiert das Geheimnisprinzip.[4]

Beispiel Datenkapselung (Abbildung rechts): Ein Mitarbeiter besitzt eine Personalnummer, einen Namen und erhält ein bestimmtes Gehalt. Neue Mitarbeiter werden eingestellt, das Gehalt vorhandener Mitarbeiter kann erhöht werden und es kann ein Mitarbeiterausweis gedruckt werden. Wie die Abbildung rechts zeigt, werden die Attribute durch die Operationen vor der Aussenwelt verborgen.

Die Idee dabei ist, eine Art Schale um das Objekt zu legen, so dass dieses vor direkten Zugriffen geschützt ist. Viele Programmiersprachen stellen dafür verschiedene Sichtbarkeiten für Attribute und Methoden bereit. Diese Sichtbarkeiten (meist public, protected oder private genannt) „steuern“ den Zugriff von aussen.

3.1. Datenkapselung in Python

Zum einen existiert eine Datenkapselung in Python, zum anderen existiert sie nicht wirklich. Beispielsweise können Attribute „streng“ (private) gekapselt werden, indem ein zweifacher Unterstrich vor den Attributnamen gesetzt wird:

```
# Abteilungsleiter6.py (Ausschnitte!)
#...
def __init__(self, name, gehalt, bueroraum):
    Mitarbeiter.__init__(self, name, gehalt) # Aufruf des übergeordneten Konstruktors
    self.buero = bueroraum
    self.__umsatz = 0

#...
# Ausgabe
def zeigeAbteilungsleiter(self):
    print "Abteilungsleiter-Ausgabe"
    print "Name: ", self.name, ", Gehalt: ", self.gehalt
```

[4] aus: Heide Balzert: Lehrbuch der Objektmodellierung. Analyse und Entwurf. Heidelberg, Berlin 1999. S. 18 und 20

```
        print "Buero:␣", self.buero.ausgabe()
        print "Umsatz:␣", self.__umsatz
        print "-----------------------------------"
# ...
```

Dann ergibt der Aufruf im Hauptprogramm (Main6.py):

```
# Ausschnitt aus Main6.py
leiter2 = Abteilungsleiter6("Manfred", 15000, buero1)
leiter1.zeigeAbteilungsleiter()     # beachte: kein self in der Klammer beim Aufruf!
leiter2.umsatz = 98000
```

die Ausgabe (im Ausgabe-Fenster):

```
Abteilungsleiter-Ausgabe
Name :  Manfred , Gehalt:  15000
Buero:  Bueronummer 0055302  mit Groesse 13 qm
None
Umsatz:  0
-----------------------------------
```

Wie sich erkennen lässt, ist der Umsatz *nicht* geändert worden, da ein direkter Zugriff nicht möglich ist. Allerdings ist auch keine (erwartete) Fehlermeldung ausgegeben worden. Bei einem Zugriff auf ein private deklariertes Attribut wird in Python keine Fehlermeldung ausgegeben!

Hintergrund: In Python findet keine strikte Datenkapselung statt, es handelt sich eher um Richtlinien, die der Entwickler befolgen sollte. Wenn nicht anders angegeben, sind in Python Klassen und Attribute immer per Default als public definiert und können somit von aussen uneingeschränkt zugegriffen werden. Die nächste Sichtbarkeitskategorie ist der *protected-* oder in Python auch weak-Modus[5] genannt. Dieser wird mit *einem* Unterstrich vor der Klasse oder Attribut eingeleitet. Es lässt sich weiterhin lesend und schreibend auf diese Attribute zugreifen, jedoch werden sie nicht automatisch in den aktuellen Namensraum eingebunden und müssen somit vom Namen her bekannt sein. Diese Variante stellt eher eine Markierung für den Entwickler dar, die aussagt, dass auf diese Attribute nicht direkt zugegriffen werden sollte. Die letzte zur Verfügung stehende Variante ist der *private-* oder auch strong-Modus. Dieser wird durch zwei Unterstriche eingeleitet.[6]

4. Vertiefung

- Ein ergänzendes Tutorial zur Vererbung findet sich u.a. auf http://www.python-kurs.eu/python3_vererbu
- Weiteres Material online: http://python.robakowski.com/oop.htm

[5] weak, engl. = „schwach“

[6] vgl. auch http://www.python-kurs.eu/klassen.php Dort das Unterkapitel Datenkapselung