Objekte, Typen, Typhierarchien, Instanzen, Klassen

Die in der Überschrift genannten Begriffe finden sich einerseits innerhalb linguistischer und ontologischer Modelle als andererseits auch in Programmiersprachen wie z.B. Python. Dabei sind bestimmte konzeptuelle Gemeinsamkeiten zu beobachten, andererseits gibt es aber auch gravierende Unterschiede.

Wir wollen zunächst, im ersten Teil, die Terminologie einführen so, wie sie im Rahmen der Linguistik bzw. der Semantik bzw. der Wissensrepräsentation eingesetzt wird. Mit Bezug darauf können wir dann im zweiten Teil die in Python verwendete Begrifflichkeit präzisieren. Im dritten Teil geht es dann darum, auf der Grundlage des zweiten Teiles selber Typen in Python zu definieren.

Teil 1: Allgemeine Grundlagen

Die nachfolgenden Ausführungen beziehen sich auf Typhierarchien im Rahmen verschiedener theoretischer und praktischer Modelle der Linguistik – z.B. im Rahmen der HPSG, wenn es um die Wortartenklassifikation geht, oder im Rahmen diverser semantischer Modelle und der Wissensrepräsentation im Zusammenhang mit der Organisation begrifflicher Konzepte.

Was ist ein **Objekt**?

Alles ist ein Objekt. In vielen Fällen wird 'Objekt' als 'Entität' bezeichnet (von lat. *ens*: "das Seiende"). Alle Typen, Subtypen, Instanzen von Typen und Klassen sind Objekte.

Was ist ein **Typ**?

Ein Typ ist die Charakterisierung derjenigen Attribute, die die Objekte, die diesem Typ angehören, aufweisen:

Typ AUTO: hat 4 Räder, hat Platz für 5 Personen, hat einen Motor, dient der Fortbewegung usw.
Typ MOTORRAD: hat 2 Räder, hat Platz für 2 Personen, hat einen Motor, dient der Fortbewegung usw.

Typen werden in Typhierarchien organisiert, bei dem jeder Subtyp die Attribute seines Supertyps erbt. Das macht die Geschichte ökonomisch, da bestimmte Eigenschaften nicht mehr für jeden Subtyp notiert werden müssen, sondern nurmehr einmal beim Supertyp:

Typ MOTORFAHRZEUG: hat einen Motor, dient der Fortbewegung usw.
Typ AUTO: ist ein Motorfahrzeug, hat 4 Räder, hat Platz für 5 Personen usw.
Typ MOTORRAD: ist ein Motorfahrzeug, hat 2 Räder, hat Platz für 2 Personen usw.

Der Supertyp ist selber Subtyp eines höher stehenden Typs:

Typ FORTBEWEGUNGSMITTEL: dient der Fortbewegung usw.
Typ MOTORFAHRZEUG: ist ein Fortbewegungsmittel, hat einen Motor usw.
Typ AUTO: ist ein Motorfahrzeug, hat 4 Räder, hat Platz für 5 Personen usw.
Typ MOTORRAD: ist ein Motorfahrzeug, hat 2 Räder, hat Platz für 2 Personen usw.

Was ist eine **Typhierarchie**?

Zwischen Subtyp und Supertyp liegt eine asymmetrische Relation vor. Die Gesamtmenge der asymmetrischen Relationen ergibt eine hierarchische Struktur, die **Typhierarchie**.

Je höher man in der Hierarchie geht, desto allgemeiner sind die Charakterisierungen, die durch die Menge der Attribute geliefert werden, je weiter nach unten man geht, desto spezifischer sind sie.

Ganz oben in jeder Hierarchie befindet sich der **allgemeinste Typ**. Diesen bezeichnen wir schlicht als **TYP**, der Typ, aus dem alle Typen abgeleitet sind:

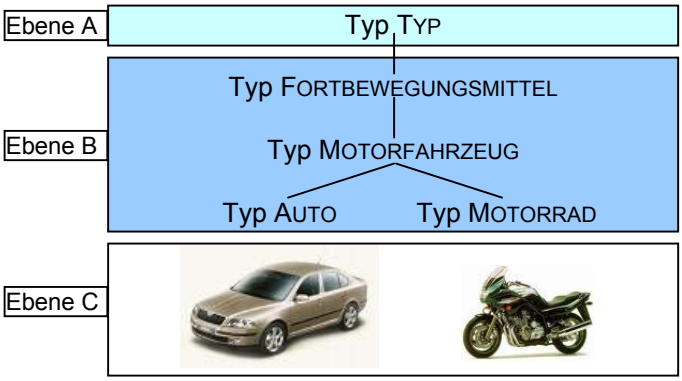
TYP
[...]
Typ FORTBEWEGUNGSMITTEL: dient der Fortbewegung usw.
Typ MOTORFAHRZEUG: ist ein Fortbewegungsmittel, hat einen Motor usw.
Typ AUTO: ist ein Motorfahrzeug, hat 4 Räder, hat Platz für 5 Personen usw.
Typ MOTORRAD: ist ein Motorfahrzeug, hat 2 Räder, hat Platz für 2 Personen usw.

Was ist eine **Instanz** eines Typs?

Typen können durch konkrete Objekt instanziiert sein. Die nebenstehende Yamaha ist beispielsweise eine Instanz des Typs MOTORRAD, der Skoda eine Instanz des Typs AUTO. Damit sind sie auch Instanzen der jeweiligen Supertypen, also MOTORFAHRZEUG und FAHRZEUG.

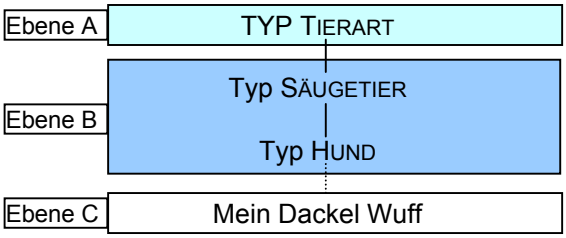
Für den allgemeinsten Typ gilt, dass alle aus diesem abgeleiteten Typen eine Instanz dieses Typs sind. Somit ist der Typ FORTBEWEGUNGSMITTEL eine Instanz des allgemeinsten Typs, ebenso der Typ MOTORFAHRZEUG, der Typ AUTO und der Typ MOTORRAD.

Wir können mithin drei verschiedene Hierarchieebenen voneinander differenzieren:



- Die Objekte in **Ebene B** sind Instanzen des allgemeinsten Typs auf **Ebene A**.
- Die Objekte in **Ebene C** sind Instanzen ihrer jeweiligen Typen und deren Supertypen auf **Ebene B**.
- Die Objekte in **Ebene C** sind keine Instanzen des allgemeinsten Typs auf **Ebene A**

Sehen Sie zur Verdeutlichung, was passiert, wenn wir die Ebenen unzulässig zusammenwürfeln, die folgende Typhierarchie und die dazu getroffenen Aussagen:



- Hunde sind Säugetiere. Mein Dackel Wuff ist ein Hund, also ist er ein Säugetier.
- Hunde sind Säugetiere. 'Säugetier' ist eine Tierart, also ist 'Hund' eine Tierart.
- 'Hund' ist eine Tierart. ?Mein Dackel Wuff ist ein Hund, also ist Wuff eine Tierart.

An diesem Beispiel wird gut deutlich, dass das umgangssprachliche 'ist_ein' zwei verschiedene Relationen abdeckt, nämlich die 'ist_ein_Subtyp'-Relation und die 'ist_eine_Instanz_von'-Relation. Wenn wir es präziser ausdrücken und davon ausgehen, dass die 'Instanz-von' Beziehung nicht ebenenüberschreitend sein kann, tritt der Fehlschluss nicht auf:

- HUND und SÄUGETIER (Ebene B) sind Instanzen des allgemeinsten Typs TIERART (Ebene A), wobei HUND ein Subtyp von SÄUGETIER ist.
- Wuff (Ebene C) ist eine Instanz von HUND, womit er auch eine Instanz von SÄUGETIER ist (beide Ebene B), nicht aber von TIERART (Ebene A).

Was ist eine **Klasse**?

Eine Klasse ist die Menge aller Objekte, die den jeweiligen Typ instanziieren. Wir können auf dieser Grundlage folgende Zuordnungen treffen:

TYP ⇒ intensionale Definition einer Klasse
KLASSE ⇒ extensionale Definition einer Klasse

Teil 2: Python

Was ist ein **Objekt** in Python?

Auch in Python gilt: alles ist ein Objekt, d.h. alle Typen, Instanzen von Typen, Attribute und Methoden sind Objekte.

Was ist ein **Typ** in Python?

Wir haben im Laufe des Seminares schon Beispiele dafür gesehen, was in Python unter *Type* fallen kann, nämlich das, was wir als *Datentypen* bezeichnet haben. Bei diesen handelt es sich um vordefinierte Typen:

```
>>> type(2)
<type 'int'>
>>> type('Hallo')
<type 'str'>
>>> type([1,2,3,4])
<type 'list'>
```

```
>>> type({1:2})
<type 'dict'>
>>> type(('a','b'))
<type 'tuple'>
```

Wodurch unterscheiden sich diese Typen voneinander? Sie unterscheiden sich durch die Menge und Art der **Attribute**, durch die sie jeweils charakterisiert sind. Diese Attribute kann man sich in Python mit der Funktion `dir()` anzeigen lassen.
Nachstehend als Beispiel die Attribute, die mit dem Typ `list` verbunden sind:

```
>>> dir(list)
['__add__',    '__class__',    '__contains__',    '__delattr__',    '__delitem__',
'__delslice__',    '__doc__',    '__eq__',    '__ge__',    '__getattribute__',
'__getitem__',  '__getslice__',  '__gt__',  '__hash__',  '__iadd__',  '__imul__',
'__init__',  '__iter__',  '__le__',  '__len__',  '__lt__',  '__mul__',  '__ne__',
'__new__',    '__reduce__',    '__reduce_ex__',    '__repr__',    '__reversed__',
'__rmul__',  '__setattr__',  '__setitem__',  '__setslice__',  '__str__',  'append',
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```
[1]

Wenn Sie die den o.a. Datentypen zugeordneten Attribute vergleichen, stellen Sie fest, dass relativ häufig die gleichen Namen auftauchen, z.B. findet sich die Methode **__add__** sowohl beim Typ **str**, als auch bei **list** und bei **int**. Wie wir aber am Anfang des Seminares bereits festgestellt habe, sind die Attribute typspezifisch, und in der Tat 'macht' z.B. die Methode **__add__** jeweils etwas anderes in Abhängigkeit des Typs seiner Argumente:

```
>>> zahl1 = 8
>>> zahl2 = 3
>>> zahl1.__add__(zahl2)
11
```

```
>>> kette1 = 'abc'
>>> kette2 = 'def'
>>> kette1.__add__(kette2)
'abcdef'
```

```
>>> liste1 = [1234]
>>> liste2 = [5678]
>>> liste1.__add__(liste2)
[1234, 5678]
```

Das in diesem Kontext relevante Konzept heißt **Namensraum**. Darunter zu verstehen ist der Geltungsbereich, den ein einen bestimmten Typ charakterisierendes Attribut hat: dieser ist nämlich auf genau diesen Typ beschränkt.

Das klingt etwas kompliziert, kann aber ganz einfach mit dem folgenden Beispiel erklärt werden: stellen Sie sich vor, in Firma ABC arbeitet ein Herr Müller in der Verwaltung, ein weiterer Herr Müller in der Werbeabteilung. Hier haben wir zwei verschiedene Objekte, die denselben Namen tragen. Wenn wir die beiden referenzieren wollen, würden wir umgangssprachlich so etwas sagen wie 'Müller aus der Verwaltung' oder 'Müller aus der Werbung'. Der Name 'Müller' hat also verschiedene Geltungsbereiche, nämlich 'Verwaltung' und 'Werbung'. Dieses sind zwei verschieden Namensräume, und in Abhängigkeit davon, welchem dieser Namensräume der Name 'Müller' angehört, bezeichnet er verschiedene Objekte. Um diesen Unterschied auch formal voneinander zu differenzieren, können wir eine Schreibweise verwenden, die an die Python-Notation angelehnt ist und wie folgt aussieht:

ABC.VERWALTUNG.MÜLLER ⇒ Müller aus der Verwaltung der Firma ABC

ABC.WERBUNG.MÜLLER: ⇒ Müller aus der Werbeabteilung der Firma ABC

Sie sehen, dass hier ein weiterer Namensraum vertreten ist, nämlich die Firma ABC – es sollte jetzt klar sein, warum: es könnte es ja auch eine Werbeabteilung und eine Verwaltung in einer Firma XYZ geben.

Zum Vergleich der Notation das folgende Beispiel aus der Python-Shell, das die Beispiele von weiter oben aufgreift und in alternativer Schreibweise gehalten ist:

```
>>> int.__add__(8,3)                    #Die Methode __add__ aus dem Namensraum int
11
>>> str.__add__('abc','def')            #Die Methode __add__ aus dem Namensraum str
'abcdef'
```

Dabei ist zu berücksichtigen, dass nicht nur Typen, sondern insbesondere eben Instanzen dieser Typen Namensräume definieren.

Was ist eine **Python-Typhierarchie**?

Eine Python-Typhierarchie ist in bestimmten Punkten analog zu einer 'normalen' Typhierarchie:

- alle Typen sind aus dem allgemeinen Typ an der Spitze der Hierarchie abgeleitet
- es gilt das Vererbungsprinzip

Zwei wesentliche **Unterschiede** sind aber in den folgenden Punkten zu sehen:

Die **Spitze** der Python Typhierarchie

An der Spitze stehen in der Python Typhierarchie **zwei** vordefinierte, primitive Konstrukte, nämlich **type** und **object**. Um das zu verstehen, müssen wir uns vor Augen halten, dass in Python (a) alles ein Objekt ist (also auch **type**) und (b) dass in Python jedes Objekt einem Typ zugeordnet ist (also auch **object**). Das bedeutet, dass diese beiden Konstrukte interdependent sind, also voneinander abhängig, was wir wie folgt prüfen können:

```
>>> isinstance(type,object)          #type ist eine Instanz von object
True
>>> isinstance(object, type)         #object ist eine Instanz von type
True
```

Dieser Punkt ist relevant, wenn es darum geht, neue Typen abzuleiten: diese werden i.d.R. in Python 2.2 aus **object** abgeleitet, nicht aus **type**.

[1] sehen Sie den Appendix am Ende für eine detaillierte Erläuterung

Python **Instanzen**

Wie in der 'normalen' Typhierarchie sind auch Python-Typen durch Objekte instanziiert. Hier gibt es also durchaus Gemeinsamkeiten zwischen Python-Typhierarchien und 'normalen' Typhierarchien – aber eben auch Unterschiede.

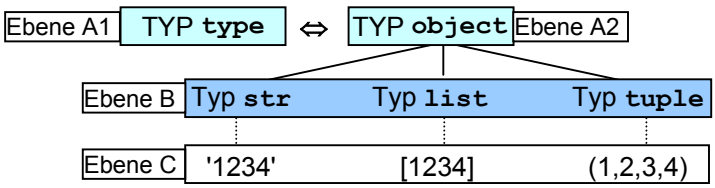
Wenn Sie sich die 'Tierart'-Hierarchie auf Seite 2 ansehen, werden Sie feststellen, dass das, wovon bei derartigen Hierarchien ausgegangen wird, die konkrete Instanz ist. Es gibt – in der 'echten' Welt – eine Reihe von Objekten, wie z.B. meinen Dackel Wuff, die Perser- und die Siamkatze meines Nachbarn, den Beagle von meiner Schwester usw. Diese beobachten wird und legen dann ein Klassifizierungssystem aufgrund gemeinsamer Merkmale und Unterschiede fest. Hier ist der Ursprung der Hierarchie in den Instanzen gelegen, d.h. wir haben es hier mit einem induktiven System zu tun, bei dem wir vom Spezifischen auf das Allgemeine schließen. Die konkreten Instanzen führen gewissermaßen eine von ihrem Typ unabhängige Existenz.

In der Python-Typhierarchie (und das gilt für alle Programmiersprachen) dagegen ist die Situation völlig anders. Hier kann nicht 'Konkretes' unabhängig von seinem Typ existieren. Wenn Sie versuchen, einer Variable ein nicht typspezifiziertes Objekt zuzuweisen, also etwas, das nicht eindeutig als Zeichenkette, Liste, Dictionary etc. ausgewiesen ist, erhalten Sie prompt eine Fehlermeldung:

```
>>> X = abcde
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    X = abcde
NameError: name 'abcde' is not defined
```

Wir können also davon ausgehen, dass die Herangehensweise bei der Python-Typdefinition deduktiv ist, d.h. der Weg führt vom Allgemeinen (**`type`** bzw. **`object`**) auf das Spezifische.

Kommen wir mit diesen Erkenntnissen zurück zu den Ebenen-Darstellungen der Typhierarchien auf Seite 1 und 2. Diese müssen wir mit Bezug auf die Python-Typhierarchie, genauer gesagt auf deren oberste Ebene, renotieren, da wir es ja hier mit zwei interdependenten allgemeinen Typen zu tun haben:



Für diese Hierarchie gelten dieselben Maßgaben wie für die Hierarchien auf S. 1 und S. 2, d.h.

- Die Objekte in **Ebene B** sind Instanzen des allgemeinsten Typs auf **Ebene A1**.
- Die Objekte in **Ebene C** sind Instanzen ihrer jeweiligen Typen (und ggf. deren Supertypen) auf **Ebene B**.
- Die Objekte in **Ebene C** sind keine Instanzen des allgemeinsten Typs auf **Ebene A1**

Wie aber bereits weiter oben deutlich wurde, muss folgendes beachtet werden:

- **Alle Objekte** auf **allen Ebenen** sind Instanzen des allgemeinsten Typs auf **Ebene A2**

Diese Aussagen können wir mit der Funktion **`isinstance()`** in Python überprüfen, hier am Beispiel einer Zeichenkette und einer Liste als Objektinstanzen von **`str`** und **`list`**:

Ebene B ⇒ Ebene A1

```
>>> isinstance(str,type)
True
>>> isinstance(list,type)
True
```

Ebene C ⇒ Ebene B

```
>>> isinstance('1234',str)
True
>>> isinstance([1234],list)
True
```

Ebene C ∅ Ebene A1

```
>>> isinstance('1234',type)
False
>>> isinstance([1234],type)
False
```

Ebenen A1,A2,B,C ⇒ Ebene A2

```
>>> isinstance(type,object)
True
>>> isinstance(object,object)
True
>>> isinstance(str,object)
True
>>> isinstance(list,object)
True
>>> isinstance('1234',object)
True
>>> isinstance([1234],object)
True
```

Was ist eine Python **Klasse**?

In Python ab Version 2.2 (und auch in kommenden Versionen) können wir die Begriffe 'Klasse' und 'Typ' synonym verwenden. Das war in den Vorgängerversionen anders. Auf diesen Punkt wollen wir hier nicht näher eingehen, um Verwirrung zu vermeiden, auf jeden Fall hat er aber insofern Konsequenzen für die aktuelle Python-Version, als die Anweisung, die wir für die Definition neuer Typen verwenden, (noch) den Namen `class` trägt. In Teil 3 verwenden wir mal den Begriff 'Klasse', mal 'Typ', meinen aber das Gleiche.

Zusammenfassung des zweiten Teils:

- **Alles** in Python ist ein Objekt
- Es gibt **type-Objekte** (z.B. `list`, `str`, `dict`) und **non-type-Objekte** (z.B. `[1,2,3]`, `'abc'`, `{'a':1}`)
- Entsprechend gibt es **Klassen** von type-Objekten und **Klassen** von non-type-Objekten
- **type-Objekte** sind in einer Hierarchie organisiert und durch **non-type-Objekte** instanziiert.

Teil 3: Konkrete Umsetzung

In den nachstehenden Abschnitten wollen wir uns anhand eines konkreten Beispieles ansehen, wie man in Python eigene Typen definieren. Wir gehen dabei nur so weit, wie es für unsere Zwecke nötig ist. Das heißt konkret, dass wir hier nur Subtypen bereits existierender Datentypen aus Ebene B einführen werden. Das hat zwei Gründe:

1. wenn wir anders vorgingen, d.h. wenn wir Typen direkt aus 'object' oder 'type' aus Ebene A ableiteten, müssten wir um einiges tiefer in die Materie eindringen (wir müssten uns z.B. genauer mit der Methode zur Definition einer Instanz dieses Typs (`def __init__`) beschäftigen und der etwas komplizierten Frage, wie Instanzattribute 'vererbt' werden können)
2. es ist davon auszugehen, dass sich in diesem Bereich mit Python 3.0 (dessen Veröffentlichung unmittelbar bevorsteht und das auch nicht mehr abwärtskompatibel sein soll) einiges ändern wird.

Wir beginnen damit, umgangssprachlich den sehr lebensnahen Sachverhalt zu beschreiben, um den es geht, und sehen uns dann an, wie wir ihn mit Python modellieren können.

Sachverhalt

Da ich bei Freunden und Familienmitgliedern zinslos Geld leihen kann, auf mein Guthaben in der Bank aber Zinsen bekomme, leihe ich mir, wenn ich mal einen größeren Betrag benötige, lieber das Geld privat, als mein Konto anzuzapfen. Zum aktuellen Zeitpunkt liegen die folgenden Fakten vor:

1. ich schulde ich meinem Freund Heinz 350 Euro, meiner Freundin Jutta 125 Euro und meiner Mutter 240 Euro
2. ich habe 420 Euro auf der Bank

Da ich (a) oft neue Schulden mache und (b) gleichzeitig meine Schulden mit dem Geld von der Bank peu à peu abstottere, verliere ich zunehmend den Überblick, wem ich noch wieviel schulde, wieviel ich insgesamt schulde, wieviel ich überhaupt auf der Bank habe usw. Jetzt möchte ich Python einsetzen, um die Geschichte zu systematisieren.

Modellierung des Sachverhaltes und Umsetzung in Python

Was wir für unser Programm benötigen, ist zunächst einmal die Beschreibung der aktuellen Finanzlage, die wir gut in Form einer Tabelle notieren können:

1.

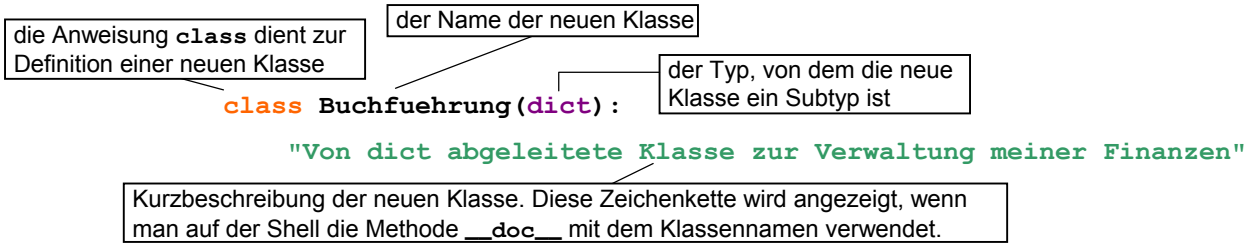
Gläubiger	Schulden
Heinz	350
Jutta	125
Mama	240

2.

Guthaben	420

Die Information in 1 können wir in Form eines Dictionaries erfassen, in dem der Gläubiger der *key*, die jeweilige Schuldensumme der *value* dieses *key* ist. Für dieses Dictionary wollen wir bestimmte Methoden definieren, die die Finanzverwaltung dann mit Bezug auf die Information in 2 ermöglichen.

Das bedeutet für uns also konkret, dass wir einen neuen Typ definieren wollen, der ein Subtyp des Typs `dict` ist. Diesen neuen Typ nennen wir `Buchfuehrung`:



Mit dieser Anweisung haben wir einen neuen Typ namens `Buchfuehrung` aus dem Typ `dict` abgeleitet.

Interessant wird die Sache allerdings erst, wenn es darum geht, typspezifische Methoden[2] zu definieren. Der folgende Screenshot zeigt, wie das Ganze in Python funktionieren soll:

```
1.  >>> Finanzen = Buchfuehrung({'Heinz':350,'Jutta':125,'Mama':240})
2.  >>> Finanzen.guthaben
3.  420
4.  >>> Finanzen.schuldensumme()
5.  715
6.  >>> Finanzen.leihen('Angela',65)
7.  >>> Finanzen.schuldensumme()
8.  780
9.  >>> Finanzen.items()
10. [('Mama', 240), ('Heinz', 350), ('Jutta', 125), ('Angela', 65)]
11. >>> Finanzen.rueckzahlen('Heinz',125)
12. >>> Finanzen.schuldensumme()
13. 655
14. >>> Finanzen.guthaben
15. 295
16. >>> Finanzen.rueckzahlen('Frank',50)
17. Keine Schulden bei Frank
18. >>> Finanzen.bankauszahlung(165)
19. >>> Finanzen.guthaben
20. 130
21. >>> Finanzen.rueckzahlen('Mama',200)
22. Zuwenig Guthaben
```

Erläuterung:

- In Zeile 1 wird der Variablen **Finanzen** ein Objekt des Typs **Buchfuehrung** zugewiesen. Darin ist festgelegt, wieviel Geld ich wem schulde. **Finanzen** ist sowohl eine Instanz von **Buchfuehrung** als auch von dessen Supertyp **dict**; **Buchfuehrung** ist ein Subtyp von **dict** und **object:**

```
>>> isinstance(Finanzen,Buchfuehrung)
True
>>> isinstance(Finanzen,dict)
True
```

```
>>> issubclass(Buchfuehrung, dict)
True
>>> issubclass(Buchfuehrung,object)
True
```

- In Zeile 2 und 3 wird der aktuelle Guthabenstand von **Finanzen** angezeigt. Dieser liegt beim ersten Aufruf des Programmes bei 420, entspricht also der oben in 'Sachverhalt' angegebenen Summe.
- Mit der Methode .**schuldensumme** werden die *values* aus **Finanzen** addiert und die Summe zurückgegeben (Zeilen 4 und 5)
- Die Methode .**leihen** hat zwei Argumente: den Gläubiger und den geliehenen Betrag. Wenn die Person bereits in der Menge der Gläubiger ist, wird der neu geliehene Betrag dem bereits geliehenen Betrag hinzu addiert, ansonsten wird ein neuer Eintrag im Dictionary gemacht (Zeilen 6 – 10)
- Die Methode .**rueckzahlen** hat zwei Argumente: den Gläubiger und den zurückgezahlten Betrag. Wenn Beträge zurückgezahlt werden, wird zunächst geprüft, ob dieses überhaupt möglich ist, sprich ob das Guthaben ausreichend ist. Wenn dem so ist, wird als nächstes geprüft, ob der Gläubiger tatsächlich in der Menge der Gläubiger vorkommt (und wenn nicht, wird eine entsprechende Meldung gemacht, Zeilen 16 und 17). Anschließend wird der Betrag sowohl vom Guthaben als auch vom Schuldbetrag im Dictionary abgezogen (Zeilen 11 – 15). Sollte allerdings das Guthaben nicht ausreichen, kommt eine entsprechende Meldung (Zeilen 21 und 22).
- Wichtig: wie man in Zeile 9 gut sehen kann, stehen **Finanzen alle dict**-Methoden zur Verfügung (hier: **.items()**, aber auch z.B. **Finanzen['Jutta']** (gibt **125** zurück) oder **Finanzen.values()** (gibt eine Liste der geschuldeten Summen zurück usw.).

Ferner definieren wir noch zwei weitere Methoden, die auf dem Umstand basieren, dass vom Bankguthaben auch unabhängig von konkreten Leih- und Rückzahlvorgängen Beträge ein- bzw. abgehen:

- Die Methode .**bankauszahlung** hat ein Argument: den Betrag, der vom Guthaben subtrahiert wird (Zeilen 18 und 19).
- Die Methode .**bankeinzahlung** hat ein Argument: den Betrag, der zum Guthaben addiert wird.

[2] Wir verwenden 'Methode' hier eigentlich synonym mit 'Funktion'. Im Grunde sind alle Methoden Funktionen, allerdings objektgebundene Funktionen, wie wir gleich sehen werden.

Wenn wir in Python objektgebundene Funktionen, sprich Methoden definieren, benutzen wir dabei die gleiche Anweisung wie bei der Definition ungebundener Funktionen, nämlich den Befehl **def**. Dabei gilt es allerdings, folgendes zu berücksichtigen:

1. **Die Attribute müssen unterhalb der Typdefinition entsprechend eingerückt aufgeführt werden.** Erklärung: Die Notwendigkeit des korrekten Eindrückens haben wir bereits kennengelernt, und auch hier gilt, dass durch die Einrückung der Skopus der Attribute klargestellt wird. Konkret heißt das, dass alle Attribute, die den Typ charakterisieren, entsprechend eingerückt unter der Typdefinition aufgeführt werden müssen. Zu diesen Attributen zählt – neben den zu definierenden Methoden – die Zuweisung meines Startguthabens von 420 an die Variable **guthaben**:

```
class Buchfuehrung(dict):
      "Von dict abgeleitete Klasse zur Verwaltung meiner Finanzen"
      guthaben = 420
      def <Methode 1>
      def <Methode 2>
      def <Methode n>
```

2. **Jede Methode muss einen Bezug zu dem Instanzobjekt aufweisen, das durch den Typ charakterisiert ist, zu dessen Attributen die Methode zählt.**
   Erklärung: Wenn wir in Python eine ungebundene Funktion definieren, hat dieses die folgende Form:
   def Funktionsname(Arg_1,Arg_2,Arg_3…Arg_n):
   <Aktionsblock>
   Wir hatten dafür schon Beispiele gesehen, z.B.

```
def quadrat(zahl):
        return zahl * zahl
```

   Wenn wir eine objektgebundene Funktion im Rahmen einer Typdefinition definieren, müssen wir hier verdeutlichen, dass sich diese Methode ausschließlich auf Instanzen dieses Typs bezieht. Das erreichen wir dadurch, dass wir als erstes Argument bei jeder Methodendefinition den Parameter **self** übergeben. Dieses Argument muss in jeder Methodendefinition vertreten sein.
   Sehen wir uns dafür die Definition der Methode .**bankeinzahlung** an:

```
def bankeinzahlung(self,betrag):
        self.guthaben += betrag
```

   Weiter oben stand, dass die Methode .**bankeinzahlung** ein Argument nimmt, nämlich den eingezahlten Betrag. Im Rahmen der Methodendefinition kommt aber ein weiteres Argument hinzu, nämlich **self**, welches klarstellt, dass es sich hier um eine Methode handelt, die auf Instanzen des Typs **Buchfuehrung** angewendet wird. Um auch **guthaben** identifizieren zu können, wird hier ebenfalls ein **self** vorangestellt und somit wird **guthaben** als zur gleichen Objektinstanz gehörig markiert.
   In der konkreten Anwendung wird die Methode mit einer entsprechenden Instanz, beispielsweise., wie auf der vorigen Seite, **Finanzen**, vor dem Punkt aufgerufen. Das gilt natürlich für alle Methoden aus der Menge der Attribute von **Buchfuehrung**.

Statt vieler weiterer wortreicher Erklärungen bietet es sich auf der Grundlage des Gesagten nun an, einfach einmal das gesamte Programm anzusehen. Dieses finden Sie auf der nächsten Seite. Sie sollten dieses Programm in Python implementieren und damit herumspielen, um ein konkreteres Gefühl für die doch etwas abstrakten Verfahren zu bekommen. Beispiele für 'Herumspielen':

1. Sie könnten auf der Shell die folgenden Befehle eingeben:

```
>>> help(Buchfuehrung.leihen)
>>> help(Buchfuehrung)
>>> Buchfuehrung.__doc__
>>> print Buchfuehrung.bankauszahlung.__doc__
```

2. Sie könnten austesten, was passiert, wenn Sie die Methoden mit Objektinstanzen verwenden wollen, die nicht zum Typ **Buchfuehrung** gehören:

```
>>> Temp = {'Freddie':200, 'Atze':375}
>>> Temp.leihen('Atze',42)
```

3. Sie könnten versuchen, eigene Methoden zu definieren, z.B. die Methode **.glaeubiger,** die wie folgt die Gläubiger durchnumeriert und mit dem Schuldenbetrag in Klammern dahinter alphabetisch ausgibt:

```
>>> Finanzen.glaeubiger()
1 Angela (65)
2 Heinz  (350)
3 Jutta  (125)
4 Mama   (240)
```

```
# -*- coding: iso-8859-1 -*-

class Buchfuehrung(dict):
    "Von dict abgeleitete Klasse zur Verwaltung meiner Schulden"

    #Klassenattribut (wird bei Ausführung der Methoden überschrieben)
    guthaben = 420

    #Methoden
    def leihen(self,person,betrag):
        """person = der Gläubiger
        betrag = die geliehene Summe"""
        if person not in self:
            self[person]=betrag
        else:
            self[person]+=betrag

    def rueckzahlen(self,person,betrag):
        """person = der Gläubiger
        betrag = die zurückgezahlte Summe"""
        if self.guthaben >= betrag:
            if person not in self:
                print "Keine Schulden bei %s" % person
            else:
                self[person]-=betrag
                self.guthaben-=betrag
        else:
            print "Zuwenig Guthaben"

    def schuldensumme(self):
        """Gesamtsumme meiner Schulden"""
        return sum(self.values())

    def bankeinzahlung(self,betrag):
        """betrag = Betrag der Einzahlung.
        Einzahlung auf das Bankguthaben"""
        self.guthaben += betrag

    def bankauszahlung(self,betrag):
        """betrag: die abgehobene Summe. Diese wird vom Bankguthaben
        abgebucht, falls darauf noch genügend Geld vorhanden ist.
        Anderfalls wird eine entsprechende Meldung gemacht"""
        if self.guthaben >= betrag:
            self.guthaben -= betrag
        else:
            print "Zuwenig Guthaben"
```

Beachten Sie, dass die Objektinstanz von **`Buchfuehrung`** flüchtig ist - sobald Sie die Python-Shell restarten, ist der Wert von **`guthaben`** wieder 420 und Sie müssen auch das Gläubiger-Schuldensumme-Dictionary neu anlegen. Wenn man das Programm ernsthaft nutzen wollte, müssten die Daten in externen Dateien gespeichert und daraus wieder geladen werden. Das grundlegende Verfahren haben wir dafür bereits kennengelernt (vgl. die Funktionen **`schreibe_datei()`** und **`lese_datei()`**), besser geht es aber unter Verwendung des Python-Modules **`pickle`**. Dazu an anderer Stelle mehr.