Übungen 2:
Hilfestellungen im Pseudo-Code, mögliche Lösungen und Erklärungen

Aufgabe 1

Pseudocode

```
Berechne das Quadrat wie folgt:
Hole vom User eine Zahl N
Setze Q auf das Ergebnis von N * N
Gib zurück 'Das Quadrat von N ist Q'
```

Python

```
def berechne_quadrat():
    N = input('Bitte Zahl eingeben\n')
    Q = N * N
    return 'Quadrat(%d) ist %d' %(N,Q)
```

Aufgabe 2

Die Fakultät einer natürlichen Zahl n ist das Produkt aller Zahlen von 1...n. Die Fakultät der Zahl 3 wäre 1*2*3, die Fakultät der Zahl 6 wäre 1*2*3*4*5*6 usw. Formal dargestellt wird die Fakultät von n durch n!. Fangen wir mit konkreten Beispielen an, deren Ablauf wir zeilenweise darstellen:

1! = 1	1! = 1	1! = 1
2! = 1 * 2	2! = 1 * 2	2! = 1 * 2
3! = 1 * 2 * 3	3! = 1 * 2 * 3	3! = 1 * 2 * 3
	4! = 1 * 2 * 3 * 4	4! = 1 * 2 * 3 * 4
		5! = 1 * 2 * 3 * 4 * 5

Wie Sie hier sehen können, läuft immer dasselbe ab. Ausgehend von einem Startwert '1' für Fakultät können wir das Verfahren an 4! wie folgt beschreiben:

Zahl, um die es geht	Berechnung der Fakultät	Welchen Wert hat Fakultät jetzt?
1	Fakultät = 1	Fakultät = 1
2	Fakultät = Fakultät * 2	Fakultät = 2
3	Fakultät = Fakultät * 3	Fakultät = 6
4	Fakultät = Fakultät * 4	Fakultät = 24

Mit Ausnahme der ersten Zeile, in der ein konkreter Wert für Fakultät gesetzt wird, wird in jeder weiteren Zeile Bezug genommen auf den in der Zeile zuvor ermittelten Wert für Fakultät. Diese wird dann mit einem 'Multiplikator' mulitipliziert. Das Ganze beginnt mit Fakultät '1' * Mulitiplikator '2' Wir müssen also Fakultät zunächst dem Wert 1 zuweisen, Mulitplikator dem Wert 2 und beide multiplizieren. Der Multiplikator wird dann in jedem weiteren Schritt um 1 erhöht; das Ganze endet, wenn Multiplikator und Fakultät den gleichen Wert haben.

Pseudocode

```
Berechne Fakultät wie folgt:
Weise F den Wert 1 zu
Weise M den Wert 2 zu
Hole vom User eine Zahl N
Mache, solange der M≤ N ist, folgendes:
      Setze F auf F * M
      Erhöhe den Wert von M um 1
Gib am Ende zurück N! = F
```

Python[1]

```
def berechne_fakultaet():
  F = 1
  M = 2
  N = input('Bitte Zahl eingeben\n')
  while M <= N:
     F = F * M
     M = M + 1
  return '%d! = %d' %(N,F)
```

Zusatzaufgabe: eine rekursive Definition für Fakultät

Eine rekursive Regel ist eine Regel, die auf sich selbst angewendet werden kann bzw. sich selbst aufruft. Aus der Syntax sind uns rekursive Regelsysteme gut bekannt, beisielsweise in

NP → Det (AP) N (PP)
PP → P NP

Hier kann von der NP-Regel aus die PP-Regel aufgerufen werden, von dieser wieder die NP-Regel und so weiter. Rekursion ist hier, informell gesagt, über zwei Regelrümpfe verteilt.

Ein einfaches 'natürlicher-Menschverstand-Beispiel' dafür, dass der erneute Aufruf einer Regel im eigenen Regelrumpf steht, ist die folgende Definition:

Jemand ist ein Italiener, wenn er in Italien geboren wurde oder wenn sein Vater ein Italiener ist.

[1] Diese iterative Definition haut nur hin, wenn N eine natürliche Zahl ist. Um Probleme zu vermeiden, könnte man prüfen lassen, ob der User wirklich ein natürliche Zahl eingibt.

Wir wissen, dass Vito in Italien geboren wurde: er ist also Italiener. Wo Fredo, Robert und Santino geboren wurden, wissen wir nicht. Was wir aber wissen, ist dass Fredo und Santino zwei Söhne von Vito sind. Da dieser als Italiener etabliert ist, können wir ableiten, dass auch Fredo und Santino Italiener sind. Über Robert können wir nichts aussagen.

Wie kann man so etwas in Python umsetzen? Dabei ist zunächst zu berücksichtigen, dass wir auf jeden Fall eine Situation angeben müssen, in der kein weiterer Aufruf der Regel erfolgt, da wir ansonsten in einer Endlosschleife landen. In den Syntax-Regeln weiter oben ist dieser Fall erreicht, wenn in der Ableitung eine NP auftaucht, die keine PP dominiert, z.B. in

[*das Buch* [*auf* [*dem Tisch* [*neben* [*dem Regal* [*unter* [*der Treppe*]$_{\mathbf{NP4}}$]$_{PP3}$]$_{NP3}$]$_{PP2}$]$_{NP2}$]$_{PP1}$]$_{NP1}$

Bei NP_4 ist sozusagen Schluss: diese konstituiert sich aus Det und N.

Bei dem Italiener-Beispiel ist das Ende erreicht, wenn das Teilziel 'ist in Italien geboren' erreicht wird. Was Python wissen muss, um herauszufinden, ob eine Person X Italiener ist, ist, wer alles in Italien geboren wurde und wer Vater von wem ist. In Zeile 7 findet der rekursive Aufruf statt:

Pseudocode

```
1. Bilde eine Menge von Personen, die in Italien geboren sind.
2. Bilde eine Menge von Vater-Sohn-Paaren
3. Ermittele, ob eine Person Italiener ist:
   4. Wenn die Person in der Menge von Personen ist, die Italien geboren sind:
   5. Gib zurück 'Die Person ist Italiener'
6. Ansonsten:
   7. Wenn die Person einen Vater hat, der Italiener ist:
   8. Gib zurück 'Die Person ist Italiener, weil ihr Vater Italiener ist'
```

An dieser Stelle muss man sich fragen, auf welche Weise man die Zeilen 1. und 2. am besten in Python umsetzt, genauer gesagt, welchen Datentyp man einsetzt. Die Antwort auf diese Frage hängt davon ab, was man später mit diesen Daten anfangen will, und wie die weiteren Zeilen zeigen, brauchen wir für Zeile 1 ein Format, bei dem wir die Menge einfach durchsuchen können, und für Zeile 2 eines, in dem bestimmte Elemente anderen Elementen zugeordnet sind. Für Zeile 1 bietet sich also eine Liste an, für Zeile 2 ein Dictionary oder eine Liste von Tupel. Hier wurde ein Dictionary verwendet:

Python

```
IN_I_GEB = ['Vito', 'Genco', 'Luca']
SV = {'Fredo':'Vito', 'Santino':'Vito', 'Fausto':'Luca'}
def Ist_Italiener(Person):
    if Person in IN_I_GEB:
      return '%s ist Italiener' %Person
    else:
      if Person in SV_keys() and Ist_Italiener(SV[Person]):
      return '%s ist Italiener, weil sein Vater Italiener ist' %Person
      else:
          return False
```

Wenn Sie sich die Erläuterungen zu Fakultät ansehen und insbesondere die Tabelle, sehen Sie, dass man die Fakultät einer Zahl N rekursiv leicht ermitteln kann, wenn man die Fakultät von N – 1 kennt: diese muss dann nur noch mit N multipliziert werden. Die Abbruchbedingung ist bei der Zahl 0 erreicht: deren Fakultät ist per Definition '1'.

Pseudocode

```
Berechne die Fakultät von N wie folgt:
    wenn N gleich 0 ist,
       gib 1 zurück
    Ansonsten gilt:
       gib Fakultaet von N - 1 * N zurück
```

Python

```
def fakultaet(N):
  if N == 0:
     return 1
  else:
     return fakultaet(N-1) * 1
```

Zu berücksichtigen ist hier, dass eine 'freundliche' Ausgabe à la 'Die Fakultät von N ist F' nicht so ohne weiteres möglich ist, eben weil die Funktion sich selber aufruft: 'N' und 'F' werden m.a.W. bei jedem erneuten Aufruf der Funktion neu gesetzt werden. Man könnte diese Funktion aber in eine andere einbauen, in der dann auch wieder ein Usereingabe eingeholt wird:

Pseudocode

```
Berechne Fakultät wie folgt:
    Hole vom User eine Zahl N
    Berechne die Fakultät F von N
    Gib am Ende F! von N zurück.
```

Python

```
def fakul():
  N = input('Bitte Zahl eingeben\n')
  F = fakultaet(N)
  print '%d! ist %d' %(N,F)
```

Aufgabe 3

Bei dieser Aufgabe geht es darum, sukzessive eine Liste von Zeichenketten aufzubauen. In diesen Fällen weist man einer Variablen zuächst – sozuagen als Starter – eine leere Liste zu, die dann im weiteren Ablauf nach und nach um die vom User eingegebenen Sätze erweitert wird. Die Methode der Wahl dafür ist **`.append()`**. Wenn der User 'Stop' eingibt, ist Schluss. Vielleicht denkt man intuitiv, man könnte das Verfahren so beschreiben:

```
#Drucke eine Zeichenkettenliste:
    #Weise der Variablen ZKL die leere Liste zu
    #Hole vom User einen Satz ein
      #Wiederhole, solange der Usersatz nicht 'Stop' ist, folgenden Vorgang:
      #Hänge den vom User eingegebenen Satz an ZKL an
    #Gib ZKL zurück
```

Das kann aber nicht funktionieren, da die Anordnung der Elemente im Aktionsblock verhindert, dass die Eingabeaufforderung wiederholt wird: es würde die erste Usereingabe in ZKL geschieben, dann ist Schluss. Es gilt prinzipiell, darauf zu achten, in welcher Reihenfolge die einzelnen Aktionen abgearbeitet werden. Wir müssen das also anders machen, und dafür bietet sich eine **`while`**-Schleife an, die auf **`True`** gesetzt wird. Das bedeutet soviel wie "solange weitermachen, bis keine Bedingung formuliert ist, die zum Abbruch führt", und dieses 'solange weitermachen' ist im konketen Fall das Einholen eines Satzes vom User: diese Anweisung muss also unter der **`while`**-Schleife stehen. Der Befehl 'Breche ab' wird per **`break`** erzeugt.

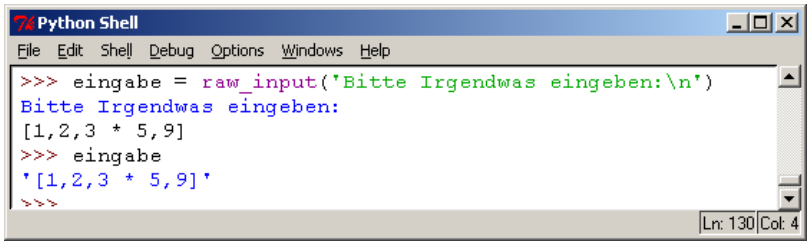
Pseudocode

```
Drucke eine Zeichenkettenliste:
    Weise der Variablen ZKL die leere Liste zu
    Wiederhole folgendes:
      Hole vom User einen Satz ein
      Wenn die Usereingabe 'Stop' ist: breche den Vorgang ab
      Hänge den vom User eingegebenen Satz an ZKL an
    Gib ZKL zurück
```

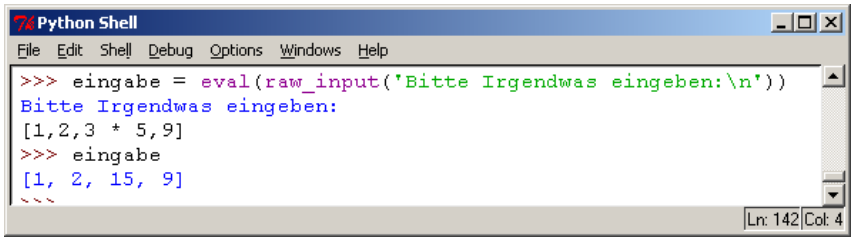
Python

```
def drucke_zk_liste():
     ZKL = []
     while True:
       Eingabe=raw_input("Bitte Satz eingeben. Vorgang mit 'Stop' beenden:\n")
       if Eingabe == 'Stop': break
       ZKL.append(Eingabe)
     return ZKL
```

Im Gegensatz zu den Aufgaben 1 und 2 haben wir hier nicht **`input()`**, sondern **`raw_input()`** verwendet. Was ist der Unterschied? Nun, **`raw_input()`** gibt die Eingabe – egal, worum es sich dabei handelt, also arithmetischer Ausdruck, Liste, Tupel etc. – in Form einer Zeichenkette für die Weiterverarbeitung zurück:



`input()` dagegen verhält sich wie die interaktive Python Shell. Wenn man dort eine Eingabe macht, und das ist immer eine Zeichenfolge, auch wenn es sich um Ziffern und Interpunktionszeichen handelt, dann versucht Python, diese Eingabe auszuwerten. Ausdrücke – neben arithmetischen Ausdrücken sind auch Funktionsaufrufe Ausdrücke – werden immer ausgewertet. Neben der automatischen Auswertung in bestimmten syntaktischen Kontexten (z.B. **`<Variable> = <Ausdruck>`**) stellt Python auch eine Funktion **`eval(<Ausdruck>)`** zur expliziten Auswertung zur Verfügung, die bei Bedarf eine Zeichenkette auswertet. **`input()`** ist also äquivalent mit **`eval(raw_input())`**:



Aufgabe 4

Pseudocode

```
Rate die Zahl:
   Weise R die zu ratende Zahl zu
   Weise N 0 zu
   Solange N nicht gleich R ist, mache folgendes:
       Hole vom User eine Zahl N
             Wenn N < R ist:
                   Drucke 'Die Zahl ist zu klein. Nochmal von vorne'
             Wenn N > R ist:
                   Drucke 'Die Zahl ist zu groß. Nochmal von vorne'
   Drucke 'Richtig!'
```

Python

```
def rate_die_zahl():
   R = 14
   N = 0
   while not N == R:
       N = input('Bitte eine Zahl zwischen 1 und 50 eingeben\n')
             if N < R:
                   print 'Die Zahl ist zu klein. Nochmal von vorne'
             if N > R:
                   print 'Die Zahl ist zu groß. Nochmal von vorne'
   print 'Richtig!'
```

Kommentar 1: durch die Zuweisung N = 0 gleich zu Beginn ist für den ersten Durchgang auf jeden Fall gewährleistet, dass die **while**-Schleife 'loslegt'. Hier hätte auch jede andere Zahl stehen können, Hauptsache, sie ist nicht gleich der Ratezahl.

Kommentar 2: statt while not N == R hätte auch while N!= R stehen können.

Kommentar 3: bei dieser Aufgabe kam die Frage nach **if, if...else, else...if** und **elif** auf. Um diesen Punkt zu klären, müssen wir ein bisschen ausholen und uns die Sache etwas genereller ansehen.

Wenn wir in Python eine Bedingung für eine Aktionen ausdrücken wollen, tun wir das mit **if**: nach dem Motto "wenn Bedingung gegeben ist: führe Aktion aus".

Wenn <Bedingung> gegeben ist, führe <Aktion> aus.	`if Eingabe == 'Julia':` `  print 'Hallo Julia'`

In vielen Fällen sind aber verschiedene Bedingungen anzuführen, die jeweils verschiedene Aktionen bedingen, und in diesen Fällen müssen wir darauf achten,

(a) wieviele Bedingungen bzw. Aktionen überhaupt gegeben sind und

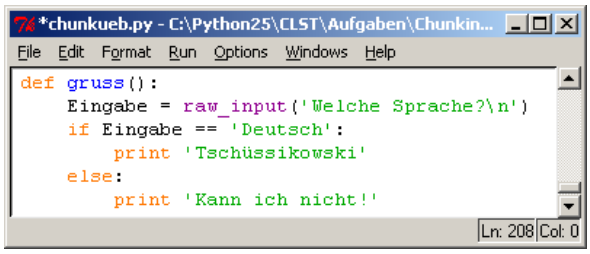
(b) wie diese eingerückt werden.

Bei **rate_die_zahl** gibt es nur drei Aktionen, nämlich Aktion$_1$: **print 'Zahl ist zu klein'**, Aktion$_2$: **print 'Zahl ist zu groß'** und Aktion$_3$: **print 'Richtig!'**. Die **while**-Schleife wird nur abgearbeitet, wenn N ≠ R, und in diesem Fall wird gepüft, ob N entweder größer oder kleiner als R ist, was jeweils verschiedene Aktionen zur Folge hat. Ist N aber = R, wird die **while**-Schleife sofort beendet und Aktion$_3$ ausgeführt.

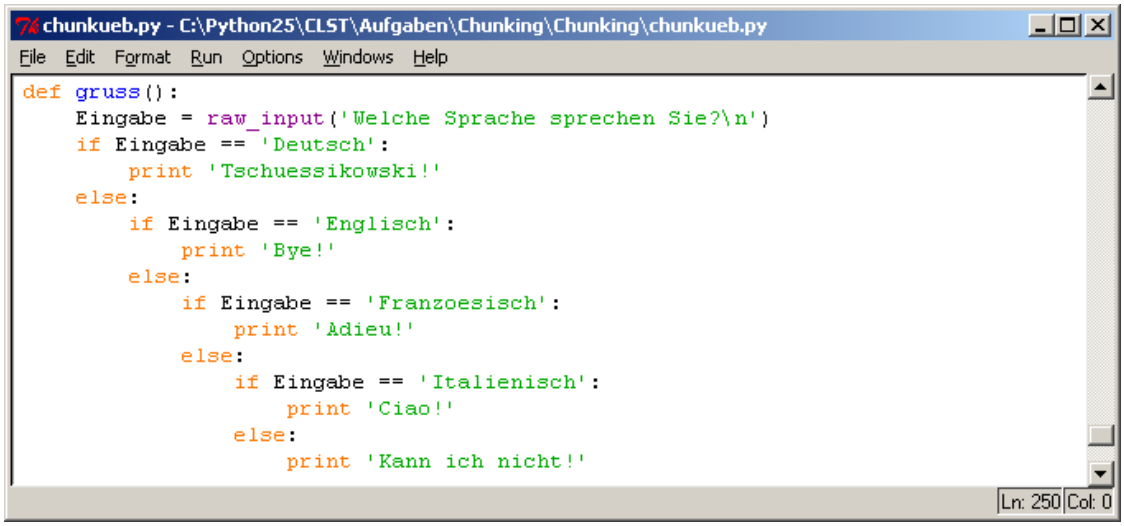
Wichtig ist dabei, dass diese letze Aktion, **print 'Richtig!'**, auf derselben Einrückungsstufe steht, wie die **while**-Schleife.

In diesem Beispiel hatten wir die Situation, dass entweder Bedingung$_1$ oder Bedingung$_2$ oder Bedingung$_3$ gegeben ist. Mehr Möglichkeiten waren gar nicht drin. Anders sieht es in dem folgenden Beispiel aus. Bei diesem soll das Programm in der Sprache reagieren, die vorher vom User eingegeben wurde.

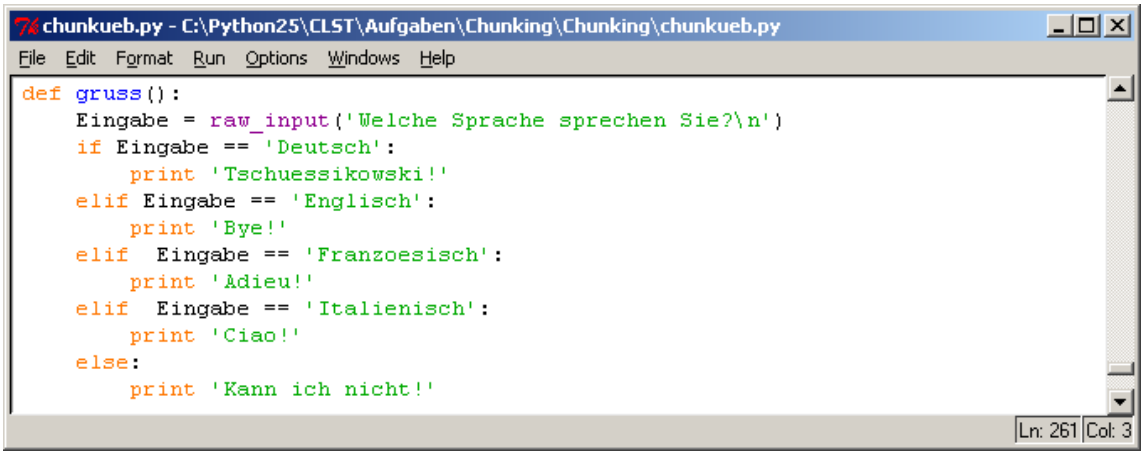
Ausgehend von der Annahme, dass bei ca. 6000 geschätzten Sprachen nicht für jede eine entsprechende Formulierung vorliegt, wollen wir für zunächst eine Bedingung definieren dafür, dass die Antwort auf Deutsch ist, und für alle anderen Fälle eine andere Meldung angeben. Hier brauchen wir etwas, das dem umgangssprachlichen 'Wenn Bedingung$_1$ gegeben ist, mache Aktion$_1$, **in allen anderen Fällen** mache Aktion$_2$'. Genau das wird durch **else:** ausgedrückt.



Gesetzt den Fall, wir wollen noch weitere Sprachen hinzunehmen, könnten wir das dann wie folgt machen:



Durch die Verschachtelung der **`else-if`**-Paare im Aktionsblock ist gewährleistet, dass die Bedingungen sukzessive abgearbeitet werden. An genau dieser Stelle nun setzt **`elif`** ein, denn mit dieser Anweisung haben wir **`else:...if`** quasi vereint. Das erspart uns die Verschachtelungstiefe der Einrückungen. Der folgende Screenshot zeigt, wie dieselbe Geschichte mit **`elif`** aussieht:



Um nun auf **`rate_die_zahl()`** zurückzukommen: wie beschrieben haben wir genau drei Bedingungen, die unterschiedliche Aktionen erfordern. Andere Fälle gibt es gar nicht, d.h. wir können die ganze Geschichte komplett **ohne `else:`** formulieren (wie auf Seite 4 zu sehen).

Wir könnten es zwar mit **`else:`** machen:

```
[...]
  while not N == R:
       N = input('Bitte eine Zahl zwischen 1 und 50 eingeben\n')
       if N < R:
            print 'Die Zahl ist zu klein. Nochmal von vorne'
       if N > R:
            print 'Die Zahl ist zu groß. Nochmal von vorne'
  else:
       print 'Richtig!'
```

Und wir könnten auch das zweite **`if`** auch ohne Konsequenzen durch ein **`elif`** ersetzen:

```
[...]
  while not N == R:
       N = input('Bitte eine Zahl zwischen 1 und 50 eingeben\n')
       if N < R:
            print 'Die Zahl ist zu klein. Nochmal von vorne'
       elif N > R:
            print 'Die Zahl ist zu groß. Nochmal von vorne'
  else:
       print 'Richtig!'
```

Diese beiden Lösungen sind aber (a) weniger klar als die Definition auf S. 4 und (b) auch nicht vollständig auf andere Fälle zu übertragen: wenn z.B. – wie in Aufgabe 3 – ein **`break`** in der **`while`**-Schleife vorkommt, würde eine solche Lösung zu Problemen führen.

Dateien lesen und schreiben: `file(Dateiname,'r')` und `file(Dateiname,'w')`

Aufgaben 5 & 6

Hier kam in der Sitzung ein Problem auf, dass dadurch gelöst wurde, dass **`schreibe_datei()`** wie folgt definiert wurde:

```
1. def schreibe_datei(Dateiname,Objekt):
2.       Datei = file(Dateiname,'w')
3.       if isinstance(Objekt,str):
4.            Datei.write('"'+Objekt+'"')
5.       else:
6.            Datei.write(str(Objekt))
7.       Datei.close()
```

Irgendwie schräg ist dabei Zeile 4, bei der **`Objekt`** in Anführungzeichen gesetzt wird, damit das Ganze hinhaut. Das Problem liegt aber nicht in der Definition von **`schreibe_datei()`**, sondern in der Kombination **`schreibe_datei()`** / **`hole_datei()`**. Bei letzerem ist **`Datei.read()`** durch **`eval()`** eingeklammert:

```
1. def hole_datei(Dateiname):
2.       try:
3.            Datei = file(Dateiname,'r')
4.            Objekt = eval(Datei.read())
5.            Datei.close()
6.            return Objekt
7.       except:
8.            return 'Das Objekt hat keinen Inhalt\n'
```

Das **`eval()`** dient ja zur expliziten Auswertung eines Ausdruckes (siehe S. 3). Wenn wir dieses in **`hole_datei()`** nicht benutzten, funktionierte auch die Eingabe von Zeichenketten mit einem schlichten **`Datei.write(Objekt)`** in Zeile 4. Dadurch verlieren wir allerdings die Möglichkeiten, Nicht-Zeichenketten wie z.B. Listen oder arithmetische Ausdrücke beim Einlesen auszuwerten.

Um es anders auszudrücken: das Ganze ist irgendwie verkorkst und blöd und diese beiden Definitionen waren ein schlechter Ort, um die Funktion **`isinstance()`** einzuführen. Pardon.

Wenn wir statt der Zeilen 3 - 6 in **`schreibe_datei()`** die **`repr()`**-Funktion nehmen, tauchen die Probleme von vorneherein nicht auf, da **`repr()`** jedes Objekt als Zeichenkette zurückgibt, also auch um Zeichenketten Anführungszeichen setzt.

Somit sieht **`schreibe_datei()`** folgendermaßen aus:

```
def schreibe_datei(Dateiname,Objekt):
    Datei = file(Dateiname,'w')
    Datei.write(repr(Objekt))
    Datei.close()
```

Teil 2: Arbeiten mit Dictionaries

Aufgabe 7

Hier haben wir wieder verschiedene Möglichkeiten für die Definition. Intuitiv scheint die folgende Möglichkeit einleuchtend:

Pseudocode

```
Drucke die Schlüssel wie folgt:
   Weise L die leere Liste zu
   Hänge dann jedes Element aus der
   Menge der Schlüssel in uebung1 an L
   Drucke am Ende L sortiert.
```

Python

```
def drucke_keys():
  L = []
  for element in uebung1.keys():
    L.append(element)
  print sorted(L)
```

Sehr viel schicker aber geht es per *List Comprehension* (≈ Listenabstraktion, falls Sie im Internet nachsehen wollen), allerdings fehlt mir die Phantasie dafür, diese in Pseudocode auszudrücken. Sehen Sie stattdessen nochmal das Handout *Tokenisierung*.

Python:

```
def drucke_keys():
  print sorted([key for key in uebung1.keys()])
```

Wir können auch schlicht folgendes machen:

Python:

```
def drucke_keys():
  print sorted(uebung1.keys())
```

Aufgabe 8

Hier hätten wir wieder verschiedene Möglichkeiten für die Definition. Am einfachsten ist:

Python:

```
def drucke_values1():
  print sorted(uebung1.values())
```

Aufgabe 9

Hier nun haben wir eine etwas andere Situation, da es jetzt darum geht, die Reihenfolge der Werte selber festzulegen. Der Wert eines Dictionary-Eintrages kann ja durch seinen Schlüssel referenziert werden, also ist es sinnvoll, zunächst die Reihenfolge der Schlüssel in einer Form anzulegen, die dann von Python sukzessive abgearbeitet werden kann, um darauf bezogen dann eine Liste der Werte zu erzeugen. Dafür bietet sich z.B. das Datenformat 'Liste' an.

Pseudocode

```
Drucke die Werte wie folgt:
     Weise L die Liste der Schlüssel aus uebung2 in gewünschter Reihenfolge zu
     Weise L1 eine leere Liste zu
     Gehe die Schlüssel in L durch und mache für jeden folgendes:
          Hänge den Wert von Schlüssel in uebung2 an L1
     Drucke L1
```

Python:

```
def drucke_values2():
     L = ['erstes', 'zweites', 'drittes', 'letztes']
     L1 = []
     for key in L:
          L1.append(uebung2[key])
     print L1
```

Das Ganze mit *List Comprehension*:

Python:

```
def drucke_values2():
     L = ['erstes', 'zweites', 'drittes', 'letztes']
     print [uebung2[key]for key in L]
```

Aufgabe 10

Pseudocode

```
Drucke die Werte wie folgt:
     Weise L die Liste der Schlüssel aus uebung2 in gewünschter Reihenfolge zu
     Weise L1 eine leere Liste zu
     Weise ZK eine leere Zeichenkette zuzu
     Gehe die Schlüssel in L durch und mache für jeden folgendes:
          Setze ZK auf den Wert dieses Schlüssels
          Hänge ZK an L1
     Gib folgendes zurück: die aus L1 abgeleitete Zeichenkette, in der die
     Element aus L1 durch ein Sternchen miteinander verbunden sind.
```

Python

```
def drucke_values3():
    L = ['erstes', 'zweites', 'drittes', 'letztes']
    ZK = ''
    L1 = []
    for key in L:
        ZK = uebung2[key]
        L1.append(ZK)
    return '*'.join(Liste)
```

Mit *List Comprehension*:

```
def drucke_values3():
    L = ['erstes', 'zweites', 'drittes', 'letztes']
    return '*'.join([uebung2.[key] for key in L)]
```

Aufgabe 11

Jetzt ist klar, wie der Hase läuft:

```
def drucke_tagkette():
    L  = ['the', 'boy', 'kicked', 'John']
    return '**'.join([uebung3.[key].lower() for key in L])
```