Lösungen zu ausgewählten Aufgaben

(3.2)

Der Umgang mit Variablen ist eigentlich so natürlich, dass das Hauptproblem ist, zu merken, wo sie stecken – auch wenn physiologisch sicher kein Programm „Kartoffeln einkaufen“ im Hirn ist, ist etwa klar, dass beim Dialog „Bringst du Kartoffeln mit?“ – „Wie viele denn?“ – „Na, vielleicht fünf Kilo.“ aus etwas überabstrahierter Sicht ein Programm („Kartoffeln kaufen“) aufgerufen wurde und eine Variable, die in diesem Programm vorkommt (nämlich die Menge) besetzt wurde.

Ganz ähnlich der Dreisatz – wir merken uns in der Regel nicht „Wenn 100g 2 Euro kosten, kosten 200g 4 Euro und ein Kilo kostet 20“, sondern „a ist c mal b durch d“ oder was ähnliches – jedenfalls werden „Platzhalter“ mit Werten besetzt und dann eine Rechenvorschrift (dieser Begriff ist beim Kartoffenkaufen sicher etwas strapaziert) ausgewertet.

Kurz: Wenn man weiß, wie man sucht, sind Variablen überall in unserem Leben. Was bleibt, ist die Formalisierung dieses Begriffs, und das wird uns noch etwas beschäftigen.

(5.2)

Richtig: raw_input gibt immer einen String zurück, selbst wenn die Eingabe auch als Zahl interpretierbar *wäre*. Das will man einerseits so haben, weil Programme sich später darauf verlassen können wollen, dass sie mit dem Ergebnis von raw_input wie mit einem String umgehen können (und etwa seine Länge bestimmen).

Andererseits wäre so eine automatische Vorverarbeitung auch nicht immer gut – wer weiß, ob ein Programm nicht daran interessiert ist, den Unterschied zwischen 0019 und 19 zu sehen? Einem int sieht man nicht mehr an, wie er entstanden ist. Oder denkt an 19.07 – das könnte auch durchaus als Zeit gemeint sein, und dann möchte man vermutlich lieber Stunden und Minuten noch im String trennen und zwei ints haben als einen float, zumal, wenn man die Schwierigkeiten, die Computer mit (dezimalen) floats haben, bedenkt.

(5.3)

In der ersten Zeile dividieren wir immer durch Potenzen von Zwei, und die Ergebnisse sind „genau“. In der zweiten Zeile dividieren wir durch Vielfache von fünf, und die Ergebnisse „stimmen“ jeweils in der letzten Ziffer nicht.

Hintergrund ist wie angedeutet, dass Python (wie auch die meisten CPUs, die überhaupt eine Ahnung von Fließkommazahlen haben) auch Fließkommazahlen intern „zur Basis 2“ speichern, das heißt, dass die Zahlen nur aus den Ziffern 0 und 1 bestehen. In diesem Format haben die meisten Zahlen, die in Dezimalschreibweise (mit Ziffern zwischen 0 und 9) nur endlich viele Ziffern haben, unendlich viele Ziffern. Mehr zu den verschiedenen Darstellungen von Zahlen erfahrt ihr in Programmieren II. Derweil könnt ihr euch merken, dass Zweierpotenzen für den Computer eine besondere Rolle spielen und es sich lohnt, wenigstens die ersten sechzehn auswendig zu kennen: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536. Außerdem ist es nützlich zu wissen, dass 2^20 ungefähr eine Million ist (genauer 1 048 576, was erklärt, warum ein „Megabyte“ je nach kommerziellem Nutzen mal etwas mehr, mal etwas weniger ist) und 2^32 rund vier Milliarden (genauer 4 294 967 296 – was der Grund ist, warum die am Anfang des 21. Jahrhunderts verbreiteten CPUs in der Regel um die 4 Gigabyte Speicher adressieren können).

(9.1)

```
incString(numberAsString) -> str
```

(9.2)

Es ist zunächst nicht selbstverständlich, dass das `i` im Hauptprogramm auch nach Ausführung der incString-Funktion immer noch den Wert `"23"` hat, zumal innerhalb der Funktion eine Variable namens i einen anderen Wert (nämlich int(s)) zugewiesen bekommt.

Grund für dieses Verhalten ist grob gesprochen, dass Variablen in Funktionen in der Regel „lokal“ sind, d.h. nur in der Funktion selbst sichtbar sind. Dies ist wichtig, um die Teilprobleme, in die wir unser großes Problem zerlegt haben, auch wirklich auseinanderzuhalten und damit für die Lokalität. Was es genau damit auf sich hat, werden wir später sehen.

(10.1)

Der entscheidende Punkt ist, dass Karl Rudolf sein Argument `str` genannt hat. Damit ist im Lokalen Namespace der Name `str` an den Wert `"7"` gebunden. Wenn nun am Ende der Funktion `str(i)` aufgerufen werden soll, sieht Python zunächst dort nach, findet den gesuchten Namen und gibt den Wert `"7"` zurück – der arme String weiß aber nicht, was er mit Argumenten anfangen soll oder wie er einen Wert zurückgeben soll („is not callable“), daher die Exception.

Generell ist es nicht empfehlenswert, irgendwelche Namen im globalen Namespace auch im lokalen zu definieren, auch wenn Python selbst damit keine Probleme hat – es verwirrt die LeserInnen von Programmen, und gelegentlich auch ihre AutorInnen. Das gilt ganz besonders für von Python vordefinierte Namen der eingebauten Funktionen.

(12.2)

Eine einfache Lösung könnte so aussehen:

```
for op1 in range(2):
  for op2 in range(2):
    print op1, op2, op1 and op2
```

(Weiterführend:) Es wäre elegant, das in eine Funktion zu packen, der dann übergeben werden kann, welchen logischen Operator sie verwenden soll. Das geht natürlich (sogar auf mehrere Arten), aber wir brauchen dazu Sprachelemente, die wir noch nicht hatten. Mal als Ausblick folgender Vorschlag:

```
def printTruthTable(oper):
  """prints the truth table for the logical operator oper.  oper is a
  string containing the operator as understood by python's evaluation
  engine.
  """
  for op1 in range(2):
    for op2 in range(2):
      print op1, op2, eval("%d %s %d"%(op1, oper, op2))

printTruthTable("and")
printTruthTable("or")
```

– mit eval kann man also offenbar Python-Ausdrücke in Strings auswerten.

(12.3)

```
def printCharIndices(aString):
  for index in range(len(aString)):
    print index, "\t", aString[index]
```

In der Tat scheint diese Funktion eher einfacher als das Beispiel im Skript. Das liegt hier daran, dass wir explizit die Indizes brauchen; in der Regel will man aber tatsächlich schlicht irgendetwas für alle Elemente der Sequenz tun, und dann ist die Iteration über die Sequenz selbst nicht nur schöner, sondern meist auch schneller.

(Weiterführend:) Der Fall, dass man das Sequenzelement und seinen Index gemeinsam braucht, tritt allerdings doch nicht so selten auf. Ab Python 2.3 gibt es daher eine eingebaute Funktion namens enumerate, die so angewandt wird:

```
def printCharIndices2(aString):
  for index, ch in enumerate(aString):
    print index, "\t", ch
```

(Wie gesagt, wenn ihr einen älteren Interpreter habt, bekommt ihr hier einen Fehler). Was hier mit den zwei Variablen hinter dem for passiert, werdet ihr sehen, wenn wir einen weiteren Sequenztyp, nämlich die Tupel, betrachten.

(12.4)

s ist ein String und damit unveränderbar. Um das zu erreichen, was hier offenbar bezweckt war, muss man einen neuen String erzeugen und s auf diesen String verweisen lassen. (Weiterführend:) Am einfachsten geht das mit Slices, die noch ein paar Folien weg sind; damit geht etwas wie

```
>>> s = s[:2]+"g"+s[3:]
```

(13.1)

```
def getFirstInfinitive(wordSeq):
  for word in wordSeq:
    if word.endswith("en"):
      return word
  else:
    return None
```

In diesem speziellen Fall wäre es auch zulässig, den else-Zweig komplett wegzulassen und nur ans Ende der Funktion ein return None zu schreiben (warum?). In der Tat könnte selbst dieses return None weggelassen werden, weil Funktionen, bei denen der Kontrollfluss einfach ohne return an ihr Ende „durchfällt“, automatisch den Wert None zurückgeben.

(14.1)

```
>>> "\\".join(path.split("/"))
```

(Weiterführend:) Wenn ihr Pfade baut, ist es wohl die beste Idee, das quasi plattformunabhängig mit der Funktion os.path.join zu machen.

(15.2)

1. l[4:6]
2. l[3:8]
3. l[3:]
4. l[:-1]
5. l[1:]
6. l[:4]+l[5:]
7. l[:3]+l[6:]
8. l[:]

Probiert aus, ob das funktioniert – Tipp: mit range(10) seht ihr an den in der Liste stehenden Zahlen, ob ihr das richtige erwischt habt. Macht euch jetzt klar, wie die Ausdrücke aussehen würden, wenn wir das umgangssprachlich erste Element nicht mehr als nulltes, sondern tatsächlich als erstes Element bezeichnen würden.

(15.3)

Es gibt keinen technischen Grund dafür – es wäre z.B. nicht schwer eine Funktion sortList zu schreiben, die eine Liste sowohl sortiert als auch zurückgibt (probiert es). Nein, der Grund ist psychologischer Natur: Die Programmierin soll erinnert werden, dass die Methoden das Objekt selbst verändern und also auch kein neues Objekt zurückgeben – was all die String-Methoden getan haben.

Tatsächlich kommt es leicht zu Konfusion, wenn ein und dieselbe Liste von verschiedenen Variablen referenziert wird. Würden Methoden wie sort oder reverse Referenzen zurückgeben, wären wilde Anfängerfehler bereits programmiert, etwa wie im folgenden:

```
(Nur ein Beispiel -- so funktioniert das im realen Python nicht!)
>>> l = range(10)
>>> l1 = l.sort()
>>> l1.append("boo")
>>> print l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 'boo']
```

Das hier angedeutete Problem kann auch beim realen Python vorkommen (vgl. Folie aliasing), aber mit weit geringerer Wahrscheinlichkeit versehentlich.

(15.4)

```
def reverseStr(aString):
  l = list(aString)
  l.reverse()
```

```
  return "".join(l)
```

(17.1)

Die einfachste Lösung ist:

```
for i in range(32, 128):
  print "%d -> '%s'"%(i, chr(i))
```

Dass bei 127 ein „leerer“ String ausgegeben wird, ist in Ordnung – auch dieses Zeichen dient als Steuerungszeichen, in diesem Fall als „Delete“ (was, wenigstens unter Unix, nochmal ein ganz anderes Thema ist).

Nett ist es, wenn man das zweispaltig ausgibt – dazu gäbe es verschiedene Möglichkeiten, mit unseren Mitteln wird man wohl am besten ein bisschen Mathematik einsetzen:

```
for i in range(48):
  print "%d -> '%s'\t%d -> '%s'"%(
    32+2*i, chr(32+2*i), 32+2*i+1, chr(32+2*i+1))
```

Die 48 ergibt sich dabei als die Zahl der Zeilen, die wir ausgeben wollen – bei $128 - 32 = 96$ Zeichen und zwei Zeichen pro Zeile sind das eben 48.

Könnt ihr das auf vierspaltige Ausgabe erweitern? Was passiert bei drei Spalten?

(19.3)

Damit auch nach drei `eval`s noch `'ab'` rauskommt, muss man schon ganz schöne Gartenzäune auffahren:

```
eval(eval(eval("'\"\\\\\"ab\\\\\"\"'")))
```

Wenn ihr das verstehen wollt, seht am besten nach, was nach jedem `eval` vom Eingabestring übrig geblieben ist. Alternativ kann man auch mit raw strings operieren:

```
eval(eval(eval(r"'r\'\"ab\"\''"))
```

(aber das ist m.E. eher schwieriger zu verstehen).

(20.1)

```
fmtStr = ("%(kat)s wurde von %(anteil)2.1f%% der "
  "Befragten als %(urteil)s eingeschätzt")
```

(String-Literale, die nur durch Leerzeichen und ähnliches voneinander getrennt sind, fügt Python automatisch zu einem einzigen String zusammen. `"drei""undzwan""zig"` ist aus Sicht von Python also genau das gleiche wie `"dreiundzwanzig"`. Die Klammern und den String habe ich gesetzt, weil ich die Zeilen ansonsten mit einem Backslash hätte verbinden müssen, während wir so wegen Pythons offener-Klammer-Regel automatisch eine verbundene Zeile bekommen.)

(20.3)

Die Wortentsprechungen in einer Liste zu halten hat allenfalls den Vorteil etwas geringeren Speicherverbrauchs. Dem steht gegenüber, dass ihr lediglich positive Zahlen übersetzen könntet. Wenn ihr weiter einen Teil der Zahlennamen berechnen könntet, würde euch das Dictionary ermöglichen, eine Ausnahmeliste zu halten (die z.B. weiß, dass 11 keineswegs einzehn heißt, wie das aus dreizehn, vierzehn, fünfzehn usf. zu erwarten wäre).

Auf der nächsten Folie werden wir sehen, wie man um den KeyError bei unbekannten Zahlen herumkommt.

(20.4)

In der positiven Fassung ist das schwierig, weil ihr noch nicht wisst, was es alles an unveränderbaren Werten gibt in Python. Umgekehrt könnt ihr probieren, was mit veränderbaren Werten passiert, denn ihr kennt sowohl Listen als auch Dictionaries als veränderbare Werte.

Also:

```
>>> d = {}
>>> d[['a', 'b']] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

Die Fehlermeldung mag schon andeuten, dass es nicht wirklich und genau die immutability ist, die die Verwendbarkeit als Schlüssel ausmacht.

(20.5)

Es gibt 2 aus. Das Beispiel zeigt einerseits, dass auch Dictionary-Literale in einem Index-Ausdruck stehen können (und nicht nur Referenzen auf Dictionaries) – das aber ist eigentlich klar, denn in Ausdrücken stehen nun mal Werte, und ob sie direkt dort stehen oder lediglich eine Referenz auf sie, ist egal.

Interessanter ist, dass Funktionen auch nur Werte sind und die Funktionsnamen Referenzen auf diese Funktionen. Erst wenn der Aufruf-Operator (das sind die runden Klammern ()) auf so einen Wert angewendet wird, wird die Funktion tatsächlich aufgerufen. Was also in der letzten Zeile passiert, ist:

1. Definiere ein Dictionary mit Zeichen als Schlüsseln und Funktionen als Werten
2. Holen den zum Schlüssel `'a'` gehörenden Wert aus dem Dictionary
3. Wende den Aufruf-Operator auf diesen Wert an und übergebe das Argument 1

(21.2)

```
def tabulate(valList, fct):
  d = {}
  for w in valList:
    d[w] = fct(w)
  return d
```

(Weiterführend:) Mit den später zu behandelnden list comprehensions und dem Konstruktor für dicts geht das auch in einer Zeile:

```
def tabulate(valList, fct):
  return dict([(w, fct(w)) for w in valList])
```

(21.3)

Eine relativ kurze und nicht allzu konfuse Lösung sieht so aus:

```
def countBigram(w1, w2, dist):
  d2 = dist.setdefault(w1, {})
  d2[w2] = d2.get(w2)+1
```

Eine Lösung mit `has_key` und Konsorten ist aber natürlich auch ok.

(21.4)

Das *Literal* `{}` bezeichnet ein neues, leeres Dictionary. Demnach wird im zweiten Quelltext vier Mal ein neues, leeres Dictionary an l drangehängt.

Im ersten Beispiel hingegen wird zunächst eine Liste, die ein leeres Dictionary enthält, erzeugt und dann vervierfacht. Dies bedeutet, dass genau *dieses* Dictionary noch drei Mal an l angehängt wird.

Im Allgemeinen sind Konstruktionen, in wie hier denen vorfabrizierte Datenstrukturen stehen, von denen niemand weiß, ob sie jemals gebraucht werden, unpythonesk – von daher ist dieses zunächst vielleicht etwas seltsam anmutende Verhalten verschmerzbar. Wenn ihr wirklich mal einen ganzen Haufen veränderbarer Datenstrukturen (mit Listen wäre natürlich das Gleiche passiert) in einer Sequenz braucht, verwendet list comprehensions.

(22.1)

Im Groben hat das erste Import ein Dictionary compute.germanWords angelegt. Das `from compute import germanWords` hat uns eine weitere Referenz auf dieses Dictionary gegeben.

Beim Reload wird ein neues Dictionary angelegt, was man daran sieht, dass die id von compute.germanWords sich ändert. Reload hat aber überhaupt keine Veranlassung, an unserem Namespace rumzufuhrwerken, und deshalb verweist germanWords weiterhin auf das „alte“ compute.germanWords – ganz genau so läuft das auch bei Funktionen, weshalb bei from-importierten Funktionen ein Reload ins Leere zu laufen scheint.

Wenn euch das noch unklar ist, malt euch das ganze im Python-Universum auf – gegenüber unseren alten Bildern solltet ihr unten nur mehrere Zeilen mit Namen vorsehen: Eine für unseren Haupt-Namespace (nennt sie „__main__“) und eine für das compute-Modul.

(24.2)

Lange Zeit hat das Programm nur Eingaben bekommen, in denen Nichtterminalsymbole vorkamen, für die es auch Regeln in der Grammatik gab. Das Programm warf aber KeyErrors, als im Eingabewort plötzlich „unbekannte“ Nichtterminale vorkammen. Übrigens ist es keineswegs gesagt, dass Achers Programm schlechter ist als unseres – es weigert sich eben, eine „unerfüllbare“ Aufgabe auszuführen, nämlich ein nur aus Terminalsymbolen bestehendes Wort abzuleiten. Unser Programm hinkt weiter und gibt am Schluss ein Wort aus, in dem noch Nichtterminale stehen.

In realen Anwendungen ist es wichtig, sich Gedanken über das Verhalten des Programms in solchen „Grenzfällen“ zu machen: Fehlermeldung oder Weiterhinken? Es gibt keine allgemeine Regel, was günstiger ist, dies hängt von der Anwendung ab. Allerdings sollten Spezifikation und Dokumentation klar sagen, was jeweils passiert.

(25.1)

```
def fak(n):
  res = 1
  for i in range(1, n+1):
    res = res*i
  return res
```

(25.3)

Ihr solltet in beiden Fällen etwas wie einen RuntimeError mit der Nachricht, die maximale Rekursionstiefe sei überschritten, sehen. Das bedeutet nichts anderes, als dass Python eine (veränderbare) Grenze setzt, wie oft Funktionen andere Funktionen aufrufen können. Dass es so eine Grenze geben muss, ist klar, denn Python muss sich für jede aufgerufene Funktion einige Dinge merken (etwa, wo die Ausführung weitergeht, wenn die Funktion endet, oder auch die Werte der in der Funktion verwendeten Variablen). Die Grenze, an die ihr hier stoßt, ist allerdings (wahrscheinlich, es sei denn, ihr hättet einen ausgesprochen lausigen Computer) künstlich und wurde eingeführt, um wahrscheinlich inkorrekte Programme abzufangen, bevor sie den Rechner ernsthaft belasten.

(26.1)

```
def catUpper(fName):
  print open(fname).read().upper()
```

(26.3)

```
def filecopy(fName1, fName2):
  srcF = open(fName1)
  dstF = open(fName2, "w")
  while 1:
    ln = srcF.readline()
    if not ln:
      break
    dstF.write(ln)
  dstF.close()
  srcF.close()
```

Die denkbare Alternative

```
def filecopy(fName1, fName2):
  stuff = open(fName1).read()
  dstF = open(fName2, "w")
  dstF.write(stuff)
  dstF.close()
```

hat den Nachteil, exzessiven Hauptspeicherverbrauchs, dafür geht das auch mit Binärdateien (genau genommen geht die erste Version unter nicht ganz irren Betriebssystemen auch mit

Binärdateien, aber dann kann der Speicherverbrauch auch exzessiv sein). Recht universell ist schließlich

```
def filecopy(fName1, fName2):
  bufSz = 1024
  srcF = open(fName1, "rb")
  dstF = open(fName2, "wb")
  while 1:
    buffer = srcF.read(bufSz)
    if not buffer:
      break
    dstF.write(buffer)
  dstF.close()
  srcF.close()
```

– dabei ist das b ein unter vernünfigen Systemen ignorierter Hinweis für Windows, die Finger von unseren Daten zu lassen. Bei Textdateien ist der nicht wichtig (und kann sogar schaden), bei Binärdateien ist er (unter Windows) lebenswichtig.

(27.1)

```
>>> return = 0
  File "<stdin>", line 1
    return = 0
           ^
SyntaxError: invalid syntax
>>> open("gnubbel.plo")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: 'gnubbel.plo'
>>> {}["a"]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 'a'
>>> int("a")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): a
```

Das Fangen wird der/dem LeserIn zur Übung überlassen...

(27.2)

```
try:
  doSomething()
except KeyboardError:
  print "Röchel"
```

Mit dem Exception-Mechanismus kann man Control-Cs nicht gänzlich ignorieren, weil auf jeden Fall ein Exception-Handler anläuft und damit der normale Kontrollfluss unterbrochen ist. Mit etwas Gebastel könnte man natürlich den Zustand zum Zeitpunkt des Control-C wiederherstellen – aber letztlich will man das wahrscheinlich nicht. Abbruchwünschen des Benutzers sollte man, ggf. nach Aufräumarbeiten, in der Regel nachkommen. Wer sowas wirklich mal braucht, kann sich im signal-Modul von Python umsehen.

(28.1)

```
>>> d = {tuple(range(2)): 'treffer'}
>>> d[(0,1)]
'treffer'
```

– es geht also.

Dass man keine veränderbaren Werte als Dictionaryschlüssel haben möchte, liegt daran, dass die Semantik von sowas sehr kitzlig wäre. In einem hypothetischen Python, das Listen als Schlüssel zuließe, könne man folgendes schreiben:

```
>>> l = [1,2]
>>> k = [1,2]
>>> d[l] = 2   # falsch!!!
>>> l.append(3)
>>> d[[1,2]]
# Soll *das* einen KeyError geben oder 2?
>>> d[l]
# oder sollte das lieber einen KeyError geben?
>>> d[k]
# und dies hier?
```

(29.1)

Der einfachste Ansatz ist

```
>>> def computeWeirdPairs(limit):
...     return [(a, b) for a in range(1,limit) for b in range(1,limit)
...             if not a%b]
...
>>> computeWeirdPairs(10)
[(1, 1), (2, 1), (2, 2), (3, 1), (3, 3), (4, 1), (4, 2),
(4, 4), (5, 1), (5, 5), (6, 1), (6, 2), (6, 3), (6, 6),
(7, 1), (7, 7), (8, 1), (8, 2), (8, 4), (8, 8), (9, 1),
(9, 3), (9, 9)]
```

Allerdings ist das ein quadratischer Algorithmus, d.h. er braucht bei tausendfacher Datenmenge eine Million Mal länger. Fällt euch eine schnellere Lösung ein?

(30.1)

```
>>> import gener1
gener1
>>> import gener1
>>> reload(gener1)
gener1
<module 'gener1' from 'gener1.pyc'>
```

– das bedeutet, dass ein zweiter import nichts mehr tut. Python merkt sich, welche module bereits importiert sind. Das gilt auch, wenn der import aus einem anderen Modul gekommen wäre; in dem Fall wäre dann lediglich der Name `gener1` neu im Namespace des Moduls aufgetaucht. Ein reload hingegen führt „globalen“ Code durchaus aus.

(31.1)

```
  def getName(self):
    return self.myName
```

Achtet darauf, dass die Definition auch innerhalb der Klassendefinition ist. Warum Programme, die die Tiny-Klasse verwenden, (in der Regel) nicht einfach `t1.myName` schreiben sollten, werden wir auf einer der nächsten Folie diskutieren.

(31.2)

Im Konstruktor braucht man jetzt natürlich ein `self.hunger = 0`.

Die Methoden `eat` und `wait` könnten folgendermaßen implementiert werden:

```
  def eat(self):
    self.hunger = self.hunger-1

  def wait(self):
    self.hunger = self.hunger+1
```

Ein alternatives Design würde das Schrauben am Hunger zentralisieren. So ein Design würde sich lohnen, wenn die Überschreitung bestimmter Grenzen des Hungers zwingend bestimmte Folgen hätte, die wir dann nur an einer Stelle warten müssten:

```
  def _changeHunger(self, change=0):
    self.hunger = self.hunger+change
```

```
    # Add checks for hunger limits here

  def eat(self): self._changeHunger(-1)
  def wait(self): self._changeHunger(1)
```

– häufig sind viele kleine Methoden ein Zeichen guten Designs, aber nicht jedes Zerhacken eines Problems in kleine Funktionen ist auch gleich gutes Design. Gute „Faktorisierung“ ist nach wie vor kaum theoretisch zu begründen (obwohl angesichts der Wichtigkeit des Themas viel darüber nachgedacht wurde) und immer noch ganz wesentlich eine Erfahrungssache.

Auch in diesem Fall ist nicht klar, welche Fassung „besser“ ist, oder ob am Schluss eine dritte Möglichkeit (z.B. eine Methode `_notifyHungerChanged`, die immer aufgerufen werden muss, wenn jemand am Hunger dreht) besser geht. Letztlich hängt diese Entscheidung auch davon ab, wie das Plüschtier auf die verschiedenen Hungerzustände reagieren soll.

(33.2)

Mir fällt keine wirkliche Anwendung ein – diese Sorte von Record ist häufig langsamer, nie aber viel schneller als Dictionaries (weil hinter ihrer Implementierung auch Dictionaries stecken), geht nur mit Python-Identifiern (`,,a b‘‘` als Schlüssel geht nicht) und ist nur von eingeschränkter Schönheit.

(Weiterführend:) Mit etwas mehr Kunst könnte man allerdings dafür Sorgen, dass nur ganz bestimmte Attribute setzbar sind und evtl. auch noch Zugriffskontrolle auf diese Attribute realisieren (vgl. insbesondere die schon erwähnten Deskriptoren). Wenn man sowas macht, mögen Konstrukte dieser Art interessant werden.

(34.1)

```
class Toy:
  def __init__(self, name):
    self.name = name

  def getName(self):
    return name

class BeepingToy(Toy):
  def beep(self):
    print "%s beeps"%self.name

class GrowlingToy(Toy):
  def growl(self):
    print "%s growls"%self.name
```

Die Funktion zum Umtaufen (sinnvollerweise `setName`) muss natürlich in die `Toy`-Klasse, denn sie spiegelt eine allgemeine Eigenschaft aller Spielzeuge wieder. Sollen Spielzeuge auch kaputt gehen können, braucht Toy eine weitere Instanzvariable (sagen wir `broken`) samt Akzessorfunktion. Beachtet, dass damit auch gleich BeepingToy und GrowlingToy die Fähigkeit zum Umgetauftwerden und Kaputtgehen bekommen.

Will man allerdings dafür sorgen, dass kaputte Spielzeuge auch nicht mehr beepen und growlen, müssen die entsprechenden Methoden in den abgeleiteten Klassen geändert werden. Könnt ihr euch ein Verfahren ausdenken, bei dem das nicht mehr nötig ist?

(34.2)

Die naheliegende Lösung (nämlich einfach die Methoden wie gewohnt definieren) sollte einfach „out of the box“ funktionieren. Tatsächlich geht das aber eleganter – das hat nicht direkt was mit Polymorphismus zu tun, sondern nur mit cleverer Kombination der Gedanken von Namespaces und dem Umstand, dass Funktionen in Python first-class („auch nur Objekte“ sind). Ihr hättet nämlich einfach schreiben können:

```
class BeepingToy(Toy):
  def beep(self):
    print "%s beeps"%self.name
```

```
  play = beep
```

Mit diesem Code wird `play` (als so genannte Klassenvariable) einfach ein weiterer Name im Namespace von `BeepingToys`, der ganz platt auf die Funktion verweist, auf die schon `beep` verwiesen hat.

(35.2)

`len` und Freunde geben überall etwas wie `<type 'builtin_function_or_method'>` oder `<type 'function'>` zurück.

Wenn ihr das für `str` und Freunde probiert und ein halbwegs aktuelles Python habt, kommt allerdings etwas wie `<type 'type'>` heraus – ich habe also, wie oben eingestanden, tatsächlich gelogen, als ich behauptet habe, str sei eine Funktion. Zu meiner Verteidigung kann ich vorbringen, dass ihr (a) mit der Auskunft, str sei im Gegensatz zu len ein Konstruktor, obwohl es doch eigentlich genau gleich funktioniert, nicht so glücklich gewesen wärt und (b) das bis vor kurzer Zeit noch die Wahrheit war:

```
Python 2.0 (#12, Dec 11 2000, 15:36:11)
[GCC 2.95.2 19991024 (release)] on linux2
Type "copyright", "credits" or "license" for more information.
>>> type(str)
<type 'builtin_function_or_method'>
```

(Weiterführend:) Dass auch open vom Typ `type` ist, also letztlich ein Konstruktor ist, hat historische Gründe. Wie oben gesagt, heißt open jetzt offiziell file – und in der Tat referenzieren beide Namen das selbe Objekt:

```
>>> id(open)
135341760
>>> id(file)
135341760
```

(35.3)

Was immer ihr gemacht habt: Generell sollte man so weit wie möglich vermeiden, Typprüfungen zu machen. Teil der objektorientierten Philosophie ist es, sich mehr dafür zu interessieren, was ein Objekt *kann* und nicht so sehr, was es *ist*. Pythonesk ist es also, einfach mal eine Methode auf ein Objekt anzuwenden. Wenn das Objekt nicht kann, was man von ihm will, bekommt man einen AttributeError (oder, wenn man Funktionen wie `len` und Freunde verwendet, auch mal einen ValueError oder einen TypeError) und kann dann immer noch Massnahmen zur Fehlerbehebung einleiten.

Der Vorteil dieses Verfahrens ist, dass eure Funktionen dann garantiert auf alle Objekte gehen, auf die sie sinnvoll andwendbar sind.

Um euch eine Idee zu geben, wie das laufen könnte:

```
def _addIntable(someInt, intableObj):
  return someInt+int(intableObj)

def _addFloatable(someInt, floatableObj):
  return someInt+float(floatableObj)

def _addSeq(someInt, objWithLength):
  return someInt+len(objWithLength)

def _addOther(someInt, someObj):
  return someInt+id(someObj)

def addTo(someInt, someObj):
  for fun in [_addIntable, _addFloatable, _addSeq, _addOther]:
    try:
      return fun(someInt, someObj)
```

```
    except (AttributeError, ValueError, TypeError):
      pass
  raise Exception("Yikes: Can't add %s"%someObj)
```

(37.1)

```
def prepend(targList, srcList):
  targList[:0] = srcList
```

Macht euch klar, wo der Unterschied zur Funktion

```
def noPrepend(list1, list2):
  return list2+list1
```

liegt.

(38.1)

1. `[a-zA-Z_][a-zA-Z_0-9]*`
2. `[A-Z][a-zA-Z_0-9]* *=`
3. `class[^:]:` – es wäre auch denkbar, etwas wie `class +[a-zA-Z_][a-zA-Z_0-9]* *\( *(([a-zA-Z_][a-zA-Z_0-9]* *, *)* [a-zA-Z_][a-zA-Z_0-9]* *\))? *:` (ich hoffe, ich habe das jetzt richtig hingekriegt – im Klartext heißt das: class, dann ein Name, dann evtl. Leerraum und möglicherweise in Klammern eine Liste von Namen (den Elterklassen), die durch Kommata und evtl. Blanks getrennt sind, und schließlich ein Doppelpunkt). Es ist klar, dass man sowas nicht mehr so machen möchte. Braucht man wirklich so komplizierte REs, hilft die verbose („wortreiche“) Syntax von Python-REs (ist bei Module Contents[1] der Python-RE-Doku erklärt) oder auch manchmal das maschinelle Zusammenbauen der REs: `"class +*(\( *[a-zA-Z_][a-zA-Z_0-9]*"}` ist schon mal etwas besser.
4. `def[^:]:`
5. `= *[0-9]+ *$`

(38.2)

Für diesen Zweck vielleicht etwas zu aufgeblasen, aber trotzdem:

```
import sys, re

def firstMatch(regExp, text):
  """returns the first match of the python regular expression
  regExp in text or None if no such match exists.
  >>> firstMatch("ab*c", "acabcb")
  "ac"
  >>> firstMatch("p?ro", "acabcb")
  """
  matOb = re.search(regExp, text)
  if matOb:
    retVal = matOb.group(0)
  else:
    retVal = None
  return retVal

if __name__=="__main__":
  if len(sys.argv)<3:
    sys.stderr.write("Usage: firstmatch.py <re> <fname>\n")
    sys.exit(1)
  try:
    txt = open(sys.argv[2]).read()
  except IOError, msg:
    sys.stderr.write("Couldn't open %s: %s\n"%(sys.argv[2],
      str(msg)))
```

[1] http://docs.cl.uni-heidelberg.de/python/lib/node105.html

```
    sys.exit(1)
  mat = firstMatch(sys.argv[1], txt)
  if mat is not None:
    print mat
```

Demgegenüber wäre auch die lean-and-mean-Fassung denkbar, je nach Einsatzziel (dieses hier ist mehr Einwegsoftware – auf die gibts zwar noch kein Pfand, aber schön ist das trotzdem nicht):

```
import sys, re
m = re.search(sys.argv[1], open(sys.argv[2]).read())
if m is not None:
  print m.group(0)
```

(weiterführend:) – oder, als Äquivalent des 400 PS-Motors (völlig nutzlos und gefährlich, aber man kann zumindest glauben, damit significant others zu beeindrucken – das hier basiert auf short circuit evaluation, die erst später kommt):

```
import sys, re
sys.stdout.write([(a is not None and a.group(0)+"\n") or ""
  for a in [re.search(sys.argv[1], open(sys.argv[2]).read())]][0])
```

(40.4)

Nein zu (1). Setzt man den die greedy RE `<em>.*</em>` auf den Beispielstring an, wird zwar das äußere Element korrekt erkannt, nicht aber das innere (das den Teil mit „betont“ mitnimmt). Die stingy RE `<em>.*?</em>` hingegen kriegt zwar das inneren Element korrekt hin, nicht aber das äußere, bei dem dann der Teil mit „betont“ fehlt.

Nein zu (2). XML und Freunde sind kontextfrei (SGML genau genommen sogar mehr als das), und Parser dafür hackt man nicht mit Regulären Ausdrücken (es sei denn, eine lexikalische Analyse reicht schon, oder man wills gar nicht so genau wissen). Erfreulicherweise hat Python auch schon Parser für SGML (und damit HTML – nicht toll, aber für die meisten Zwecke brauchbar) und XML (sowohl SAX als auch DOM) an Bord. Nutzt sie, wenn ihr Markup-Sprachen parsen müsst.

(41.1)

```
>>> map(lambda n: -n, range(10))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Natürlich hättet ihr fast den gleichen Effekt auch mit `range(-9,1)` erreichen können – aber mit map und eigenen Funktionen könnt ihr weit komplexere Operationen auf alle Elemente einer Sequenz ausführen.

(41.2)

Karl hat doch schon wieder Klammern hinter einen Funktionsnamen gemacht, obwohl er die Funktion selbst an map übergeben wollte und nicht den Wert, den sie zurückgibt. Er hätte das hier machen sollen:

```
>>> map(swapPair, d.items())
[(1, 'a'), (2, 'b')]
```

Wenn ihr noch nicht versteht, was da passiert, probiert das hier:

```
>>> print swapPair
>>> print swapPair((1,2))
```

(41.3)

Ein: Startwert, Grenze, Schrittweite

Sei *Wert* gleich der Startwert, *Ergebnis* die leere Liste

Solange Wert kleiner Grenze ist

    Hänge Wert an Ergebnis an

    Inkrementiere Wert um Schrittweite

Interessante Grenzfälle z.B.: Startwert $>$ Grenze, Startwert $==$ Grenze, Schrittweite $>$ Grenze $-$ Startwert, negative Schrittweite usf.

Übrigens gibt es eine ganze Disziplin, die durch Beobachtung des Verhaltens von Software auf ihre Spezifikation zu Schließen versucht. Dieses so genannte reverse engineering ist in den letzten

Jahren von Industrieseite stark unter Beschuss geraten und in den USA unter dem DMCA in den meisten Fällen strafbar. Hättet ihr diese Aufgabe in Florida gelöst, wärt ihr jetzt schon im Gefängnis...

(41.4)

Der wichtigste Grenzfall ist sicher die leere Liste, und da versagt die Funktion auch gleich – reduce hat, ein Blick auf den Pseudocode verrät das, mit einer leeren Sequenz keinen Wert. Das Python-reduce allerdings erlaubt die Angabe eines dritten Arguments, das der Sequenz in jedem Fall vorangestellt wird. Hier sollte das wohl "" sein.

Ein weiterer Grenzfall ist die einelementige Sequenz, denn bei ihr muss gar nicht gejoint werden. Ein weiterer Blick auf den Pseudocode von reduce erklärt, warum das kein Problem ist.

Von allen anderen möglichen Grenzfällen (leeres sep, falsche Typen usf.) isoliert uns Python in dem Sinne, dass es einfach von Natur aus „das Richtige“ tut.

(41.5)

```
def fak(n):
  return reduce(operator.mul, range(1, n+1), 1)
```

Ganz richtig ist das natürlich nicht – eine robuste Implementierung sollte wohl alles, was nicht natürliche Zahl ist, zurückweisen. Diese Fassung gibt als Fakultät von -1 auch 1 zurück, was sozusagen in jeder Hinsicht falsch ist (will heißen: Die Fakultät ist für -1 gar nicht definiert, es ist also schon schlecht, überhaupt etwas zurückzugeben. Aber selbst wenn man die Fakultät so erweitert, dass sie für mehr als nur für natürliche Zahlen geht – das ist die Gamma-Funktion –, ist ihr Wert bei -1 nicht 1, sondern, na ja, richtig viel: Die Gamma-Funktion hat bei -1 einen Pol).

(41.6)

```
>>> def makeCheckBeginChar(begChar):
...     return lambda s: s.startswith(begChar)
...
```

Der Kommentar oben zur Frage der Python-Versionen gilt hier auch.

(44.1)

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

internalError = 0
fileNotFoundError = 1
invalidOperationError = 2
syntaxInInputError = 3

englishTable = [
"Internal error",
"File not found",
"Invalid operation",
"Syntax error in input file",
]

germanTable = [
"Interner Fehler",
"Datei nicht gefunden",
"Ungültige Operation",
"Syntaxfehler in Eingabedatei",
]

def reportError(errCode=-1, errTable=englishTable, fatal=0):
  if errCode==-1:
    msg = "No error specified"
```

```
  else:
    msg = errTable[errCode]
  sys.stderr.write("%s: %s\n"%(sys.argv[0], msg))
  if fatal:
    sys.exit(1)

if __name__=="__main__":
  reportError()
  reportError(internalError)
  reportError(internalError, errTable=germanTable)
  reportError(invalidOperationError, fatal=1)
```

Der Zugriff auf `sys.argv[0]` gibt den Namen des aktuell laufenden Programms aus – es ist zumindest unter Unix guter Ton, das bei Fehlermeldungen zu tun.

Die ist übrigens kein besonders schlauer Weg, um zu Internationalisieren (d.h. etwa Fehlermeldungen in Landessprachen auszugeben), schon weil das viel zu wenig automatiert ist. Wenn ihr sowas machen wollt, solltet ihr euch GNU gettext ansehen – die Idee dabei ist, den jeweiligen String im Quellcode als Schlüssel in eine Datenbank von Übersetzungen zu verwenden.

(46.1)

```
aList = range(10)
listLen = len(aList)
i = 0
while i<listLen:
  print aList[i]
  i = i+1
```

Eine weniger schöne Alternative (die aber das, was in for-Schleifen hinter den Kulissen passiert, ganz gut abbildet):

```
aList = range(10)
i = 0
try:
  while 1:
    print aList[i]
    i = i+1
except IndexError:
  pass
```

(51.2)

Ja, aber das habt ihr in diesem Kurs nicht gelernt. Tatsächlich kann hinter einem `raise` ziemlich viel stehen, traditionell insbesondere Strings. Solange nur `Exceptions`, Unterklassen davon oder deren Instanzen geraiset werden, sind die Ausdrücke äquivalent. Und das sollte der Fall sein, solange ihr nicht uralte Module von anderen Leuten verwendet oder aber selbst irgendeinen Mist `raiset`. Und davon solltet ihr die Finger lassen, nicht nur, weil `except` nach `ids` fängt und wir oben schon gesehen haben, dass etwa gleichlautende Strings verschiedene `ids` haben können.

(51.3)

Gut, wir sind hier in der Short Circuit-Hölle. Zunächst wird das Programm für beliebige nichtnumerische Argumente abstürzen. Ansonsten spielen nur die ersten beiden Kommandozeilenargumente eine Rolle, und wesentlich verschiedenes verhalten ist auch nur zu erwarten, wenn sich die Argumente logisch unterscheiden, also sagen wir zwischen Nullen und Einsen. Gehen wir die möglichen Fälle durch:

Wenn das Programm gar kein Argument bekommt, scheitert schon der erste Vergleich, womit der Wert des ganzen ersten Teils und damit, weil alles nur mit and verbunden ist, der Wert des ganzen logischen Ausdrucks feststeht: Er ist falsch, es wird also nichts ausgegeben.

Bekommt das Programm eine Null als Argument, wird der zweite Teil des ersten Teilausdrucks ausgewertet, es wird also eine Null ausgegeben. Diese Null macht aber auch den ersten Teilausdruck falsch, so dass der Rest der Bedingung wieder nicht ausgewertet wird.

Übergeben wir hingegen eine eins, ist der erste Teilausdruck wahr und der zweite wird ausgewertet (denn er könnte den ganzen and-Ausdruck noch falsch machen). Das ist in dem Fall auch so, aber weil sys.argv in diesem Fall nicht länger als 2 ist, wird der zweite Teil des or-Ausdrucks auch noch ausgewertet. Er druckt eine Null, gibt diese auch zurück, und falsch oder falsch ist falsch, womit der Wert des gesamten if-Ausdrucks wieder feststeht und der dritte Teilausdruck nicht mehr ausgewertet werden muss.

Bleiben die Fälle mit zwei Argumenten: 0 0 macht den ersten Teilausdruck wegen dessen zweiten Teils falsch, weswegen nichts anderes ausgewertet wird, es wird also nur eine 0 ausgegeben. Der Fall 0 1 ist analog.

Im Fall von 1 0 gibt der erste Teilausdruck eine eins aus und ist wahr, so dass der zweite Teilausdruck ausgewertet wird. Die Bedingung an die Länge von sys.argv ist dieses Mal wahr, so dass der zweite Teil des zweiten Teilausdrucks nicht ausgewertet wird, aber zur Beurteilung des Gesamtausdrucks auch noch der dritte Teilausdruck angesehen werden muss. Er gibt 0 aus und falsch zurück, insgesamt wird also 1 0 ausgegeben.

Übergeben wir 1 1, läuft alles analog, nur ist mit dem dritten Teilausdruck jetzt der gesamte logische Ausdruck wahr. Mithin wird, weil `not int(sys.argv[2])` falsch ist, der zweite Teil des or-Ausdrucks im print-Statement ausgewertet und folglich noch eine Null ausgegeben. Die Gesamtausgabe ist also 1 1 0.

Ihr ahnt schon, man will Short Circuit Evaluation eigentlich nicht in nichttrivialer Weie mit Funktionen mit Seiteneffekten kombinieren. Solange es keine Seiteneffekte gibt, macht sie aber Programme schneller, ohne dass irgendwas passiert.

(51.4)

```
x+(y**2)
(a.attr)+(-5)
f((a+b)) is not (a<b)
lambda a, b: ((not a) or ((b.attr)[4]))
```

Wenn euch diese Aufgabe verwirrt hat, liegt das eben daran, dass Python das alles so macht, wie man es erwarten würde.

Die Sache mit dem lambda ist übrigens etwas blöd, weil lambda so ein komischer Operator ist – er ist zwar binär (seine Argumente sind die Argumentliste und der Ausdruck), aber ein Teil (das Lambda) steht vor dem ersten Argument, ein weiterer Teil nach einem Infix-:. Es wäre vielleicht schöner gewesen, wenn die Designer von Python eine Syntax wie `a, b lambda not a or b.attr[4]` vorgesehen hätten. Aber das hätte mehr Probleme gegeben und wäre von jeder Warte außer der des Ausdrückeschreibens auch hässlicher gewesen.

(54.2)

hon. Bei `BinaryNode` müssen wir vor allem aufpassen, dass wir nicht über `None` iterieren – `None` versteht nicht das Iterator-Protokoll (warum wohl?), und wir würden einen AttributeError kriegen, wenn wir die Iteration probieren würden.

```
  def __iter__(self):
    yield self
    for node in [self.left, self.right]:
      if node is not None:
        for child in node:
          yield child
```

(54.5)

```
import itertools

def enumerate(iterable):
  return itertools.izip(itertools.count(), iterable)
```

– richtig lustig wird das erst, wenn ihr Laufzeit und Speicherverbrauch für große iterables messt.

(56.3)

Eine mögliche Implementation wäre:

```
class KeyValTree(BinaryTree):
  """This class encapsulates a tree of key/value pairs.

  It's somewhat unfortunate that what we call value in the BinaryNode
  now is the key of the key value pair.  We compensate with a bit
  of name shuffling magic.

  Additional magic in here is that you can create an "empty" node that
  is filled in as soon as it should receive a key/value.  This is mainly
  useful for the root node.
  >>> d = KeyValTree()
  >>> d["bla"] = "foo"
  >>> d["drei"] = 3
  >>> d["aaa"] = 9
  >>> d["bla"]
  'foo'
  >>> d["bla"] = 10
  >>> d["bla"]
  10
  >>> print d
    9
  10
    3
  """
  def __init__(self, key=None, value=None):
    BinaryTree.__init__(self, key)
    self.nodeValue = value

  getKey = BinaryTree.getValue

  def getValue(self):
    return self.nodeValue

  def setValue(self, value):
    self.nodeValue = value

  def __cmp__(self, other):
    try:
      return cmp(self.getKey(), other.getKey())
    except AttributeError:
      return -1

  def insert(self, key, value):
    """inserts newNode into the tree in a position such that
    the property of being sorted is maintained.

    Unfortunately, we need to replicate some code from
    SortedTree -- this may be a hint that we need to refactor,
    now that we have the additional functionality.  It would
    be concivable to let SortedTree.insert take a Node as
    well (it would then have to replace the node in case of
    a perfect match to make things work here).
    """
    if self.getKey() is None:
      self.value = key
      self.setValue(value)
```

```
    if key<self.getKey():
      setMeth, getMeth = self.setLeft, self.getLeft
    elif key==self.getKey():
      self.setValue(value)
      return
    else:
      setMeth, getMeth = self.setRight, self.getRight
    if not getMeth() is None:
      self.getMeth().insert(key, value)
    else:
      setMeth(KeyValTree(key, value))

  __setitem__ = insert

  def retrieve(self, key):
    """returns a the value for key if present, raises KeyError otherwise
    """
    try:
      if self.getKey()>key:
        return self.getLeft().retrieve(key)
      elif key==self.getKey():
        return self.getValue()
      else:
        return self.getRight().retrieve(key)
    except AttributeError:
      raise KeyError(key)

  __getitem__ = retrieve
  iterkeys = BinaryTree.__iter__

  def itervalues(self):
    yield self.getValue()
    for child in [self.getLeft(), self.getRight()]:
      if child is not None:
        for val in child.itervalues():
          yield val
```

Beachtet, dass sowohl `insert` als auch `retrieve` eigentlich nichts mehr machen müssen, nachdem sie ihren rekursiven Aufruf abgesetzt haben (in insert könnte man das noch etwas deutlicher machen, in retrieve stört die Exception-Behandlung etwas, aber das soll hier nicht kümmern). Solche Funktionen heißen endrekursiv und können vom System potenziell erheblich effizienter bearbeitet werden als rekursive Funktionen ohne diese Eigenschaft. Mehr dazu in Programmieren II – Python ist das ohnehin egal.

Wenn ihr Lust habt, implementiert doch einfach noch weitere Methoden, die ihr von Dictionaries kennt: keys, values, items (prima zum Traversieren...), has_key, iterkeys usf.

(58.1)

Eine denkbare Implementation sieht so aus:

```
class Stack:
  def __init__(self):
    self.elements = []

  def __len__(self):
    return len(elements)

  def push(self, element):
    self.elements.append(element)
```

```
  def pop(self):
    return self.elements.pop(element)

  def peek(self):
    return self.elements[-1]
```

Besonders schlaue Menschen könnten versucht sein, sich durch Erben von list das Leben einfacher zu machen:

```
class CheatStack(list):
  def push(self, element):
    self.append(element)

  def peek(self):
    return self.elements[-1]
```

Dies täte zwar alles, was wir wollen, aber leider noch viel mehr (es erlaubt nämlich alle Zugriffe auf den „Stack“, die wir auch auf Listen machen können). Wirklich schlimm ist das nicht, solange alle NutzerInnen der Klasse wissen, was sie tun. Häufig schadet es aber nicht, Operationen, die nicht gehen sollen, gar nicht erst anzubieten. Die erste Klasse tut dies ohnehin, bei `CheatStack` könnten wir alle Methoden, die wir verbieten wollen, überschreiben (und dann Exceptions auslösen) oder gleich aus dem Objekt löschen (`del self.append` geht in der Tat).

(58.2)

Trickreich ist hier vor allem, das Ignorieren der character-Events hinzukriegen, damit nicht Daten aus ignorierten Elementen in unsere Ausgabe „reinbluten“.

In erster Näherung ist das zu erreichen, wenn man den Konstruktor so anfängt:

```
  def __init__(self, interestingEls, scrwid=79, *args):
    ContentHandler.__init__(self, *args)
    self.ie = interestingEls
    self.scrwid = scrwid
    self.indentSize = 2
```

und dann in start- und endElement die Ausgabe jeweils durch

```
    if name in interestingEls:
```

konditioniert. Um zu unterdrücken, dass Daten aus ignorierten Elementen „herausbluten“, muss dann in characters noch auf

```
    if self.elementStack[-1] in interestingEls:
```

bedingt werden (hier sieht man ein Beispiel, in dem eine peek-Methode eines „echten“ Stacks praktisch ist).

(58.3)

Mein Vorschlag wäre, die startElement-Methode folgendermaßen abzuändern:

```
  def startElement(self, name, attrs):
    self._emitChars()
    fmtAtts = " ".join(['%s="%s"'%it for it in attrs.items()])
    if fmtAtts:
      fmtAtts = " "+fmtAtts
    self._printIndented("<%s%s>"%(name, fmtAtts))
    self.elementStack.append(name)
```