Einführung in Python

Frank Krafft, Jens Stephan und Martin Schimmels

16. April 2013

Fehler, Verbesserungen, Ergänzungen etc. an martin@mschimmels.de

Widmung

- Allen, die sich für freie Software einsetzen und freie Software schreiben.
- Allen, die ihr Wissen nicht für sich behalten wollen, sondern es mit anderen teilen wollen. Wissen ist ein Gut, das sich vermehrt, wenn man es teilt.

  „Wenn Informationen allgemein nützlich sind, wird die Menschheit durch ihre Verbreitung reicher, ganz egal, wer sie weitergibt und wer sie erhält.“ [1]
- Allen, die verstanden haben, dass freie Software nicht ein Hirngespinst ist, sondern wirtschaftliches und politisches Denken beeinflusst: der Gedanke, der dahinter steckt, kann Politik und Wirtschaft menschlicher machen.
- Kris Kristofferson (Freedom is just another word for nothing left to lose)

[1] Richard Stallman, Datum und Ort unbekannt

Inhaltsverzeichnis

I. Grundlagen 23

1. Was ist Python? 25
1.1. Der Name . . . 25
1.2. Was für eine Sprache? . . . 25
1.3. Warum Python und nicht eine andere Sprache? . . . 26
1.4. Ausführung von Python-Programmen . . . 27

2. Programmieren 29
2.1. Was heißt „Programmieren“? . . . 29
2.2. Was ist also ein Programm? . . . 30
2.3. Geschichte . . . 30
2.4. Was braucht man, um mit Python zu arbeiten? . . . 30
2.5. Hilfsprogramme . . . 31
2.6. Voraussetzungen . . . 32
2.6.1. Das Dateisystem . . . 32
2.6.2. Regeln für Dateinamen . . . 33
2.6.3. Datei-Operationen . . . 33
2.7. Was heißt „Programmieren“? Fortsetzung! . . . 34
2.7.1. Strukturierte Programmierung: Ein Überblick . . . 34
2.8. Programmierstil und Konventionen . . . 36
2.9. Reservierte Wörter . . . 38
2.10. Fehler (zum ersten) . . . 38
2.11. Fehler finden . . . 39
2.12. Aufgaben . . . 39

II. IDE 41

3. Die Entwicklungsumgebung IDLE: Zahlen und Variable 43
3.1. . . . und bevor es losgeht 43
3.2. Rechnen in der Entwicklungsumgebung . . . 43
3.3. Ein Speicherplatz mit einem Namen: Variable . . . 49
3.3.1. Variable in Mathematik und in der Programmierung . . . 49
3.3.2. Variablennamen . . . 49
3.3.3. Wertzuweisung . . . 50
3.3.4. Übungen zu Variablen . . . 51
3.3.5. Zuweisungsmuster . . . 51
3.4. Rechnen und vergleichen . . . 52
3.4.1. Komplexe Zahlen . . . 54
3.4.2. Andere Zahlsysteme . . . 55
3.5. Shortcut-Operatoren . . . 55

4. Richtig programmieren 57
4.1. Entwicklungsumgebungen . . . 57
4.1.1. Die Entwicklungsumgebung Eric . . . 57
4.1.2. IDE unter Windows . . . 57
4.1.3. So sieht’s aus . . . 58
4.1.4. Der Programm-Rahmen . . . 58
4.2. Das absolute Muss: Hallo Ihr alle . . . 59
4.3. Die ersten einfachen Programme . . . 60

III. Texte und andere Daten 61

5. Texte 63
5.1. Grundlegendes zu Texten . . . 63
5.2. Operationen auf Texten . . . 65
5.3. Kodierungen . . . 68
5.4. Formatierung von Zeichenketten . . . 68
5.5. Reguläre Ausdrücke . . . 69
5.5.1. Allgemeines zu regulären Ausdrücken . . . 70
5.5.2. Beispiel für Zahlen . . . 71
5.5.3. Ein komplizierterer Text . . . 71

6. Strukturierte Daten 73
6.1. Überblick . . . 73
6.2. Listen . . . 73
6.2.1. Definition von Listen und Listenelemente . . . 73
6.2.2. Veränderung von Listen . . . 75
6.2.3. Tricks mit Listen . . . 75
6.2.4. Kopie einer Liste . . . 77
6.3. Dictionaries . . . 78
6.4. Tupel . . . 79
6.5. Zusammenfassung . . . 79
6.6. Übungen zu Listen, Dictionaries … . . . 79

IV. Strukturen 81

7. Programmstrukturen 83
7.1. Gute Programme, schlechte Programme . . . 83
7.2. Eins nach dem anderen: Die Sequenz . . . 83
7.2.1. Aufgaben zu Sequenz . . . 83
7.3. Wenn … dann: Die Alternative . . . 84
7.3.1. Blöcke und Einrückungen . . . 84
7.3.2. Beispiel für eine Mehrfach-Entscheidung . . . 85
7.3.3. Ein Trick für verschachtelte Entscheidungen . . . 86
7.3.4. Übungen zu Auswahl . . . 88
7.4. Schleifen . . . 88
7.4.1. Zähl-Schleifen . . . 89
7.4.2. While-Schleifen . . . 92
7.4.3. Verschiedene Probleme und ihre Lösung mittels Schleifen . . . 93
7.4.3.1. Die Anzahl der Elemente einer Liste bestimmen . . . 93
7.4.3.2. Anzahl der Elemente einer Liste mit einer bestimmten Eigenschaft . . . 94
7.4.3.3. … wie eben, aber mit Ergebnisliste . . . 94
7.4.3.4. Das größte Element einer Menge finden . . . 94
7.4.3.5. An welcher Stelle steht das größte Element einer Menge? . . . 95
7.4.4. Übungen zu Schleifen . . . 96
7.5. Funktionen . . . 98
7.5.1. Allgemeines zu Funktionen . . . 98
7.5.2. Definition und Aufruf von Funktionen . . . 98
7.5.2.1. Einfache Funktionen . . . 98
7.5.2.2. Rückgabewerte und Parameterlisten . . . 99
7.5.2.3. Gültigkeitsbereiche und Namensräume . . . 102
7.5.3. Funktionen wiederverwenden . . . 102
7.5.4. Globale Variable . . . 103
7.5.5. Rekursive Funktionen . . . 105
7.5.6. Funktionen als Parameter von Funktionen . . . 106
7.5.7. Übungen zu Funktionen . . . 108

V. Modularisierung 109

8. Module 111
8.1. Mehr Mathematik . . . 111
8.2. Eigene Module . . . 114

VI. Objektorientierte Programmierung 117

9. Klassen 119
9.1. Ist schon klasse, so eine Klasse! . . . 119
9.2. Objekte, Objekte, Objekte . . . 123
9.3. Kapselung . . . 126
9.4. Vererbung (ohne Erbschaftssteuer) . . . 128

10. Arbeitsblätter zu Klassen (Danke, Jens!) 131
10.1. UML-Diagramme und Python-Code . . . 131
10.2. Vererbung . . . 132
10.3. Objektvariable vs. Klassenvariable . . . 133
10.4. Polymorphie . . . 134
10.5. Übungsaufgabe . . . 135
10.6. Assoziationen Teil 1 . . . 136
10.7. Klassenarbeit . . . 137
10.8. Assoziationen Teil 2 . . . 138
10.9. Liste von Objekten . . . 139

11. Ein etwas längeres Programm 141
11.1. Objektorientierter Entwurf . . . 141
11.2. Objektorientierte Programmierung . . . 142
11.2.1. Die Klasse `Konto` . . . 142
11.2.2. Das aufrufende Programm . . . 142
11.2.3. Realisierung der Methoden „Einzahlen“ und „Auszahlen“ . . . 143
11.2.4. Die Verzinsung . . . 144
11.2.5. Die ersten Verbesserungen . . . 145
11.2.6. Übungen zu Klassen . . . 146

VII. Fehler und Ausnahmen 147

12. Fehler... 149
12.1. .. macht jeder . . . 149

VIII. Permanente Speicherung 153

13. Dateizugriff 155
13.1. Dateien allgemein . . . 155
13.2. Fortführung des Konten-Programms . . . 157
13.3. In die Datei damit!!! . . . 159
13.4. Übungen zu Dateien . . . 160

14. Datenbanken 161
14.1. MySQL . . . 161

IX. Noch ein paar Beispiele zur OOP 165

15. Ein weiteres Projekt: ein Getränkeautomat 167
15.1. Objektorientierter Entwurf und Realisierung der Klasse . . . 167

15.2. Die Waren kommen in den Automaten . . . 170
15.3. Geld regiert die Welt . . . 170
15.4. Jetzt kann gekauft werden! . . . 171
15.5. To Do!! . . . 172

16. Noch ein Beispiel: Zimmerbuchung in einem Hotel 173
16.1. Vorstellung des Projekts . . . 173
16.2. Der erste Entwurf: Eine Klasse für Hotelzimmer . . . 173
16.3. Der zweite Entwurf: ein aufrufendes Programm . . . 175
16.4. Der dritte Entwurf: das Hotel . . . 175
16.5. Der vierte Entwurf: Methoden werden ergänzt . . . 178

17. Und noch ein Beispiel: ein Adressbuch 181
17.1. Vorstellung des Projekts . . . 181
17.2. Der erste Entwurf: eine Liste von Listen . . . 181
17.3. Der zweite Entwurf: ein fauler Trick . . . 182
17.4. Der dritte Entwurf: eine Liste von Dictionaries . . . 183
17.5. Der vierte Entwurf: Aktionen (aber nur angedeutet)!! . . . 183
17.6. Der fünfte Entwurf: Aktionen (jetzt tut sich wirklich etwas)!! . . . 185
17.7. Der sechste Entwurf: persistente Speicherung . . . 188
17.8. Jetzt wird es objektorientiert . . . 189
17.8.1. Der Klassenentwurf . . . 189
17.8.2. Zur Klasse `Adresse` kommt die Klasse `Adressbuch` hinzu . . . 191
17.8.3. Dauerhafte Speicherung . . . 193

18. Immer noch das Adressbuch — nur schöner 197
18.1. Das Adressbuch kommt in die Datenbank . . . 197

19. Eine Ampel 203
19.1. Der Entwurf . . . 203
19.2. Die Realisierung . . . 203
19.3. Persistente Speicherung . . . 204
19.4. Eine Kreuzung hat 4 Ampeln! Der Entwurf. . . . 205
19.5. Die Realisierung . . . 207

X. Grafik! Internet! 209

20. CGI-Programme 211
20.1. HTML und Kollegen . . . 211
20.2. Allgemeines zu HTML . . . 211
20.2.1. Was ist HTML? . . . 211
20.2.2. Grundlagen . . . 211
20.2.3. Obligatorische HTML-Befehle . . . 211
20.2.4. Struktur eines HTML-Dokuments . . . 212
20.2.5. Absätze und Leerzeichen . . . 212
20.2.6. Hervorhebungen . . . 212
20.2.7. Listen . . . 213
20.2.8. Links . . . 213
20.3. CGI (Das Common Gateway Interface) . . . 213
20.4. Die Pflicht: hallo, ihr alle da draußen . . . 214
20.5. Hello world als dynamische Web-Seite . . . 214
20.6. Ein eigener Mini-Webserver . . . 215
20.7. Dynamik durch Schleifen . . . 216
20.7.1. Vorwärtszählen 216
20.7.2. .. und rückwärtszählen . . . 217
20.7.3. Das kleine 1 x 1 . . . 218
20.8. Formular-Eingabe und Auswertung . . . 221
20.9. Eine Rechnung . . . 224

20.10Jetzt aber wirkliche Dynamik 231

21. Eine Kreuzung auf meiner Web-Seite **235**
21.1. Die Ampel kommt ins Netz 235
21.2. Die Kreuzung im Netz ist auch nicht schwerer 237

22. Programme mit grafischer Oberfläche **245**
22.1. Tkinter 245
22.2. Problem: Temperaturen umrechnen (zuerst ohne grafische Oberfläche) 245
22.3. . . . und jetzt mit grafischer Oberfläche 247

XI. Python für Verwaltungsaufgaben 251

23. Python als Sprache für den eigenen Rechner **253**
23.1. Das Betriebssystem 253

XII. Glossar 255

Glossar **260**

A. Lösungen zu Aufgaben **261**
A.1. aus Kapitel 7: Programmstrukturen 261
A.2. aus Kapitel 8: Schleifen 261
A.3. aus Kapitel 10: Funktionen 267

B. GNU Free Documentation License **269**

C. GNU Free Documentation License **275**

D. Literatur zu Python **283**

E. Index **285**

Abbildungsverzeichnis

1. Was ist Python?
1.1. John Cleese (fast jugendfrei) . . . 25
1.2. Interpreter und Compiler . . . 27
1.3. Compiler . . . 28
1.4. Interpreter . . . 28

3. Die Entwicklungsumgebung IDLE: Zahlen und Variable
3.1. Eric Idle (pfeift Always look on the bright side of life) . . . 43
3.2. Die IDE Idle . . . 44
3.3. Erste Berechnung in der IDE Idle . . . 45
3.4. Rechnungen in Idle . . . 45
3.5. Division von ganzen Zahlen . . . 46

4. Richtig programmieren
4.1. Die IDE Eric . . . 57
4.2. Die IDE PyScripter . . . 58

5. Texte
5.1. Texte in der IDE Idle . . . 64
5.2. Multiplikation von Texten . . . 65

9. Klassen
9.1. Klassendiagramm `Mensch` . . . 122
9.2. Klassendiagramm `Schueler` . . . 128

15. Ein weiteres Projekt: ein Getränkeautomat
15.1. Klassendiagramm `Automat` . . . 167
15.2. Klassendiagramm `Ware` . . . 170

16. Noch ein Beispiel: Zimmerbuchung in einem Hotel
16.1. Klassendigramm `Hotelzimmer` . . . 173
16.2. Klassendiagramm `Hotel` . . . 176

17. Und noch ein Beispiel: ein Adressbuch
17.1. Klassendiagramm `Adresse` . . . 189

19. Eine Ampel
19.1. Klassendiagramm `Ampel` . . . 203
19.2. Eine Kreuzung . . . 206
19.3. Kreuzung: Freie Fahrt von Nord nach Süd . . . 206
19.4. Kreuzung: Alle warten . . . 206
19.5. Kreuzung: Freie Fahrt von West nach Ost . . . 207
19.6. Kreuzung: Wieder warten alle . . . 207

20. CGI-Programme
20.1. Eingabe-Formular in HTML(Namen) . . . 222
20.2. Eingabe-Formular für eine Rechnung in HTML . . . 224
20.3. Ausgabe einer Rechnung in HTML (aber wirklich nicht schön) . . . 228
20.4. Ausgabe einer Rechnung in HTML (so kann man das akzeptieren) . . . 231

22. Programme mit grafischer Oberfläche

22.1. Klassendiagramm `Temperatur` 245
22.2. Klassendefinition: GUI für die Temperatur 247
22.3. Temperaturprogramm 249
22.4. Beendigung des Temperaturprogramms 250

Tabellenverzeichnis

2. Programmieren
2.1. Betriebssystem-Befehle . . . 33
2.2. Reservierte Wörter in Python: . . . 38

3. Die Entwicklungsumgebung IDLE: Zahlen und Variable
3.1. Arithmetische Operatoren in Python . . . 48
3.2. Variablennamen . . . 50
3.3. Vergleichsoperatoren . . . 53
3.4. Verschiebeoperatoren . . . 54
3.5. Verschiedene Zahlsysteme und Umrechnung . . . 55
3.6. Shortcut-Operatoren . . . 56

5. Texte
5.1. Steuerzeichen . . . 65
5.2. Zeichenkette als Feld betrachtet . . . 66
5.3. Format-Codes . . . 69
5.4. Format-Option-Codes . . . 69
5.5. Reguläre Ausdrücke: Zeichen . . . 70
5.6. Reguläre Ausdrücke: Methoden . . . 71

6. Strukturierte Daten
6.1. Überblick Strukturierte Daten . . . 79

7. Programmstrukturen
7.1. Schlüsselwort . . . 97
7.2. Verschlüsselung . . . 98

9. Klassen
9.1. Sichtbarkeit und Kapselung . . . 127

12. Fehler...
12.1. Fehler-Konstanten . . . 152

19. Eine Ampel
19.1. Die Ampel-Phasen . . . 207

Vorwort

Typographische Konventionen

- Schreibmaschinenschrift wird für Code(-Fragmente) benutzt. Ebenso werden `Variablennamen` in Schreibmaschinenschrift gesetzt.
- **Befehlsnamen** werden fett gesetzt.
- `Dateinamen` werden Schreibmaschinenschrift gesetzt.
- Ein- und Ausgaben in IDLE werden wie folgt gesetzt. Eingaben sind an den >>> zu erkennen.

```
>>> print 'Hallo'
Hallo
```

-

> TIPP
>
> 
>
> Randnotizen stehen im Online-Dokument in einem blauen Kasten. Randnotizen stehen im gedruckten Text in einem Rahmen mit einem eingekreisten großen „i“ am Rand.

-

> WICHTIG
>
> 
>
> Wenn etwas besonders wichtig ist, steht das im Online-Dokument in einem orange-roten Kasten. Wenn etwas besonders wichtig ist, steht das im gedruckten Text in einem Rahmen mit einem Ausrufezeichen am Rand.

-

> ANMERKUNG
>
> 
>
> Eine Anmerkung steht im Online-Dokument in einem grünen Kasten. Auf eine Anmerkung wird im gedruckten Text durch den ausgestreckten Zeigefinger hingewiesen.

-

> ACHTUNG
>
> 
>
> Das Achtung-Zeichen weist auf eine Situation hin, bei der leicht ein Fehler gemacht werden kann. Das wird im Online-Dokument durch einen violetten Kasten hervorgehoben.

Freiheit

Dieser gesamte Text ist frei im Sinne der „Freien Software“. Er steht unter der „Free Documentation Licence“, die im Anhang nachzulesen ist. Das bedeutet, dass dieser Text beliebig kopiert und verbreitet werden darf. Er darf auch modifiziert werden. Einige der Rechte, so wie sie in der „Free Documentation Licence“ FDL erläutert werden, werden hier anschließend aufgeführt.

Das Urheberrecht dieses Textes bleibt bei mir. Die wichtigste Bestimmung ist, dass dieser Text immer unter der FDL bleibt, auch wenn er von Dir verändert wird und dass die FDL immer Teil dieses Textes sein muss.

Du darfst beliebig viele Kopien dieses Textes auf beliebigen Medien machen. Auf jeder Kopie muss aber ein urheberrechtlicher Vermerk aufgeführt sein und die FDL muss Teil der Kopie sein.

Den eigentlichen Text darfst Du nach Deinen Vorstellungen bearbeiten, Teile hinzufügen, Verbesserungen anbringen, Teile weglassen, sofern Du einige Regeln einhältst

1. Der veränderte Text muß deutlich sichtbare Vermerke enthalten, die die Veränderung, den Autor der Veränderung und das Datum der Veränderung angeben.
2. Der veränderte Text verbleibt unter der FDL und enthält die FDL.
3. Der Text darf nicht in ein proprietäres Format umgewandelt werden.
4. Eine Kopie des veränderten Textes wird mir zugesandt.

Dieser Text ist in DocBook-XML geschrieben. Dabei habe ich mich an die Spezifikationen gehalten, wie sie in dem Buch von [DocBook-XML] beschrieben wird. Er wird normalerweise in drei Formaten verteilt:

1. als Menge von XML-Dateien
2. als Menge von HTML-Dateien, die aus XML generiert wurden (mit fast keinen Modifikationen)
3. als eine pdf-Datei, die mit dem pdflatex-Befehl erstellt wurde, nachdem einige Veränderungen angebracht wurden.

Da ich selber es absolut unerträglich finde, wenn Fachbücher der Informatik in schlechtem Deutsch geschrieben sind,[2] bitte ich auch alle Leser, dass sie mir eventuelle sprachliche Fehler mitteilen, gerade auch wenn es nur Leichtsinnsfehler oder kleinere Ungenauigkeiten sind.[3]

Gleichberechtigung

In diesem ganzen Text wird immer für einen generischen Begriff nur das in der deutschen Sprache übliche grammatikalische Geschlecht benutzt. Eine Fachkraft oder sogar eine Koryphäe auf dem Gebiet der Python-Programmierung kann also durchaus auch mal ein Mann sein und ein Python-Programmierer auch eine Frau. Vor allem bei der Bezeichnung von mehreren Personen, vulgo Plural genannt, vermeide ich so unschöne Formulierungen wie „Python-Programmiererinnen und Python-Programmierer“. Sollte jemand daran Anstoß nehmen, so steht jedem Leser es frei, den Quelltext mit einem (zum Beispiel in Python selbstgeschriebenen) Programm und mit den in diesem Text erworbenen Kenntnissen und einigem Zusatzwissen in XML (hierzu bietet sich das Buch von Drake [Python and XML] an) so abzuändern, dass solche generischen Begriffe in eine nach seinen Vorstellungen politisch korrekte Form umwandelt werden. [4]

Wer? Wo? Wie?

Dieses Buch kann man natürlich überall lesen, am Schreibtisch, auf dem Balkon, vor dem Fernsehgerät,[5] am Strand . . . . Bei vielen Kapiteln ist es aber ganz sinnvoll, dass man dieses Buch am Arbeitsplatz vor sich nimmt, Computer eingeschaltet und alles so vorbereitet, dass man sofort den einen oder anderen Python-Befehl eingeben kann oder eine kleine Übung machen kann, ein Programm schreibt, weil durch das Lesen die Idee zu einem Programm kommt . . .

[2] Es ist leider betrüblich zu sehen, wie viele Autoren geraden von Fachbüchern im Bereich der Datenverarbeitung ihre Muttersprache nicht mehr beherrschen. Zeichensetzung und Rechtschreibung folgen da oft Regeln, die niemand zuvor gehört hat — und der Autor kurz danach wieder vergessen hat. Manchmal fragt man sich da, ob der Autor schon jemals ein Programm in welcher Programmiersprache auch immer geschrieben hat, wenn man weiß, mit welcher Genauigkeit man dabei sprachliche Regeln einzuhalten hat

Besonders peinlich sollte das für das Lektorat eines Verlages sein, aber auch da hat man manchmal das Gefühl, dass dort Analphabeten eingestellt worden sind. Die typische Reaktion, wenn man einen solchen Lektor auf einen Fehler hinweist, ist dann ein „Haben Sie etwa noch nie einen Fehler gemacht?“

[3] Ich bitte wirklich ausdrücklich darum, mir Dinge NICHT durchgehen zu lassen, die heutzutage an Schulen viel zu vielen Schülern nachgesehen werden.

[4] Wenn es sein muss, sogar mit dem grossen Binnen-I. horribile dictu.

[5] Ja, es soll wirklich Fernsehsendungen geben, bei denen man nichts versäumt, wenn man wegschaut — und manchmal ist es trotzdem besser, man bleibt vor der Glotze hängen.

Eine Begründung

Ich höre und vergesse. Ich sehe und erinnere mich.
Ich tue und ich verstehe.

(Konfuzius [6])

Einfache Aufgaben sollten einfach gelöst werden. Einfach ist schön, einfach ist elegant. Auf dem Gebiet der Programmierung heißt das: das Zusammenklicken eines Programms ist vor allem für Anfänger abzulehnen, weil dadurch das Programm, das man schreiben will, undurchschaubar wird. Durch diese Methode der Software-Produktion werden Bibliotheken, Module, Klassen eingebunden, die die Logik des Problems verschleiern, die Platz kosten, die das Programm langsam machen, die ein Programm unleserlich machen. Die Folge davon ist: eine komplizierte Lösung erfordert komplexe Resourcen.

Das soll an einem alltäglichen Beispiel demonstriert werden. Ein Programm benötigt einen Text von der Standard-Eingabe. Der Begriff „Standard-Eingabe“ bedeutet, dass in der Umgebung, in der das Programm ausgeführt wird, über die Tastatur etwas eingegeben wird. Dies bedeutet insbesondere weiter, dass nicht etwa eine grafische Oberfläche geschaffen wird, in der sich vielleicht ein Fenster öffnet, in das man etwas eingeben kann. Damit man auch sieht, was man eingegeben hat, wird die Eingabe danach auch noch ausgedruckt. Das macht das Programm natürlich viel länger !!! Dazu folgen hier 3 Lösungen in 3 Programmiersprachen:

1. In perl gibt es die ganz kurze Version:

   ```
   $s = <> ;
   print $s;
   ```

   oder die fast schon geschwätzige Version:

   ```
   $s = <STDIN> ;
   print $s;
   ```

   Was fragt der typische Schüler, wenn er dieses Stück Programmcode sieht?

   - Warum steht da ein Dollar-Zeichen?
   - Was heißt „STDIN“?
   - Warum steht das „STDIN“ in größer-kleiner-Zeichen?
   - Muss da ein Strichpunkt stehen?

2. In Python geht das fast noch kürzer:

   ```
   print raw_input('Text eingeben: ')
   ```

   Das funktioniert, aber das ist nicht besonders schön, deswegen schreibt man in Python oft wesentlich mehr. In beiden Fällen erhält aber man auch wesentlich mehr Komfort, nämlich eine ausdrückliche Aufforderung, etwas einzugeben:

   ```
   s = raw_input('Text eingeben: ')
   print s
   ```

   Was fragt der typische Schüler, wenn er dieses Stück Programmcode sieht?

   - Was heißt „raw_input“?
   - Warum steht der Text in Anführungszeichen?

3. In Java sieht das Programm-Fragment so aus:

[6] ???

```
import java.io.*;
public class stdLesen {
  public static void main(String[] args)
  throws IOException{
    BufferedReader eingabe =
      new BufferedReader(new InputStreamReader(System.in));
    String s;
    while ((s = eingabe.readLine()) != null && s.length() != 0)
      System.out.println(s);
  }
}
```

Was fragt der typische Schüler, wenn er dieses Stück Programmcode sieht?

- Was bedeutet „java.io.*“?
- Was bedeutet importieren?
- Was bedeutet „public“?
- Was heißt „class“?
- Was bedeutet „static void main“?
- Was bedeutet „String[] args“?
- Was heißt „throws“?
- Was ist eine „IOException“?
- Was sollen die vielen geschweiften Klammern?
- usw.

Ich bin sicher, dass jeder Lehrer zustimmt, wenn ich sage, dass für fast jedes Wort und fast jedes Zeichen hier eine Frage nach dem Sinn auftauchen wird.

Hier sollte sich jeder Lehrende kurz überlegen, wie lange er bräuchte, um den jeweiligen Programm-Code seinen Schülern zu erläutern, so dass es jeder von ihnen versteht. (Eine kleine Entscheidungshilfe: In einem der Standardbücher zu [Java] steht ein ähnliches Programm-Fragment, das dieses Problem löst, auf Seite 612.)

Man könnte das Beispiel vereinfachen und das Standard-Programm, das jedem Anfänger in fast jeder Programmiersprache vorgeschlagen wird, seit Kernighan/Ritchie ihr Werk über die Sprache „C“ geschrieben haben, das Programm, das nichts tut außer den Text „Hallo, world!“ ausgeben, in verschiedenen Programmiersprachen untersuchen (dann müssen natürlich „C“ und „C++“ auch dabei sein!).

> Vor allem die Schüler sind es, die mit Recht eine Abkehr von den bisher benutzten Unterrichtssprachen verlangen, weil die damit fabrizierten Produkte in der Regel keinen Wiedererkennungswert mehr besitzen im Vergleich zu den sonst vorherrschenden Programmen.
>
> Es besteht allerdings die Gefahr - insbesondere beim Einsatz moderner Entwicklungsoberflächen und „Interface-Buildern“, wie sie z.B. bei Delphi üblicherweise eingesetzt werden - dass nun schöne bunte Programme entstehen, die Vermittlung wichtiger Informatik-Konzepte aber zu kurz kommt. [7]

Deswegen hat dieser Text ein Motto: **Keep it simple. Je einfacher, desto besser für die Lernenden.** Damit will ich verhindern, dass Tools eingesetzt werden, die für die professionelle Arbeit konzipiert sind, aber für den Unterricht im Programmieren überdimensioniert sind. Die Software, die wir einsetzen, soll das können, was man braucht, um Probleme des Kurses zu lösen, und sie soll es dem Lernenden ermöglichen, das möglichst einfach zu tun. Sie soll nicht ungezählte Fähigkeiten beherrschen, die man braucht, um ein großes Programmpaket in einem großen oder mittleren Unternehmen zu erstellen, die man aber nicht braucht, um programmieren zu lernen.

Was wir hier einsetzen, orientiert sich also nicht an dem, was Industrie und Wirtschaft produktiv benutzen. Für Lernende viel sinnvoller ist es, Software zu benutzen, die das Exemplarische hervorhebt und damit dem Lernenden ermöglicht, Gelerntes auf andere Umgebungen zu übertragen.

[7] aus: <http://www.b.shuttle.de/b/humboldt-os/python/>

> Die Programmierung graphischer Oberflächen erfolgt heute im kommerziellen Bereich in der Regel mit Hilfe sogenannter „Interface-Builder". Auch wenn es Sinn machen kann, solche Werkzeuge Schülern im Informatikunterricht an die Hand zu geben (Java-Workshop, Python-Glade, PythonWin), so sollte dies nicht gleich im Anfängerunterricht geschehen. Ein wichtiges Ziel des Unterrichts besteht darin, grundsätzliche Konzepte und Arbeitsweisen zu verstehen. Komfortable Oberflächen verstellen oft den Blick für das Wesentliche, machen u.U. oberflächlich. **Die Oberfläche ist nicht das System!**[8]

Weil „learning by doing" in der Programmierung nicht nur ein netter Spruch ist, sondern wirklich das einzig erfolgversprechende Rezept, sollte jeder Lernende möglichst viele Übungsaufgaben machen. Ein paar Aufgaben sind bereits am Ende mancher Kapitel aufgeführt. Im Sinne des oben Gesagten sollte jeder sich ermutigt fühlen, Aufgaben hinzuzufügen.

> Bereits beim Zusammenspiel zwischen Variablen und Schleifen zeigen sich die Probleme vieler SchülerInnen im Umgang mit Steuerstrukturen. (Einfaches Beispiel: Wie bestimmt man das Maximum aus fünf eingegebenen Zahlen?) [9] Ein grundsätzliches Dilemma des Unterrichts besteht nun darin, dass häufig eine große Lücke zwischen dem passiven Beherrschen (also dem Nachvollziehen eines von der Lehrperson erstellten Programms) und dem aktiven Beherrschen (also dem selbständigen Erkennen, wann und wie die jeweilige Steuerstruktur zur Problemlösung einzusetzen ist) klafft und dass diese Lücke nur sehr schwer durch Erklärungen durch die Lehrperson zu überbrücken ist. [10]

Und im Sinne dieses Zitates sollten sich Dozenten davor hüten, Lösungen zu Aufgaben an die Lernenden auszugeben. Jeder Programmieranfänger sollte das schöne Gefühl kennenlernen, ein Problem selbst gelöst zu haben. Und nur beim Durchdenken einer eigenen Lösung kann ein Neuling das Für und Wider eines Lösungsansatzes bewerten.

[8] aus: <http://www.b.shuttle.de/b/humboldt-os/python/>

[9] Übrigens in Python ein triviales Problem. Da in Python dynamische Parameterlisten für Funktionen zum Sprachumfang gehören, liefert `print max(3,4,12,67,34,8)` das korrekte Ergebnis. Es ist Python völlig egal, wieviele Parameter übergeben werden.

[10] aus: <http://de.wikibooks.org/wiki/Programmieren_leicht_gemacht_-_ad%C3%A4quate_Modelle_f%C3%BCr_den_Einsatz_im_Unterricht#Einleitung>

Teil I.

Grundlagen

1. Was ist Python?

Always look on the bright side of life

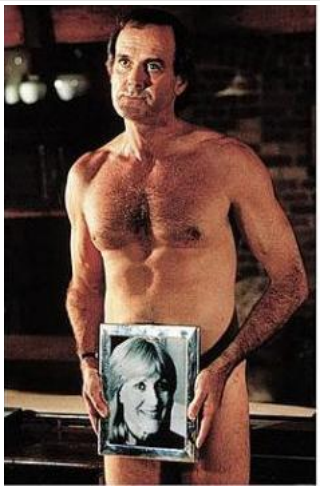
(Eric Idle [1])

1.1. Der Name

Python? Ein Tier? Eine Schlange? Nein, eine Sprache!! Eine Programmiersprache!!

Und der Name dieser Sprache hat in erster Linie nichts mit der Schlange zu tun, sondern mit **Monty Python's Flying Circus** „Das Leben des Brian" oder „Ein Fisch namens Wanda" kennt wahrscheinlich wirklich jeder, genauso wie wohl die meisten die bekanntesten Schauspieler dieser so britischen Komikertruppe kennen: John Cleese und Eric Idle.

Abbildung 1.1. John Cleese (fast jugendfrei)



Wie Mark Lutz in seinem Buch „Programming python"[2] schreibt: „You don't have to run out and rent *The Meaning of Life* or *The Holy Grail* to do useful work in Python, but it can't hurt."

Aus diesem Grund sollte sich niemand wundern, wenn typische Mustertexte in Python-Beispielen „Always look on the bright side of life" oder „And now for something completely different" lauten. Es fällt auf, dass Programmierer oft einen eigenen Sinn für Humor haben — besonders wenn sie Engländer sind und Monty Python im Kopf haben.

1.2. Was für eine Sprache?

Aber was für eine (Programmier-)Sprache ist denn Python? Python ist

- keine Skriptsprache, aber man kann damit Skripte schreiben[3]
- eine objektorientierte Sprache, aber man muss keine Objekte (explizit) benutzen
- keine Sprache, mit der man Programme mit grafischen Oberflächen schreibt; aber man kann das trotzdem sehr gut, weil es viele grafische Bibliotheken gibt

[1] Life of Brian

[2] [Programming Python], Seite 15

[3] Skripte bezeichnen Programme, mit denen man andere Programme aufruft, Betriebssystem-Aufrufe macht, Geräte anspricht, Dateien im Dateisystem manipuliert …

- eine der drei großen P-Sprachen: perl, php und eben Python. Python ist wirklich eine Sprache, mit der man dynamische Web-Seiten erstellen kann. Und viele machen das. Es existiert sogar einen Web-Applikations-Server, der in Python geschrieben ist und Python deswegen als interne Sprache benutzt: Zope.
- eine der am häufigsten benutzten Sprachen im WWW. Wenn man sich fragt, in welcher Sprache die Programme geschrieben sind, die weltweit am häufigsten benutzt werden, dann kommt man auf Python. Glaubst Du nicht? Nur zwei Unternehmen, die Python wesentlich benutzen:
  - Google
  - You Tube

  Überzeugt?

1.3. Warum Python und nicht eine andere Sprache?

Kann man denn wirklich mit einer Sprache, bei der der Erfinder an etwas wie „Always look on the bright side of life“ gedacht hat, etwas Sinnvolles machen, oder kommt da nur Blödsinn raus? Ach nein: „Der Sinn des Lebens“ .

Was sind also die Vorteile von Python als Programmiersprache?

1. Python-Code ist gut lesbar, oder wie Peter Walerowski sein Buch [4] beginnt: „Python-Programme kann man lesen!“ Damit sind Python-Programme gut nachvollziehbar, damit gut zu verändern und zu verbessern (Wartbarkeit).
2. Python-Code unterstützt die Wiederverwertbarkeit von Code durch Objektorientiertheit und die Aufteilung von Code in Module.
3. Python-Code ist plattformunabhängig. Ein auf einem Rechner geschriebenes Programm ist meistens ohne Veränderung auf einem anderen Rechner lauffähig.
4. Python bringt schon eine große Zahl von Bibliotheken mit, viele weitere können aus dem Internet besorgt werden.
5. Python-Code ist im Verhältnis zu Java-Code relativ kurz, im Verhältnis zu perl-Code sehr klar. Wie Lutz in [Programming Python], S. 4, schreibt, sind Python-Programme nur ein Fünftel bis ein Drittel so lang wie Programme in C++ oder Java. Schau einfach noch mal zurück zum Vergleich zwischen perl, Python und Java

   Eine schöne Seite, die eine Idee davon gibt, was die Unterschiede der Programmiersprachen sind, ist <http://www.programmieraufgaben.ch>. Unter den Aufgaben sind ein paar klassische Probleme in verschiedenen Sprachen gelöst. Es lohnt sich wirklich, hier zweimal hereinzuschauen. Schau zum ersten Mal jetzt rein, und schau Dir speziell Probleme an, bei denen auch eine Lösung in Python vorliegt. Vergleiche den Code (insbesondere die Länge des Codes) von Python mit dem in anderen Sprachen. Ich empfehle für einen ersten Blick die Lösungen zu „Prüfziffer auf Euro-Banknoten“. Ein zweites Mal kann man dann auf diese Seite schauen, wenn man einen ersten Eindruck von Python hat, denn dann sieht man, warum Lösungen in Python nicht nur besonders kurz, sondern vor allem besonders elegant sind.
6. Als Folge davon ist Python schnell bei der Entwicklung und Erstellung von Programmen. Die Kürze der Programme, die im vorigen Punkt angesprochen wurde, bewirkt, dass man oft keine Klimmzüge machen muss, um ein Problem zu lösen.
7. Python ist freie Software. Es ist in den meisten Linux-Distributionen schon enthalten, zu Windows frei aus dem Internet herunterzuladen. Das heißt, dass die (Weiter-)Entwicklung von Python nicht von den finanziellen Interessen eines Unternehmens abhängt, sondern von den Bedürfnissen der Nutzer, also der Programmierer. Oder anders gesagt: die Sprache bleibt so schön, wie sie ist, wenn WIR es wollen.
8. Python-Programme können wie ganz normale Programme ausgeführt werden; es gibt aber auch eine Test-/Entwicklungsumgebung, in der man Programm-Fragmente leicht und ohne großen Aufwand testen kann. Außerdem kann man Python-Code interaktiv auf der Shell / Kommandozeile schreiben.
9. Python hat eine gute Datenbank-Schnittstelle.

[4] [Python], Seite 13

10. Die meisten (alle?) Python-Distributionen haben schon eine grafische Bibliothek dabei, nämlich Tkinter. Die hat den Vorteil, dass sie freie Software ist (ich weiß, ich wiederhole mich). Außerdem ist Tcl/Tk selbständig lauffähig, das heißt, man kann für die grafische Oberfläche eine Art „Prototyping“ machen und dann das erzeugte Fenster relativ einfach nach Tkinter umwandeln.

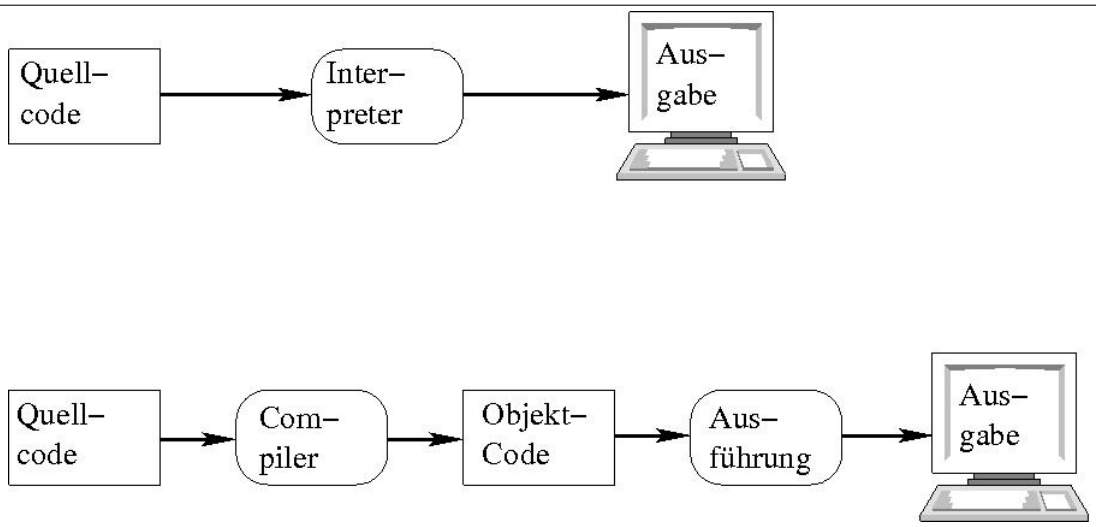
Gibt es auch Nachteile? Ja sicher! Aber die will ich hier nicht aufzählen, da ich ja von Python überzeugt bin. Als größte Schwäche von Python — aber da muss man schon wieder etwas über die Ausführung von Programmen in verschiedenen Sprachen wissen — muss die Ausführungsgeschwindigkeit genannt werden. Richtig. Nur wird das wahrscheinlich von uns (den Lesern und mir, dem Schreiber) nur ein ganz kleiner Prozentsatz merken, denn Programme, die wir schreiben, sind so „klein“, dass die Ausführungsgeschwindigkeit kein ernstzunehmendes Kriterium ist. Wichtiger ist wahrscheinlich für jeden von uns die gesamte Zeit, die wir in ein Programm stecken: die Zeit für den Entwurf, für die Umsetzung, also die eigentliche Programmierung, die Zeit für die Tests und dann letztendlich die Laufzeit des Programms. Dabei werden wir merken: Python ist schnell!!

1.4. Ausführung von Python-Programmen

Hier steht in der Überschrift der Begriff „Programm“. Was ein Programm ist, wurde bisher aber noch nicht erwähnt; und die Erklärung folgt auch erst im nächsten Kapitel.

Grundsätzlich unterscheidet man bei der Ausführung von Programmen zwei verschiedene Methoden: Programm-Code kann entweder interpretiert werden oder er wird kompiliert[5] und das Kompilat wird dann ausgeführt. Die unterschiedliche Verarbeitung wird in dem Bild dargestellt.

Abbildung 1.2. Interpreter und Compiler

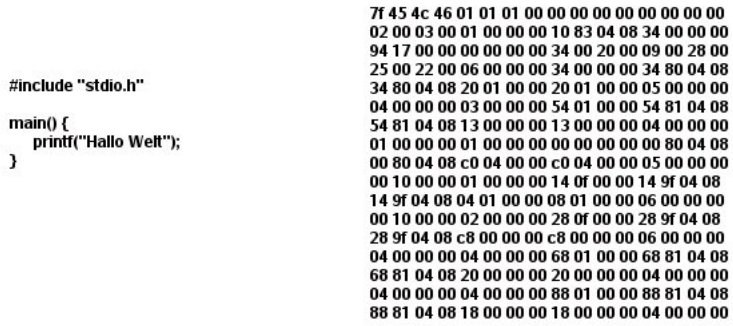


Zuerst soll in kurzen Worten die zweite Methode beschrieben werden.

Das, was ein Programmierer schreibt, wird als Programm-Quellcode bezeichnet. Dieser Quellcode ist in einer Programmiersprache geschrieben, die sich meistens vom Vokabular her an der englischen Sprache orientiert. Dieser Quellcode ist vom Computer nicht zu verstehen, sondern muss erst mittels eines eigenen Programms in Maschinensprache — das sind die langen Reihen von Nullen und Einsen — umgewandelt, auf englisch compiliert, werden.

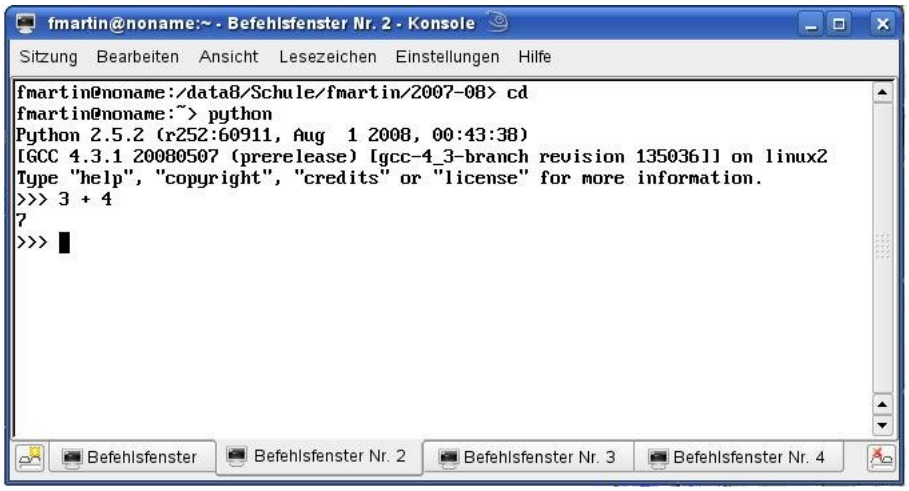
[5] Englisch wird dieses Wort und alle davon abgeleiteten natürlich mit „c“ geschrieben, im Deutschen mit „k“ aber im folgenden Text wird es in beiden Formen auftauchen. „Compiler“ sieht nur so, mit „C“, genießbar aus!

Abbildung 1.3. Compiler



Python wird nicht compiliert, sondern interpretiert. Das Programm, das Python-Quelltexte interpretiert, heißt auch *python* (allerdings mit kleinem „p“). Ein „Interpreter“ liest einen Quelltext zeilenweise und führt die Anweisung, die diese Zeile enthält, aus. Wenn Python auf dem Rechner installiert ist [6] , kann man diesen Interpreter sofort aufrufen.[7] Das machen wir jetzt! Nachdem der Interpreter sich mit dem Python-Prompt (>>>) arbeitsbereit gemeldet hat, geben wir dem Interpreter die erste Aufgabe:

Abbildung 1.4. Interpreter



Brav, der Interpreter hat die Zeile richtig gelesen und die Aufgabe zu unserer Zufriedenheit gelöst.

[6] das heißt auch, dass die Pfade alle richtig gesetzt sind

[7] Das gilt für Linux- und Mac-Systeme. Unter den Windows-Systemen muss unter Umständen zuerst die Position des Python-Interpreters in die Umgebungsvariablen eingetragen werden.

2. Programmieren

There are two ways of constructing a
software design:
One way is to make it so simple that there are
obviously no deficiencies,
and the other way is to make it so complicated
that there are no obvious deficiencies

(C. A. R. Hoare[1])

2.1. Was heißt „Programmieren“?

In grauer Vorzeit, also etwa vor 50 Jahren, wurden Computer hauptsächlich dazu benutzt zu rechnen (denkt heute noch irgendjemand daran, wenn er seinen **Rechner** einschaltet, dass der **rechnen** könnte oder sollte). [2] Heute, in den ersten Jahren des 21. Jahrhunderts, wird der Computer meistens dazu benützt, Zeichenketten, so die allgemeinste Bezeichnung für etwas, das sowohl Buchstaben als auch Ziffern und Sonderzeichen zum Inhalt hat, zu bearbeiten. [3]

Was macht ein Programmierer also? Er schreibt Programme. Na gut, das hilft auch nicht weiter. Oder doch? Ein Programm ist ein Algorithmus, der von einer Maschine abgearbeitet werden kann. Algorithmus? Nehmen wir doch dafür einfach mal die Beschreibung „Kochrezept“. [4] Wer schon einen Algorithmus formuliert hat, der weiß, dass die Umgangssprache oft nicht das geeignete Werkzeug dazu ist, da die Umgangssprache Mehrdeutigkeiten zulässt. Wenn man etwa das erste Kochrezept auf der in der Fußnote zitierten Seite betrachtet, dann wird auch dem in Küchendingen nicht so bewanderten Leser klar sein, dass es einen Unterschied macht, ob man „einen oder zwei zarte Elefanten“ nimmt. Also ist es angebracht, eine formale Sprache zu entwerfen, die Mehrdeutigkeiten vermeidet. Existiert eine solche formale Sprache, dann muss man noch einen Übersetzer schreiben, der den Algorithmus, geschrieben in dieser formalen Sprache, in die Maschinensprache des Rechners übersetzt.

Für diese Übersetzung gibt es vom Prinzip her 2 Möglichkeiten:

1. Das in einer Programmiersprache geschriebene Programm, der sogenannte Quelltext oder auf neudeutsch die Source, wird als ganzes in Maschinensprache umgewandelt. Dabei wird es auf Syntaxfehler untersucht; im Falle, dass Syntaxfehler auftreten, wird die Umwandlung abgebrochen und eine Fehlermeldung ausgegeben.

   Falls Programmcode aus anderen Quell-Dateien benutzt werden soll, wird dieser auch eingebunden. Das Ergebnis wird als Datei auf die Festplatte des Rechners geschrieben und liegt ab diesem Zeitpunkt in Maschinensprache vor. Beim erneuten Aufruf des Programms wird diese Datei benutzt.

   Dieser Übersetzer wird Compiler genannt.
2. Das Programm, das als Quellcode vorliegt, wird Befehl für Befehl in Maschinensprache übersetzt und ausgeführt. Das bedeutet insbesondere, dass auf die Festplatte des Rechners keine Datei mit einem ausführbaren Programm geschrieben wird. Bei jedem erneuten Aufruf des Programms wird der Quelltext wieder Zeile für Zeile abgearbeitet.

   Ein Programm, das diese Arbeit verrichtet, wird Interpreter genannt.

[1] zitiert nach: http://en.wikiquote.org/wiki/C._A._R._Hoare

[2] Das Beste, was ich zu diesem Thema gesehen habe, war ein (ernstgemeinter!! wirklich!!) Artikel mit Bild in einer Tageszeitung über einen Sachbearbeiter einer Behörde. Auf dem Bild waren zu sehen: vor ihm der Computer, in der Hand der Taschenrechner. Darunter als Bildunterschrift sinngemäß etwa, dass der Computer diesem guten Menschen viel Arbeit abnimmt, aber dass der gute Sachbearbeiter manche Sachen eben doch noch mit dem Taschenrechner berechnen muss. So weit zum Thema Volksverdummung.

[3] So wie ich das gerade auch mache (obwohl Programmieren oder Mathematik mir jetzt mehr Spaß machen würde).

[4] Beispiele für Kochrezepte findet man z.B. unter <http://f3.webmart.de/f.cfm?id=2959373&r=threadview&a=1&t=2769129>

2.2. Was ist also ein Programm?

Im vorigen Kapitel haben wir schon gesehen, dass Python interpretiert wird. Das bedeutet insbesondere, dass, sofern Python installiert ist und der Interpreter zur Verfügung steht, eine (korrekte) Anweisung in Python ausgeführt werden kann. Aber eine Anweisung ist noch kein Programm; na ja, jedenfalls kein richtiges Programm.

Ein Programm besteht nach der klassischen Definition aus einer Eingabe, einer Verarbeitung und einer Ausgabe. Man spricht deshalb vom EVA-Prinzip, wenn man die Anfangsbuchstaben der 3 Bereiche aneinanderreiht. Heutzutage erfolgt die Eingabe meistens über die Tastatur, die Verarbeitung ist das Ausführen eines Algorithmus, so wie er durch den Programmcode vorgegeben ist und das Ausgabemedium ist standardmäßig der Bildschirm.

2.3. Geschichte

Im Laufe der vergangenen 50 Jahre wurden viele solche formale Sprachen entworfen, meistens mit dem Ziel, Probleme aus einem bestimmten Anwendungsbereich zu lösen. Zu diesen Programmiersprachen wurden dann auch verschiedene Compiler entwickelt, verschiedene, weil es verschiedene Zielrechner und damit verschiedene Maschinensprachen gibt, aber auch, weil es manchmal an einem Compiler Verbesserungsmöglichkeiten gab, auch, weil handfeste wirtschaftliche Interessen dahinterstecken.

ANMERKUNG

Die Sprache COBOL wurde zum Beispiel entworfen, um vor allem administrative oder wirtschaftliche Probleme zu lösen, die Sprache Fortran, um technische, mathematische oder naturwissenschaftliche Probleme zu bearbeiten, bei denen es darauf ankam, große Mengen an Zahlen zu bearbeiten, die Sprache C entstand, weil man eine Sprache benötigte, die sehr maschinennah arbeitet aber trotzdem leicht zu programmieren ist. Dabei gilt, dass jede Programmiersprache (mit Einschränkungen) alles kann, aber nicht alles gleich gut. Die wenigsten Programmierer hätten Spaß daran, Mathematik-Programme in COBOL zu schreiben!!

Es gibt aber auch Programmiersprachen, die für eine breite Vielfalt von Problemstellungen geeignet sind, die genausogut für die Verarbeitung von Zahlen wie für die Verarbeitung von Texten genommen werden können. Welche Programmiersprache man lernt, und auch, welche Programmiersprache man später hauptsächlich benutzt, hängt von den persönlichen Vorlieben ab: denen des Lehrers und später den eigenen. Die erste Programmiersprache, die man lernt, sollte aber vor allem eines sein: leicht zu erlernen.

2.4. Was braucht man, um mit Python zu arbeiten?

Nun, man braucht zuerst einmal Python selber. Python ist eine Programmiersprache (ich weiß, ich wiederhole mich), aber als eine solche ist sie zweierlei: erstens ein Konzept, eine Idee. Irgendjemand, genauer gesagt Guido von Rossum, ist auf die Idee gekommen, dass es doch gut wäre, wenn es ein Programm gäbe, das bestimmte Anweisungen versteht und sie ausführen kann. Also hat Guido sich vorgenommen, einen Entwurf zu schreiben, in dem erst einmal steht, welche Vokabeln das Programm kennen soll und dann, welche grammatikalischen Regeln gelten sollen, um diese Wörter zu sinnvollen Sätzen zusammenzufügen. Das ist das Konzept.

Und zweitens ist Python natürlich ein Programm, damit eine Datei (genauer gesagt: ein ganzes Paket von Dateien), das eine Umsetzung von Guidos Konzept bildet.

Also muss sich jeder, der in Python programmieren will, diese Datei besorgen. Und da haben wir Glück: Python (als Konzept) und die Dateien, die Python auf einem Rechner ausführbar machen, sind freie Software. [5] „Freie Software“ hat etwas mit Freiheit zu tun, nicht mit dem Preis. Um das Konzept

[5] hierzu sollte man sich das gute Buch von Grassmuck [Freie Software] besorgen, das — das Thema erfordert es ja schon — auch

zu verstehen, ist an „frei“ wie in „freier Rede“[6], und nicht wie in „Freibier“ zu denken. [7]

ANMERKUNG

So ist Python unter jedem gebräuchlichen Betriebssystem verfügbar. Linux-Benutzer haben den Vorteil, dass Python zu fast jeder Distribution dazugehört und meistens bei Standard-Installationen mitinstalliert wird, weil einige Programme unter Linux eben Python benötigen.

Unter Microsoft-Betriebssystemen [8] muss Python gesondert besorgt werden.

Falls dieser Text auf CD zu Dir gekommen ist, ist die Chance groß, dass auch in einem Verzeichnis „Windows“ auf dieser CD alle Programme sind, die zur Arbeit mit Python nötig sind, zusammen mit einer README-Datei und einem Installationsprogramm (install.bat), das alles für den Betrieb von Python erledigt.

2.5. Hilfsprogramme

Als wichtigste Voraussetzung für das Programmieren wird ein Editor benötigt. Das ist **KEIN** Textverarbeitungsprogramm. Open Office und Word und wie die Textverarbeitungsprogramme alle heißen sind ungeeignet. Ein Editor soll keine Formatierungsanweisungen produzieren, sondern den Text genauso speichern, wie ich ihn eingebe.

Hingegen soll ein Editor die Strukturen einer Programmiersprache unterstützen. Die bei Windows mitgelieferten Editoren (Wordpad, Editor) sind aus diesem Grund ungeeignet. Unter Unterstützung einer Programmiersprache verstehe ich:

- Der Editor sollte bestimmte Konstrukte einer Sprache farblich hervorheben (englischer Fachausdruck: syntax highlighting)
- Der Editor sollte mehrere Dateien parallel geöffnet halten können. Du wirst es schnell merken, dass es Deine Arbeit ungemein erleichtert, wenn Du im Editor Dein letztes Werk neben der aktuellen Arbeit sehen kannst, denn meistens verwendest Du eine Technik, die beim letzten Programm erfolgreich war, gleich noch einmal.
- Ein Editor sollte das Ordnunghalten unterstützen und, das ist bei Python besonders wichtig, die vorgeschriebenen Ordnungskriterien beherrschen. Damit ist gemeint, dass der Editor automatisch die zusammengehörigen Dinge als zusammengehörig sichtbar macht.
- Der Editor sollte automatische Wortvervollständigung beherrschen. Damit ist gemeint, dass der Editor bei allen längeren Wörtern, die ich tippe, in seinem Gedächtnis kramt und einen sinnvollen Vorschlag macht, wie das Wort jetzt zu Ende geschrieben wird. Das wird auch jeder schätzen, wenn ein Programm einen gewissen Umfang erreicht. Und die Anwendung dieser Fähigkeit hilft bei der Reduzierung der Fehlerzahl. Wenn ich eine Variable „mindestAbnahme“ deklariert habe, dann hilft es Tippfehler zu vermeiden, wenn ich nach den ersten 3 Buchstaben „min“ schon den Vorschlag „destAbnahme“ bekomme. Wenn ich dieses Wort zehn Mal tippen müsste, dann würde ich sicher den einen oder anderen Fehler machen.

Es gibt viele Editoren, die diese Bedingungen erfüllen und Open Source Software sind. Die sind meistens von Programmierern für Programmierer gemacht — wirklich nicht die schlechteste Voraussetzung. Da möge jeder sich seinen Lieblingseditor heraussuchen. Der beste Editor ist immer der, mit dem ich selber gut auskomme. Einen Vorschlag mache ich trotzdem: SciTE (beachte die Groß-/Kleinschreibung) ist Open Source, und SciTE gibt es für Windows und für Unix/Linux.

Ein Programm in einer Skriptsprache wie Python wird anfangs in einer Shell (für Windows-Benutzer: auf der Kommandozeile) aufgerufen und macht seine Ausgaben genau dort. Deswegen sollte jeder die

fast „frei“ ist, nämlich bei der Bundeszentrale für politische Bildung für einen geringen Unkostenbeitrag zu erhalten ist. siehe: <www.bpb.de>

[6] [FSF]

[7] Hat eigentlich schon einmal jemand untersucht, inwiefern Freibier die Entwicklung freier Software fördert?

[8] , die ab jetzt einfach mit *Windows* bezeichnet werden, gleichgültig welcher Ausprägung (Windows XP, Vista etc.)

wichtigsten Shell-Befehle (für Windows-Benutzer: die guten, alten DOS-Befehle) kennen und auch auf der Shell sich im Verzeichnisbaum zurechtfinden.

Die nächste Stufe der Hilfsprogramme sind die „Integrierten Entwicklungsumgebungen“, kurz IDE (Integrated Development Environment). Hier existieren nebeneinander ein Editor, eine Shell, oft noch ein Debugger und öfters noch andere Anwendungen unter einem Dach. Für Python bietet sich in erster Linie „Eric“ an, allerdings muss auf Windows noch eine Qt-Bibliothek installiert werden. Eine Alternative auf Windows ist der PyScripter, der auch Open Source ist. Eher spartanisch, aber für die Belange eines Anfängers völlig ausreichend, ist die IDE „geany“, natürlich auch Open Source.

2.6. Voraussetzungen

2.6.1. Das Dateisystem

Als Vorkenntnisse sollte man einiges über die Organisation des Dateisystems auf dem eigenen Rechner wissen, nämlich was eine Datei, was ein Verzeichnis ist, wie man Verzeichnisse anlegt, Dateien kopiert, umbenennt, verschiebt. Hilfreich ist, wenn man das nicht nur mittels einer grafischen Oberfläche [9] kann, sondern auch als Betriebssystem-Befehl (also unter Linux auf einer *shell* und unter Windows auf der *Kommandozeile*). Aber das ist wahrscheinlich im 21. Jahrhundert zuviel verlangt. Mark Lutz schreibt in einer Fußnote seines Buches [Programming Python] Seite 266,

> In the first edition of the book *Learning Python*, for example, my coauthor and I directed readers to do things like „open a file in your favourite text editor“ and „start up a DOS command console“. We had no shortage of email from beginners wondering what in the world we meant.

Dem ist fast nichts hinzuzufügen. Oder doch? Eine Frage sei erlaubt: Warum richten sich Dozenten, Lehrer, Didaktiker nach dem, was große Konzerne propagieren, nämlich, dass man Computer bedienen kann, ohne zu verstehen, was man da tut und warum legen sie nicht mehr Wert darauf, dass Prinzipien des Computers verstanden werden, bevor man mit einer schönen grafischen Oberfläche herumspielt?

Also muss man sich (oder als Dozent: den Lernenden) zuerst einmal klarmachen, dass das, was man in einen Editor oder in eine Entwicklungsumgebung eingibt, erst dann „im Rechner“ ist, wenn man es in einer Datei gespeichert hat. Zum Glück bieten sowohl Editoren als auch Entwicklungsumgebungen hier Icons an, mit Hilfe derer man diese Operationen durchführen kann. Nach dem Anwählen dieser Icons öffnet sich in der Regel ein Fenster, in dem man

1. angeben kann, in welchem Verzeichnis
2. und unter welchem Dateinamen

die Datei gespeichert werden soll. Zum zweiten Punkt folgt gleich noch ein eigenes Unterkapitel. Hier soll allerdings schon auf die Eigenart eines Software-Konzerns von der Westküste der USA eingegangen werden. Die Entwickler, die für diesen Konzern ein „Betriebssystem“ entworfen haben, meinen seit einigen Jahren, dass sie wissen, was wir, die Benutzer wollen. So ist unter diesem Betriebssystem standardmäßig eingestellt, dass in allen Dateisystem-Betrachtern, also auch bei dem, der beim Speichern einer Datei aufgerufen wird, „bekannte Dateiendungen“ nicht angezeigt werden. Das führt oft zu unerwarteten Schwierigkeiten und ungeahnten Ergebnissen.

ACHTUNG

Wer unter Windows ernsthaft arbeiten will, sollte sich von Microsoft nicht bevormunden lassen und sofort den Schalter umlegen, um grundsätzlich alle Dateiendungen anzeigen zu lassen.

[9] Glück gehabt! Das [dummdeutsch]e Wort „Benutzeroberfläche“ konnte ich im Text gerade noch vermeiden. Der Benutzer bin ich, meine Oberfläche nenne ich Haut, und was die auf einem Computer-Bildschirm soll, habe ich noch nie verstehen können.

2.6.2. Regeln für Dateinamen

Es gibt einen großen Software-Konzern, der versucht, seinen Benutzern weiszumachen, dass Datei- und Verzeichnisnamen Sonderzeichen und Leerzeichen enthalten dürfen. Nun, über die Sonderzeichen könnte man noch diskutieren. Aber dann muss auch gewährleistet sein, dass jedes mögliche Betriebssystem in jeder möglichen Lokalisierung[10] auch diese Sonderzeichen anzeigt, und diese Fähigkeit auch an alle vom Betriebssystem aufgerufenen Programme weitergibt. Bei den Leerzeichen hört aber der Spaß auf. Ein Leerzeichen ist im normalen Text das Zeichen, das zwei Wörter voneinander trennt; es trennt zwei Informationseinheiten voneinander. Aus der Sicht des Betriebssystems ist aber ein Datei- bzw. Verzeichnisname **eine** Informationseinheit. Da hat ein Leerzeichen überhaupt nichts zu suchen.

ACHTUNG

Wer unter Windows die IDE „pyscripter“ benutzt, merkt es sehr schnell, dass er Leerzeichen in Dateinamen hat: das Programm hängt sich auf.

Das im vorigen Absatz Gesagte gilt auch für Verzeichnisnamen. Auch hier haben Sonderzeichen und Leerzeichen nichts zu suchen.

2.6.3. Datei-Operationen

Falls man die nötigen Befehle des Betriebssystems nicht beherrscht, sollte man sich ein Handbuch nehmen und nachschlagen, wie man unter seinem Betriebssystem

- eine Datei anlegt
- eine Datei löscht
- eine Datei kopiert
- eine Datei verschiebt / umbenennt
- ein Verzeichnis anlegt
- ein Verzeichnis löscht
- einen Link auf eine Datei anlegt (geht das überhaupt unter den Microsoft-Betriebssystemen?) [11]
- Zugriffsberechtigungen liest
- Zugriffsberechtigungen setzt

Ohne den Anspruch der Vollständigkeit gebe ich hier die Befehle, die für die oben aufgelisteten Operation benutzt werden, unter Unix-artigen Betriebssystemen und unter Windows in einer Tabelle wieder.

Tabelle 2.1. Betriebssystem-Befehle

Aktion	Unix/Linux	Windows
Datei anlegen	**touch dateiname**	
Datei löschen	**rm dateiname**	**del dateiname**
Datei kopieren	**cp alteDatei neueDatei**	**copy alteDatei neueDatei**
Datei verschieben / umbenennen	**mv alteDatei neueDatei**	**ren alteDatei neueDatei**
Verzeichnis anlegen	**mkdir verzeichnisname**	**md verzeichnisname**
Verzeichnis löschen	**rmdir verzeichnisname**	**rmdir verzeichnisname**
Link auf eine Datei anlegen	**ln originalName neuerName**	
Zugriffsberechtigung lesen	**ls -l dateiname**	**attrib dateiname**
Zugriffsberechtigung setzen	**chmod xxxx dateiname**	**attrib xxxx dateiname**

[10] also der Einstellung der Sprache der Darstellung

[11] Einen Link auf eine Datei anzulegen bedeutet, dass die Datei nur einmal existiert, aber unter verschiedenen Namen angesprochen werden kann. Das erscheint sinnlos? Ist es aber nicht. Mit „Name“ ist dabei natürlich der voll-qualifizierte Name gemeint, d.h. dass vor allem in einem Verzeichnis die „reale“ Datei existiert, in einem anderen Verzeichnis nur der Link, der Verweis auf die Originaldatei.

2.7. Was heißt „Programmieren“? Fortsetzung!

Programmieren heißt erst einmal, Befehle (Anweisungen, oft auch mit dem englischen Wort *statements* bezeichnet) zu schreiben, die der Computer mit Hilfe von Python (oder einer anderen Programmiersprache) versteht und ausführen kann. Aber warum soll der Computer das machen? Erstens: er kann vieles schneller als ich. Zweitens: ihm wird nicht so schnell langweilig wie mir. Drittens: auch nach Stunden wird er nicht müde und macht deswegen keine Leichtsinnsfehler.

2.7.1. Strukturierte Programmierung: Ein Überblick

Die langweiligste Art der Programmierung ist die, dass man einem Rechner die Aufforderung gibt, zehn oder hundert oder tausend verschiedene Befehle nacheinander auszuführen. So etwas wird eine Sequenz genannt. Das bedeutet, dass ich dem Rechner erst einmal diese vielen Befehle aufschreiben muss. Die meisten von uns würden das als völlig uneffektiv ablehnen, denn in der Zeit, in der wir die Befehle aufgeschrieben hätten, hätten wir das Problem schon „zu Fuß“ gelöst.

In der Frühzeit der Datenverarbeitung war das allerdings ein übliches Vorgehen. Es gab verhältnismäßig wenige Rechner und relativ viele Benutzer, es gab wenige Eingabegeräte und wenige Ausgabegeräte. Also habe ich meine Befehle aufgeschrieben, irgendwann in der Nacht hat ein freundlicher Mitarbeiter des Rechenzentrums meine Befehle dem Computer zum Fressen gegeben und am nächsten Morgen konnte ich das Ergebnis abholen.

Interessanter ist es da schon, den selben Befehl tausendmal ausführen zu lassen. Das ist doch schon eine ganz schöne Zeitersparnis: ich schreibe einen Befehl auf und weise meinen Rechenknecht an, das doch bitte 3000mal zu machen.[12] Dies wird in der Datenverarbeitung eine Schleife genannt.

Es ist auch interessanter für uns, wenn wir dem Rechner überlassen können, was er macht, zum Beispiel in allen Dateien, in denen sowohl das Wort „Salat“ als auch das Wort „Gurke“ auftaucht, die Gurke durch eine Salatgurke zu ersetzen, in allen anderen die Gurke durch eine Gewürzgurke. Der Rechner (das Programm) steht also vor der Alternative: entscheide mal, ob das eine Salatgurke oder eine Gewürzgurke ist, aber bitte intelligent!

[12] Denken wir an unsere Schulzeit zurück und die Strafarbeit: Du schreibst jetzt 100mal „Ich soll meinen Nachbarn nicht mit der Stecknadel pieksen.“

ANMERKUNG

Soll ein Programm-Quelltext (in Python) Kommentare enthalten? Allgemein gilt: den Programm-Quelltext sollte nicht nur ich, der Schreiber, auch noch nach einigen Monaten oder Jahren verstehen können, sondern auch ein eventueller Leser, der etwas an dem Programm ergänzen oder verbessern möchte. Hilfreich dabei ist natürlich, dass man die Variablen, grundsätzlich alle Bezeichner, mit Namen versieht, die auf die Bedeutung verweisen. Natürlich könnte man beispielsweise in einem Programm zur Berechnung von Größen in rechtwinkligen Dreiecken die Variablen so benennen: `Gandhi`, `Hilde` und `Schatzi` Aber ob dadurch der folgende Quelltext besonders verständlich wäre, ist doch zu bezweifeln: **Schatzi = Gandhi / Hilde** Etwas besser wäre schon die Benennung `gegenkathete`, `hypotenuse` und `sinus` womit das obige Programmfragment zu **sinus = gegenkathete / hypotenuse** wird und damit doch verständlicher. Trotzdem: auch so würde das wahrscheinlich kein Programmierer schreiben, denn diese Spezies Menschen zeichnet sich durch Faulheit aus, und selbst bei Editoren, die automatische Wort-Ergänzung beherrschen, ist das für viele Programmierer zu viel Schreibarbeit, weswegen man eher etwas fände wie **sin = gegenk / hyp**

Manchmal ist es aber hilfreich, wenn man im Quelltext eine kurze Erläuterung findet, was die Variable für eine Bedeutung hat. Mathematiker speziell — das sind in der Regel noch faulere Menschen als Programmierer — benennen Zähler grundsätzlich mit *i*, wenn sie einen weiteren Zähler brauchen, heißt der i_2 und das Optimum an Kreativität ist der Name *j* für einen weiteren Zähler. Ein häufiges Problem in der Informatik ist es, eine Matrix zu durchsuchen. Dazu benötigt man in der Regel zwei Zähler, einen für die Zeilen, einen für die Spalten. Also, pfiffig wie Programmierer sind, werden diese beiden Zähler *zz* und *sz* benannt. In einem Quelltext ist es dann durchaus sinnvoll, dass man dazu einen Kommentar schreibt. Kommentare werden durch einen Lattenzaun (*#*) eingeleitet. Alles von diesem *#* bis zum Zeilenende wird von Python ignoriert und ist nur für die Augen eines eventuellen Lesers bestimmt:

Beispiel 2.7.1 Kommentare

```
# zz ist der Zeilenzähler für die Matrix
# sz ist der Spaltenzähler für die Matrix
zz = 1
```

Eine besonders hilfreiche Eigenschaft von Kommentaren ist, dass man damit eine einfache Hilfe-Funktion für zum Beispiel eine Funktion schreiben kann. Der Hilfetext wird in die dreifachen Anführungszeichen an den Anfang der zu beschreibenden Funktion geschrieben

Beispiel 2.7.2 Dokumentation einer Funktion

```
>>> def quadrate(zahl):
  '''Der Funktion quadrate wird als
  Parameter eine Zahl mitgegeben,
  die Funktion liefert das Quadrat der
  Zahl zurück'''

  return zahl * zahl
```

Der Hilfetext kann jetzt dadurch aufgerufen werden, dass man mit Hilfe der print-Funktion den Namen der Funktion, gefolgt von zwei Unterstrichen, gefolgt vom Text *doc*, gefolgt von nochmals zwei Unterstrichen eingibt.

Beispiel 2.7.3 Dokumentation einer Funktion (Ausgabe)

```
>>> print quadrate.__doc__
Der Funktion quadrate wird als
Parameter eine Zahl mitgegeben,
die Funktion liefert das Quadrat der
Zahl zurück
```

Und die Funktion funktioniert so wie gewünscht:

Beispiel 2.7.4 Beispiel für eine einfache Funktion

```
>>> quadrate(3)
9
```

2.8. Programmierstil und Konventionen

Viele der Bemerkungen, die in Büchern zu anderen Programmiersprachen gemacht werden, sind in Python zum Glück überflüssig. Ordnung (und damit ein guter Programmierstil) ergibt sich in Python automatisch durch das Prinzip der Einrückung. So wie in jedem Text einer natürlichen Sprache wie Deutsch ist der Leser froh, wenn er auf einen Blick erkennt, was zusammenhängt. Der Autor benutzt zu diesem Zwecke die altbekannten Mittel der Gliederung: Kapitel, Absätze, Sätze, Satzteile.

In Python ist Einrückung nicht eine Möglichkeit, sondern eine Pflicht. Was zusammengehört, muss auf der selben Einrückungsstufe stehen. Wer schon einmal einen Programmierkurs gegeben hat, weiß das sehr zu schätzen. Jeder weiß, dass Anfänger des Programmierens, wenn sie nicht müssen, oft schnell etwas hinschreiben, ohne sich eine Struktur zu überlegen. Das kann dann in einer Sprache wie perl so aussehen:

Beispiel 2.8.1 Schlechter Stil! So nicht!! (Das ist auch nicht Python)

```
#!/usr/bin/perl
 $caps = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";$kls  = "abcdefghijklmnopqrstuvwxyz";
 $gesamt = 0;

 sub toCaps {local($c) = @_;

    if (ord($c) > 96 && ord($c) < = 123) {
 $cc = substr($caps,index($kls,$c),1);
 $gesamt++;} elsif (ord($c) > 64 && ord($c) <= 91) {$cc = $c;
 $gesamt++;} else {$cc = " ";}return $cc;
 } print "Eingabe eines Textes: ";

 $zeile = <STDIN>;
 chomp $zeile;
 foreach $j(0..25) {$anz[$j] = 0;}

 foreach $i(0..length($zeile)-1) {$z = substr($zeile,$i,1);
          if ($z ne " ") {
 &toCaps($z); $caps_pos = index($caps,$cc); $anz[$caps_pos]++;  } else {
 $cc = $z;}
 }

 open(AUS,">haeuf.txt");
 print AUS "Buchstaben insgesamt: $gesamt\n";
 print AUS "Buchstabe absolut relativ\n";
 foreach $j(0..25) {
    print AUS "  ".substr($caps,$j,1)." :        ".
    $anz[$j]."      ".$anz[$j]/$gesamt."\n";}close AUS;
 exit 0;
```

Das sieht aus, wie wenn es ohne Punkt und Komma geschrieben wäre.[13] Dieses perl-Programm ist lauffähig. Aber wirklich nicht gut lesbar. Eine Gliederung ist nicht zu erkennen. In Python ginge so etwas nicht!

ANMERKUNG

In jedem Unternehmen gibt es heutzutage etwas wie eine „corporate identity“. In jeder EDV-Abteilung mit Programmierern gibt es dafür eine Sammlung von Konventionen. Da wird dann etwa festgelegt, dass Variablennamen immer mit einem Kleinbuchstaben anfangen, oder dass Variablennamen, die eine Zahl zum Inhalt haben, immer mit „z_“ anfangen. Jede dieser Sammlungen von Richtlinien kann unterschiedlich sein, jede hat aber ihre Berechtigung. Für sich selber als Anfänger der Programmierung sollte man solche Richtlinien auch aufstellen und sich daran halten. Es hilft!

Für Python gibt es auch eine solche Richtlinie. Man findet sie unter PEP 8 <http://wiki.python.de/PEP8(Uebersetzung)>

Da Python-Entwickler und Python-Programmierer eine gewisse Neigung zu dummen Sprüchen haben[14], findet man auch Regeln für die Programmierung in Python in Python selber: Das Zen von Python. Dazu gibt man einfach in Python den Befehl **import this** ein.

[13] Ist es auch. Die Gliederungsmerkmale sind die Strichpunkte.
[14] Du weißt noch, woher die Sprache ihren Namen hat?

2.9. Reservierte Wörter

In einer Programmiersprache muss man, wie in jeder gesprochenen Sprache auch, Vokabeln lernen. In einer Progammiersprache heißen diese Vokabeln „Reservierte Wörter“ oder „Schlüsselwörter“. Der erfreuliche Aspekt einer Programmiersprache: es gibt sehr wenige Vokabeln. [15] Keine Sorge: Vokabeln abfragen, wie früher im Latein-Unterricht, wird es bei der Python-Programmierung nicht geben. Man sollte aber hier einmal kurz drüberlesen, um sich vielleicht das eine oder andere Wort doch einzuprägen. Das einzige Problem mit den reservierten Wörtern ist, dass man sie nur in dem von den Python-Entwicklern vorgesehenen Zusammenhang benutzen darf. Dafür sind sie reserviert!

Die reservierten Wörter in Python stehen in der folgenden Tabelle.

Tabelle 2.2. Reservierte Wörter in Python:

and	as	assert	break	class
continue	def	del	elif	else
except	exec	finally	for	from
global	if	import	in	is
lambda	not	or	pass	print
raise	return	try	while	with
yield				

Beachte: In Python 3 ist print kein Schlüsselwort mehr, sondern eine Funktion!

2.10. Fehler (zum ersten)

She knows there's no success like failure
And that failure's no success at all

(Bob Dylan [16])

Wie in jeder natürlichen Sprache auch kann man in einer Programmiersprache etwas richtig ausdrücken mit dem Ergebnis, dass der Empfänger das versteht; man kann aber auch etwas fehlerhaft ausdrücken, was zur Folge hat, dass der Empfänger nichts versteht. In einer Programmiersprache ist der Empfänger der Interpreter oder der Compiler.

Die Reaktion des Empfängers ist aber sehr unterschiedlich. In einer Nachricht, die in einer natürlichen Sprache vom Sender zum Empfänger geht, versucht der Empfänger auch bei Fehlern noch den Inhalt der Nachricht zu verstehen, im Zweifelsfall dadurch, dass er den Sender um Bestätigung der Interpretation bittet. In einer Programmiersprache ist die Reaktion eindeutig: der Compiler bzw. der Interpreter beendet seine Arbeit einfach dadurch, dass er meldet: mit dem Geschriebenen kann ich nichts anfangen, weil es einen oder mehrere Fehler enthält.

Das ist für den Schreiber zuerst mal sehr enttäuschend. Und dann ist es mit Arbeit verbunden, denn jetzt muss der Schreiber seinen Text auf mögliche Fehler untersuchen. Bevor ich auf die einzelnen Arten von Fehlern eingehe muss ich aber hier eine allgemeine Bemerkung machen.

Zu Beginn des 21. Jahrhunderts ist das Unterrichten von Programmierung, ganz unabhängig von der Programmiersprache, wesentlich schwieriger als in den 70er-Jahren des 20. Jahrhunderts. Der Grund ist ganz einfach: kaum jemand kann heute noch richtig schreiben. Besonders schwierig ist das Unterrichten in Programmierung an der Schule, denn korrektes Schreiben ist leider überhaupt nicht mehr nötig, um einen Schulabschluß zu erhalten, nicht einmal mehr im Abitur. Das schöne Wort „Verbesserung“ ist sowieso aus dem Wortschatz jedes (Deutsch-)Lehrers verschwunden nach dem Motto: „das tu ich mir doch nicht an, dass ich den Quatsch ein zweites Mal lese“, weswegen Schüler dieses Wort und die damit verbundene Tätigkeit nicht mehr kennen. Ein Beispiel aus einer durchschnittlichen 12. Klasse: in einem Programm sollte eine Funktion „erhöhen“ geschrieben werden. Die verschiedenen Schreibweisen für das Wort „erhöhen“ variierten je nach Schüler von 2 bis 4 (erhoeen, erhöen, errhöhn, erhoehn ...). Aufgefordert, den Programmtext nochmals zu lesen, war die Antwort meistens: Das hab ich schon gemacht, da seh ich keinen Fehler.

[15] Wer eine masochistische Ader hat: es gibt Sprachen mit noch wesentlich weniger Vokabeln. Die Sprache „brainfuck“ (J'y suis pour rien) hat überhaupt keine Vokablen und kennt auch nur acht Befehle. Viel Spaß. Siehe: <http://de.wikipedia.org/wiki/Brainfuck>

[16] Love Minus Zero *auf:* Bringing it all back home

Was also in einem Aufsatz die Note 2-3 ergibt, führt in der Programmierung zur Note 6.
Jetzt also zu den verschiedenen Arten von Fehlern:

- Syntax-Fehler. In der deutschen Sprache gelten etwa die Syntax-Regeln, dass ein Satz mit einem Großbuchstaben beginnt und mit einem Satzzeichen (Punkt, Ausrufezeichen, Fragezeichen) endet. In jeder Programmiersprache gibt es ähnlich einfache Syntax-Regeln. Fehler in der Syntax sind die am einfachsten zu findenden Fehler (sollte man meinen; siehe aber dazu nochmals den vorhergehenden Absatz). In einer natürlichen Sprache sind Syntaxfehler noch kein Hindernis für eine funktionierende Kommunikation. In einer Programmiersprache ist der Unterschied der von einem lauffähigen Programm zu ... nix. Ein Stück Programmcode mit Syntaxfehlern ist nix.
- Semantische Fehler. Diese Fehler sind gekennzeichnet durch syntaktisch korrekte Anweisungen in einem Programm. Verglichen mit einer natürlichen Sprache bedeutet das, dass die einzelnen Sätze in dem kommunizierten Text korrekt sind. Trotzdem macht das Programm nicht das, was es soll. Der Text ergibt keinen Sinn (oder nicht den erwarteten Sinn). Semantische Fehler sind nicht immer leicht zu finden, selten durch Probieren, Umkodieren, Ändern von Programmen. Leichter findet man solchen Fehler, wenn man sich vom Rechner löst und das Problem neu durchdenkt.
- Laufzeitfehler. Wie der Name schon sagt, treten solche Fehler erst bei der Ausführung des Programms auf. Syntax und Semantik scheinen in Ordnung, aber das Programm bricht aus irgendeinem (auf den ersten Blick unerklärlichen) Grund ab. Ein typisches Beispiel ist, dass man im Programm codiert hat, dass etwas an der 5. Stelle (von irgendetwas) gelesen werden soll. Das „irgendetwas“ hat aber nur 3 Stellen.

2.11. Fehler finden

Wie findet man aber Fehler? Es gibt natürlich Programme, bei denen man eventuelle Fehler mit klassischen Mitteln findet, nämlich mit

- exaktem Lesen
- gründlichem Nachdenken

Vor allem bei den Übungsprogrammen, die ja selten mal länger als 50 oder 100 Zeilen sind, ist der Aufwand überschaubar, und scharfes Hinschauen und Nachdenken hilft hier. Aber bei längeren Programmen ist man dankbar für eine Hilfe, aus historischen Gründen „debugger“ genannt. Ein Debugger erlaubt es, die Ausführung eines Programmes an einer beliebigen Stelle zu unterbrechen und den Inhalt der Variablen zu diesem Zeitpunkt anzuschauen. Für viele Programmiersprachen und in vielen IDE's (siehe nächstes Kapitel) existieren Debugger. Auch das sollte sich ein Programmieranfänger einmal anschauen, wenn er eine IDE auswählt: die IDE ist die beste, die mir am besten gefällt.

Die „Hardcore“-Programmierer verschmähen natürlich einen Debugger. Okay, in Wirklichkeit tun sie es nicht, aber ihr Debugger heißt **print**! Das, was ich im vorigen Absatz geschrieben habe, erledigen sie durch die Ausgabe von Variablen-Inhalten mittels der **print**-Anweisung ... und haben dadurch einen Minimal-Debugger.

2.12. Aufgaben

1. Lies diagonal über die Python-Seite im WWW: <http://www.python.org/> (Gewöhne Dich gleich daran: in der Informatik ist vieles nur auf Englisch zu finden!)
2. Wenn Du Kinderbücher magst: schau Dir mal „Snake Wrangling for Kids“ an. Das gibt es in einer deutschen Übersetzung unter <http://code.google.com/p/swfk-de/downloads/list>

Teil II.

IDE

3. Die Entwicklungsumgebung IDLE: Zahlen und Variable

3.1. ... und bevor es losgeht ...

Die fast wichtigste Regel, wenn man sich ans Programmieren begibt, kommt natürlich hier auch als erstes: **Erst denken, dann tippen**

Das hört sich einfach an und selbstverständlich, aber viele, auch erfahrene Programmierer lesen eine Aufgabenstellung, setzen sich an die Tastatur und fangen an zu schreiben. Dann ist das Programm fertig, und der Programmierer stellt fest: ist das umständlich geschrieben, ist das unschön, das geht doch verständlicher. Es ist wie beim Aufsatzschreiben: ein gutes Konzept macht das eigene Werk übersichtlicher, verständlicher und meistens auch kürzer.

Die zweite Regel ist fast genauso wichtig: **Ein Programm ist erst fertig, wenn es ausreichend getestet ist.**

Und dann hast Du das Programm geschrieben, hast es einmal gestartet und es hat funktioniert, wie Du Dir das vorgestellt hast. Also legst Du die Füße hoch, freust Dich, dass Du mit Deiner Arbeit fertig bist und gibst das Ergebnis Deinem Chef (wenn programmieren Dein Job ist) oder Deinem Lehrer (wenn Du Schüler oder Student oder sonst irgend ein Lernender bist und das eine Hausaufgabe oder eine Prüfungsaufgabe war). Der Chef (oder Lehrer) startet das Programm, und es tut sich nichts (oder etwas ganz unerwartetes).

Denn Du als Programmierer hast vergessen, dass derjenige, der das Programm anwendet (oder überprüft) leider nicht genau die Daten eingibt, die Du eingegeben hast und mit anderen Daten läuft das Programm überhaupt nicht.

3.2. Rechnen in der Entwicklungsumgebung

Eine Entwicklungsumgebung, auf englisch *Integrated Development Environment*, wird mit IDE abgekürzt. Da ist es kein Zufall, dass die Entwicklungsumgebung von Python **idle** heißt. Zum einen, weil idle ja ein schönes englisches Wort ist, dessen Bedeutung Mathematikern und Programmierern zum Prinzip geworden ist, zum anderen weil der fröhliche Mensch auf dem Bild, wie allen Freunden des „Leben des Brian“ bekannt, Eric Idle heißt.

Abbildung 3.1. Eric Idle (pfeift Always look on the bright side of life)

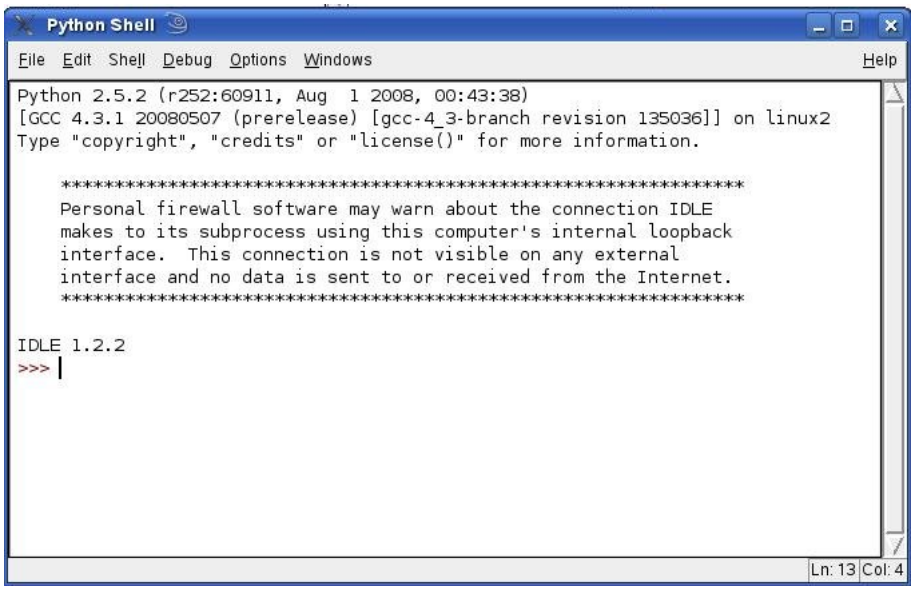
Zu beachten ist unter Linux (SuSE; die anderen Distributionen habe ich nicht ausprobiert), dass IDLE nicht standardmäßig installiert wird. Das ist in einem separaten Paket.

ANMERKUNG

Anscheinend ist IDLE nicht mehr die Standard-Python-Shell. Mit dem neuen Python ist eine verbesserte Entwicklungsumgebung namens „ipython“ im Python-Paket enthalten. Trotzdem sind hier noch die Bildschirmphotos von idle enthalten. Die von ipython unterscheiden sich nur wenig hiervon.

Rufen wir also einfach mal IDLE auf! Der Bildschirm von IDLE sieht so aus!

Abbildung 3.2. Die IDE Idle

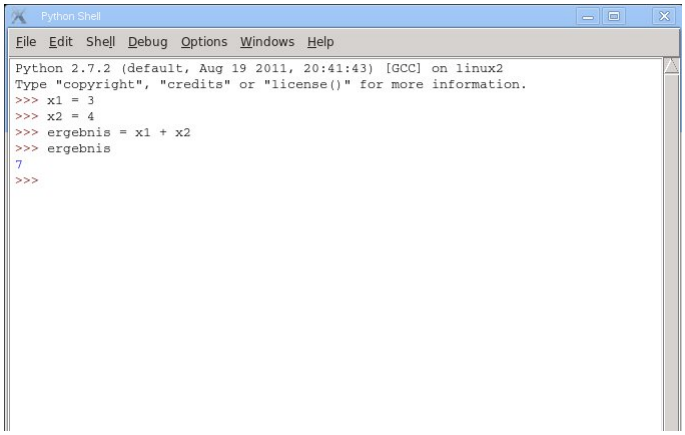


Jetzt geben wir einfach mal etwas in IDLE ein, in Erinnerung dessen, dass ein Computer ein Rechner ist, eine komplizierte Mathematik-Aufgabe:

```
>>> x1 = 3
>>> x2 = 4
>>> ergebnis = x1 + x2
```

Das IDLE-Fenster sieht danach so aus:

Abbildung 3.3. Erste Berechnung in der IDE Idle



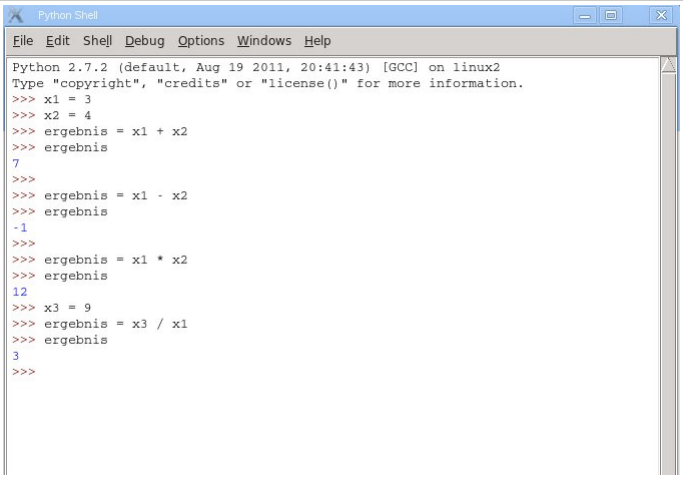
Nachdem wir unser erstes Python-Programm geschrieben haben, werden wir mutig bis übermütig. Lassen wir den Rechner rechnen! Die arithmetischen Operatoren sehen vertraut aus, zwar nicht so, wie man sie mit der Hand schreibt, sondern so wie man sie auf der Computer-Tastatur sieht.

- Addition 6 + 3
- Subtraktion 6 - 3
- Multiplikation 6 * 3
- Division 6 / 3
- Exponentiation 2**3
- Modulus (Rest) 5%3

Das ist der passende Zeitpunkt, um in IDLE ein bißchen herumzuspielen und ein paar Rechnungen ausführen zu lassen. Und vielleicht ist das auch ein Grund, immer IDLE laufen zu lassen. [1]

Das Ergebnis überrascht uns nicht. Der Rechner kann rechnen.

Abbildung 3.4. Rechnungen in Idle



Was passiert aber, wenn wir

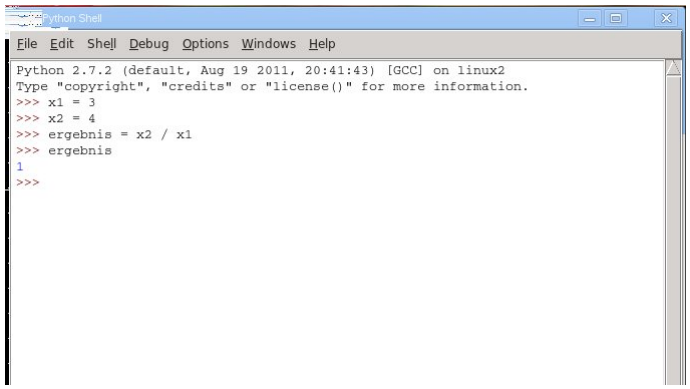
```
>>> x1 = 3
>>> x2 = 4
>>> ergebnis = x2 / x1
```

[1] Wer unter Linux / KDE arbeitet, weiß die Möglichkeit von virtuellen Bildschirmen zu schätzen und wird sich wahrscheinlich jetzt schon längst einen virtuellen Bildschirm eingerichtet haben, auf dem nur IDLE läuft.
Und wer unter Windows arbeitet, weiß vielleicht gar nicht, wie komfortabel grafische Oberflächen sein können!!!

eingeben? Hier kommt die Eingabe mit Ergebnis:

```
>>> x1 = 3
>>> x2 = 4
>>> ergebnis = x2 / x1
>>> ergebnis
1
```

Abbildung 3.5. Division von ganzen Zahlen



So hatten wir uns das nicht gedacht!! Was ist passiert? Da wir die Zahlen als ganze Zahlen eingegeben haben, ist das Ergebnis auch eine ganze Zahl. Um das gewünschte Ergebnis zu bekommen, müssen wir Python „austricksen“ und die Zahlen bereits als Dezimalzahlen eingeben. Beachte dabei: die Grundlage bei der Entwicklung von Python (wie von fast allen Programmiersprachen) ist englisch, also gibt es kein Dezimalkomma, sondern einen Dezimalpunkt.

WICHTIG

In Python 3 gilt das nicht mehr. Der Divisions-Schrägstrich / ergibt hier immer eine Dezimalzahl als Ergebnis. Wenn man in Python 3 eine Ganzzahl-Division durchführen will muss man den Doppel-Schrägstrich // benutzen.

ANMERKUNG

Wir merken uns also: das Ergebnis einer arithmetischen Operation hat immer den Typ der Operanden, wenn diese den selben Typ haben. Wenn die Operanden verschiedene Typen haben, dann wird der umfassendere Typ genommen. [a]

[a] Das ist mathematisch (als Mengen gesehen) zu verstehen: $\mathbb{N} \subseteq \mathbb{Z} \subseteq \mathbb{Q} \subseteq \mathbb{R} \subseteq \mathbb{C}$

WICHTIG

In Python 3.x gilt das nicht mehr.

Das probieren wir gleich aus, indem wir in IDLE

Beispiel 3.2.1 Divisionen

```
    >>> x1 = 3.0
    >>> x2 = 4.0
      >>> ergebnis = x2 / x1
      >>> ergebnis
1.3333333333
```

eingeben.
Die Operation

```
    >>> x1 = 3
    >>> x2 = 4
      >>> ergebnis = x2 / x1
    >>> ergebnis
    1
```

hingegen führt die Division so durch, wie wir es in der Grundschule gelernt haben:

```
            4 / 3 = 1 Rest 1
```

Es gibt aber wirklich viele Anwendungen, bei denen man genau das will, und dazu will man noch den Rest berechnet haben. Dazu gibt es den Rest-Operator, korrekt Modulus genannt. Wir geben also ein, um das zu testen:

Beispiel 3.2.2 Modulo-Rechnen

```
      >>> 8 % 3
      2
```

und das liefert uns den Rest 2 bei der Division von 8 durch 3.
Python bietet uns sogar noch mehr:

Beispiel 3.2.3 Division und Rest

```
   >>> divmod(13,3)
   (4,1)
```

liefert als Ausgabe ein Paar von Zahlen, wobei die erste Zahl das ganzzahlige Ergebnis von 13 / 3 ist, die zweite Zahl der Rest bei dieser Division. Dieses Paar ist ordentlich in Klammern geschrieben, und das soll an dieser Stelle uns erst einmal nur erfreuen. Was diese Schreibweise genau bedeutet, folgt in einem späteren Kapitel.

Bisher haben wir Berechnungen durchgeführt und das Ergebnis der Rechnung implizit ausgeben lassen, das heißt, wir haben Python nicht ausdrücklich angewiesen, etwas auszugeben. Der Befehl, um etwas an den Bildschirm auszugeben, lautet **print**.

Tabelle 3.1. Arithmetische Operatoren in Python

Operation	math. Zeichen	Beispiel	Ergebnis
Addition	+	6 + 3	9
Subtraktion	-	6 - 3	3
Multiplikation	*	6 * 3	18
Division	/	6 / 3	2
Modulus	%	8 % 3	2
Exponentiation	**	6**3	216

WICHTIG

Ab Python 3 ist **print** kein Befehl, sondern eine Funktion.

Das bedeutet, dass statt bisher **print irgendwas** jetzt **print(irgendwas)** geschrieben werden muss. Das ist ja aus der Mathematik bekannt: Argumente einer Funktion werden in Klammern geschrieben, auch wenn in Mathematik es eigentlich nur eine Art von Funktionen gibt: f(x).

Das scheint keinen Unterschied zu machen, ob man in IDLE

```
>>> 3 + 4
```

oder

```
>>> print 3 + 4
```

eingibt. Aber das ausdrückliche Ausgeben mit „print“ ist intelligent, wenn mit Dezimalzahlen gerechnet wird. Ein Beispiel dazu:

Beispiel 3.2.4 Rundungsfehler

```
>>> 2 / 5.0
0.40000000000000002
>>> print 2 / 5.0
0.4
```

Man sieht: die Ausgabe mit „print“ rundet intelligent! [2]

[2] Das hört sich für den Laien seltsam an: dass 0.4 eine gerundete Zahl sein soll, ist doch sehr gewöhnungsbedürftig. Aber man muss sich ins Gedächtnis rufen, dass der Computer im Dualsystem rechnet. Das heißt, unsere Eingaben (in diesem Fall 2 und 5) werden zuerst in Dualzahlen umgewandelt. Dabei entstehen bereits Fehler (das sind eigentlich nur Ungenauigkeiten). Danach wird mit diesen Dualzahlen gerechnet, am Ende wird das Ergebnis wieder in eine Dezimalzahl verwandelt. Aus Sicht des Rechners ist das exakte Ergebnis für diese Division also 0.40000000000000002.

3.3. Ein Speicherplatz mit einem Namen: Variable

Man gave names to all the animals
In the beginning
Long time ago

(Bob Dylan [3])

3.3.1. Variable in Mathematik und in der Programmierung

Sinnvoll werden Berechnungen erst, wenn man die Werte nicht jedes Mal eingeben muss, sondern sie irgendwo speichern kann. Hier kommt das Prinzip der Variablen ins Spiel.

Variable haben einen Variablennamen und einen Wert. (Der Typ der Variablen wird in Python implizit gesetzt. So könnte man meinen, aber es ist viel schöner, als man es sich vorstellen kann: durch die Zuweisung eines Wertes an eine Variable wird die Variable ein Objekt des Typs, der durch den zugewiesenen Wert bestimmt ist.) Vorerst benutzen wir nur einfache Typen, das heißt Zahlen-Typen und Text-Typen.

Variable kennen wir schon aus der Mathematik. Allerdings sind die Mathematiker da sehr einfallslos (jedenfalls in der Schule), denn Variable heißen hier grundsätzlich `x`. Aber das Prinzip der Variablen ist hier das selbe: es wird ein Platzhalter geschaffen, der einen Namen bekommt, und diesem Platzhalter kann man einen Wert zuweisen. Und nach einer Weile einen anderen Wert, so dass man mit der Zeit in der Mathematik eine Wertetabelle bekommt.

In der (Schul-)Mathematik kennt man in der Analysis eine Variable, die dann fast immer `x` heißt, ganz selten `t`, wenn man Funktionen in der Physik untersucht, bei der die Zeit variabel ist. In der Linearen Algebra gibt es dann in der Schule schon mal 3 Variable, die dann `x`, `y` und `z` heißen, wenn man sie nicht sogar einfach x_1, x_2 und x_3 nennt. In der Informatik benötigt man, um die Herstellungskosten eines Produktes zu berechnen, viel mehr Informationen, also auch viel mehr Variable, die diese Informationen speichern, wie zum Beispiel `stromkosten`, `abschreibungskosten`, `personalkosten`, `materialkosten` und noch viele mehr.

In der Programmierung ist man da viel flexibler (aber das hat man auch erst im Laufe der Jahre gelernt). Variablennamen können länger als ein Zeichen sein, und das ist für die Lesbarkeit eines Programms immer sehr sinnvoll. Denn ein Programm ist besser zu lesen und zu verstehen, wenn die Variablennamen schon einen Hinweis auf die Werte geben, die sie später enthalten sollen. Natürlich kann man in einem Programm für geometrische Berechnungen auch Variablen `l` und `b` nennen, aber auf den ersten Blick verständlicher sind die Variablennamen `laenge` und `breite`.

3.3.2. Variablennamen

Variablennamen müssen mit einem Buchstaben oder mit einem Unterstrich beginnen, dann dürfen eine beliebige Anzahl von Buchstaben, Zahlen oder Unterstrichen folgen.

ACHTUNG

Auch wenn die Entwickler eines weitverbreiteten Betriebssystems etwas anderes behaupten: in allen Bezeichnern, so auch in Variablennamen, haben Leerzeichen und Sonderzeichen nichts zu suchen.

Python lehnt also eine Variable `länge` ab; richtig ist die Variable `laenge`.

Es ist sinnvoll und in produktiven Umgebungen oft auch gefordert, dass man sich beim Schreiben von Programmen an gewisse Regeln hält. Firmen etwa legen in internen Richtlinien fest, dass z.B. Variablennamen mit einem Kleinbuchstaben beginnen. Wenn der Variablenname ein zusammengesetztes Wort ist, dann wird der Wortanfang des zweiten (und jedes folgenden Wortes) mit einem Großbuchstaben geschrieben. Eine andere mögliche Vereinbarung ist, dass jeweilige weitere Wortanfänge durch einem Unterstrich „_“ mit dem vorigen Teil des Variablennamens verbunden werden. Es trägt einerseits zur Lesbarkeit von Programmcode bei, wenn hier Einheitlichkeit herrscht.[4] Andererseits ist es auch für

[3] Man gave names to all the animals *auf:* Slow Train Coming
[4] siehe hierzu auch PEP 8 <http://wiki.python.de/PEP8(Uebersetzung)>

die maschinelle Verarbeitung von Dateien, und Programme sind aus der Sicht des Dateisystems erst mal nur Dateien, hilfreich, wenn etwa eine Python-Prozedur eine ganze Menge von Dateien bearbeiten soll und Variablennamen nach dem selben Schema aufgebaut sind.

Tabelle 3.2. Variablennamen

erstes Zeichen	weitere Zeichen
Buchstabe; Unterstrich	Buchstabe; Ziffer; Unterstrich

Beispiele für zulässige Variablennamen:

- `zahl1`
- `zahl2`
- `grosseZahl`
- `ganzGrosseZahl`

Nicht zulässig sind (aus den angegebenen Gründen):

- `12Grad` (fängt nicht mit Buchstaben oder Unterstrich an)
- `grosse Zahl` (enthält Leerzeichen)

Auch hier muss ich auf etwas eingehen, was in unserem Kulturkreis lange Zeit eine Selbstverständlichkeit war: der Satz „Brandt hat in Moskau liebe Genossen“ hat eine andere Bedeutung als der Satz „Brandt hat in Moskau Liebe genossen“. Wenn man verschiedene Zeichen benutzt, bewirkt das eine unterschiedliche Bedeutung. Erst mit dem Aufkommen von PC's und dem eines (sehr schlichten) Betriebssystems aus einem inzwischen sehr großen Softwarehaus von der Westküste der USA[5] versuchte man, uns weiszumachen, dass die Datei `liebe.txt` und die Datei `Liebe.txt` das selbe ist. Und leider schlagen sich immer noch viele Software-Entwickler und PC-Benutzer mit diesem Übel herum.

In der Programmierung gilt einfach: Die Groß-/Kleinschreibung ist bei Variablen relevant!! `liebe` ist etwas anderes als `Liebe`. Punkt. Aus. Ende der Diskussion.

3.3.3. Wertzuweisung

Die Zuweisung eines Wertes an eine Variable ist leider inzwischen für viele Programmieranfänger ein riesiges Problem. Also muss ich hier ein paar Sätze darüber schreiben. Ein Wert, egal ob ein Zahlenwert wie „2,56“ oder ein Zeichenkettenwert wie „And now for something completely different“ schwebt aus der Sicht eines Programmierers nie frei in der Landschaft herum: er steht auf einem Speicherplatz. Wie ich oben geschrieben habe: der Speicherplatz hat in modernen Programmiersprachen einen Namen. Egal, ob ich den Wert lesen will, mit diesem Wert arbeiten möchte (ihn vergrößern, verändern, Teile davon ersetzen usw.), ihn löschen will oder was mir sonst noch einfällt: ich muss als Programmierer wissen, in welcher Variablen dieser Wert gespeichert ist. Wenn ich weiß, dass der Wert „And now for something completely different“ in der Variablen `unsinn` gespeichert ist, dann kann ich in einem Programm zum Beispiel die Anweisungen

- gib (den Inhalt der Variablen) `unsinn` aus
- verändere (den Text, der in der Variablen) `unsinn` (steht), so dass darin `strange` statt `different` steht
- lösche (den Inhalt der Variablen) `unsinn`

schreiben.

In fast allen Programmiersprachen, auch in Python, ist der Zuweisungsoperator das „=“, das Gleichheitszeichen.[6] Links des Gleichheitszeichens steht der Name der Variablen, der etwas zugewiesen wird, rechts des Gleichheitszeichens steht der Wert, der zugewiesen wird. **a = 17** bedeutet also, dass der Variablen mit dem Namen `a` der Wert 17 zugewiesen wird.

[5] schon rausgekriegt, welches ich meine?

[6] Eine Ausnahme bilden die „Wirth“-Sprachen, also Pascal, Modula und Oberon, die alle auf Niklaus Wirth zurückgehen. Hier ist der Zuweisungsoperator ein „:=“. Dafür ist der Vergleichsoperator auf Gleichheit hier ein einfaches „=“.

Das hat den großen Nachteil, dass Zuweisungen immer zwei Tastendrücke bedeuten, und Zuweisungen sind häufiger als Vergleiche! Den Mathematiker allerdings erfreut es, denn so lernt er es in den Anfangssemestern: **a := 17** bedeutet: a hat jetzt per Definition den Wert 17. Genau das ist aber eine Zuweisung.

WICHTIG

Gewöhne Dir gleich an, vor und nach dem Gleichheitszeichen genau ein Leerzeichen zu lassen ... und merke Dir das auch für alle anderen Texte die Du schreibst, nicht nur für Programme! [a]

[a] siehe hierzu auch PEP 8 <http://wiki.python.de/PEP8(Uebersetzung)>

Gültige Zuweisungen sind also:

Beispiel 3.3.1 Korrekte Zuweisungen

```
>>> zahl1 = 123.56
>>> zahl2 = 99
>>> text1 = 'Das Leben des Brian'
>>> text2 = 'Volleyball macht Spass'
```

Ungültige Zuweisungen (die dann zu Fehlermeldungen führen) sind:

Beispiel 3.3.2 Fehlerhafte Zuweisungen

```
>>> 6 = zahl
>>> 'Mathematik' = 12
```

3.3.4. Übungen zu Variablen

Weil dieses Konzept der Wertzuweisung an Variable für Anfänger so problematisch ist, folgen hier einige Übungen. Die Übungen sind sehr ausführlich formuliert: so langatmig bekommt man wahrscheinlich nie mehr Anweisungen, wie etwas programmiert werden soll!!

1. Weise einer Variablen mit dem Namen `laenge` den Wert 4 zu, weise einer anderen Variablen mit dem Namen `breite` den Wert 12 zu, weise einer Variablen mit dem Namen `flaeche` den Ausdruck `laenge * breite` zu. Was macht das Programm also? Welchen Namen würdest Du diesem Programm geben?
2. Weise einer Variablen mit dem Namen `laenge` den Wert 4 zu, weise einer anderen Variablen mit dem Namen `breite` den Wert 12 zu, weise einer Variablen mit dem Namen `umfang` den Ausdruck `2 * laenge + 2 * breite` zu. Was macht das Programm also? Welchen Namen würdest Du diesem Programm geben?
3. Weise einer Variablen mit dem Namen `laenge` den Wert 4 zu, weise einer anderen Variablen mit dem Namen `breite` den Wert 12 zu, weise einer dritten Variablen mit dem Namen `hoehe` den Wert 7 zu,weise einer Variablen mit dem Namen `volumen` den Ausdruck `1 / 3 * laenge * breite * hoehe` zu. Was macht das Programm also? Welchen Namen würdest Du diesem Programm geben?

3.3.5. Zuweisungsmuster

Aus einer netten Internetseite von (siehe [Lusth]) übernehme ich hier die Idee der „Zuweisungsmuster". Lusth unterscheidet

- das Transfer-Muster
- das Veränderungs-Muster
- das Wegschmeiß-Muster

Nehmen wir uns also die drei Muster der Reihe nach vor, aber in einer Kurzfassung. Wer interessiert ist, sollte sich den Originaltext vornehmen!

1. Beim Transfer-Muster betrachten wir 2 Variable:

Beispiel 3.3.3 Transfer-Muster

```
>>> a = 5
>>> b = 7
```

(oder in Worten: der Variablen `a` wird der Wert 5, der Variablen `b` der Wert 7 zugewiesen.) Was passiert aber, wenn ich schreibe: **a = b**? Welchen Wert hat nach dieser Zuweisung die Variable `a`, welchen die Variable `b`?

Nun, einfach ist eine Antwort für `b`: an `b` ändert sich nichts, also muss der Wert von `b` immer noch 7 sein. Anders sieht es bei `a` aus: `a` wird `b` zugewiesen. Genau muss ich sagen: der Variablen `a` wird der Wert zugewiesen, der in der Variablen `b` steht. Folglich steht in der Variablen `a` nach der Zuweisung **a = b** auch der Wert 7.

2. Beim Veränderungs-Muster wird der Inhalt einer Variablen verändert.

Beispiel 3.3.4 Veränderungs-Muster

```
>>> a = 5
>>> a = a + 3
```

Was passiert? In der ersten Zeile wird der Variablen `a` der Wert 5 zugewiesen. In der zweiten Zeile steht, in Worte gefasst: nimm den Wert der Variablen `a`, zähle zu diesem Wert 3 dazu und speichere das Ergebnis wieder in der Variablen `a`. Damit hat `a` den Wert 8.

3. Das Wegschmeiß-Muster ist das, was viele Anfänger benutzen!

Beispiel 3.3.5 Wegschmeiß-Muster

```
>>> a = 5
>>> a + 3
```

Das sieht sehr ähnlich aus wie das vorige Muster. Der Variablen `a` wird in der ersten Zeile der Wert 5 zugewiesen. In der folgenden Zeile wird der Wert von `a` um 3 erhöht ... und weggeschmissen. Genau gesagt: der erhöhte Wert wird nirgends gespeichert, ist also für alle Zeit verloren.

3.4. Rechnen und vergleichen

Nachdem wir jetzt elementares Rechnen in IDLE ausprobiert haben, sollten auch noch weitere Operatoren genannt werden. Hier kommen zuerst einmal die üblichen Vergleichsoperatoren.

Besonders zu beachten in der obigen Tabelle sind die beiden Operatoren für die „Gleichheit". Das sollte man auch in IDLE einmal ausprobieren, indem man zwei Variable definiert, die den selben Inhalt haben:

Beispiel 3.4.1 Gleichheit

```
>>> p1 = 431 * 1289
>>> p2 = 555 559
```

Dann ergibt

Tabelle 3.3. Vergleichsoperatoren

Zeichen	Bedeutung
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich
==	gleich (gleicher Wert)
is	gleich (das selbe Objekt)
<>	ungleich (verschiedene Werte)
!=	ungleich (verschiedene Werte)
is not	ungleich (verschiedene Objekte)

```
>>> p1 == p2
True
```

das Ergebnis *True*, während

```
>>> p1 is p2
False
```

das Ergebnis *False* liefert. *True* bzw. *False* sind hier selbstverständlich die beiden logischen Wahrheitswerte „wahr“ bzw. „falsch“.

Ebenso ist beachtenswert, dass Python hier beim Vergleich von Zahlen den Typ berücksichtigt.

```
>>> a = 3
```

belegt eine Variable *a* mit dem Wert 3. Der Vergleich

```
>>> a == 6 / 2
True
```

liefert folglich den Wert „True“. Auch wenn man vergleicht

```
>>> a == 6. / 2.
True
```

(man beachte die beiden Punkte! Hier werden zwei Dezimalzahlen geteilt!), erhält man den Wert „True“. Der Vergleich

```
>>> a is 6 / 2
True
```

liefert auch den Wert „True“, wenn man allerdings Python fragt, ob

```
>>> a is 6. / 2.
False
```

gilt, bekommt man die Antwort „False“.

Speziell für Mathematiker und Informatiker interessant sind die Verschiebe-Operatoren. Verschiebe-Operatoren verschieben das (unter Umständen gedachte) Dezimalkomma nach rechts oder links. Im Dezimalsystem bewirkt das eine Multiplikation mit 10 (Komma wird um eine Stelle nach rechts verschoben: aus 12,0 wird 120) oder eine Division durch 10 (Komma wird um eine Stelle nach rechts verschoben: aus 12,0 wird 1,20)

So sieht das also im Dezimalsystem aus.

```
>>> 753 * 10 = 7530
```

Da der Computer Zahlen im Dualsystem darstellt, bedeutet die oben beschriebene Operation (Ziffernfolge um eins nach links verschieben, hinten mit Null auffüllen) hier eine Multiplikation mit 2. Aus diesem Grund gibt es in Python (wie in vielen anderen Programmiersprachen) spezielle Operatoren für die Multiplikation mit 2 (bzw. mit Potenzen von 2) und die Division durch 2 (bzw. durch Potenzen von 2).

Tabelle 3.4. Verschiebeoperatoren

Operator	Beschreibung	Bedeutung	mathematisches Beispiel
<<	Verschiebung nach links	Multiplikation mit 2 (Potenz von 2)	3 << 1 = 6
			3 << 2 = 12
			3 << 4 = 48
>>	Verschiebung nach rechts	Division durch 2 (Potenz von 2)	16 >> 1 = 8
			16 >> 2 = 4
			16 >> 3 = 2

3.4.1. Komplexe Zahlen

You say you want a real solution

(John Lennon / Paul McCartney[7])

Für Mathematiker, Physiker und Techniker besonders interessant ist, dass Python von Haus aus komplexe Zahlen beherrscht. Komplexe Zahlen werden (fast) genauso geschrieben, wie es Mathematiker gewöhnt sind, das kleine „i“ wird durch ein kleines „j“ ersetzt:

```
>>> (1 + 2j)
```

Das sollte man jetzt (sofern man gerade vor dem Rechner sitzt) einmal ausprobieren. Eine komplexe Zahl besteht ja aus einem Realteil und einem Imaginärteil. Auch die kann man sich natürlich anzeigen lassen:

Beispiel 3.4.2 Komplexe Zahlen

```
>>> kzahl = 5 + 2j
>>> kzahl.real
5
>>> kzahl.imag
2
```

[7] Revolution *auf* The Beatles (White Album)

3.4.2. Andere Zahlsysteme

Für Programmierer interessant und deswegen in fast jeder Programmiersprache als ein Merkmal realisiert ist die Fähigkeit, Zahlen in anderen Zahlsystemen als dem Dezimalsystem darzustellen. Oktalzahlen werden mit einer führenden *0* eingegeben, Hexadezimalzahlen mit einem führenden *0x* und Dualzahlen mit einem führenden *0b*.

Tabelle 3.5. Verschiedene Zahlsysteme und Umrechnung

010 ist die Oktalzahl 10	$010 = 8_{dez}$
0x10 ist die Hexzahl 10	$0x10 = 16_{dez}$
0b10 ist die Dualzahl 10	$0b10 = 2_{dez}$

Dann sollte es natürlich auch möglich sein, von einem in das andere Zahlsystem umzurechnen. Die dafür zuständigen Funktionen heißen

1. `hex` für die Umrechnung einer Zahl im Dezimalsystem in eine hexadezimale Zahl. Das Ergebnis ist eine Zeichenkette mit vorangestelltem `0x`.
2. `oct` für die Umrechnung einer Zahl im Dezimalsystem in eine oktale Zahl. Das Ergebnis ist eine Zeichenkette mit vorangestelltem `0`.
3. `int` für die Umrechnung einer Zeichenkette (also auch einer Hexadezimal- oder Oktalzahl) in eine Zahl im Dezimalsystem.

In idle eingegeben sieht das so aus:

Beispiel 3.4.3 Umwandlung von Zahlen in ein anderes Zahlsystem

```
>>> zahl = 15
>>> hex(zahl)
'0xf'
>>> oct(zahl)
'017'
>>> bin(zahl)
'0b1111'
>>> x = 0b1111
>>> x
15
>>> oct(x)
'017'
>>> hex(zahl)
'0xf'
```

3.5. Shortcut-Operatoren

Es gibt in Python (wie in vielen anderen Programmiersprachen) Abkürzungen für oft benutzte Rechenoperationen, auf englisch „shortcuts“. Für den Programmieranfänger sind diese abgekürzten Schreibweisen oft nicht so übersichtlich wie eine ausführliche Schreibweise. Aber wenn man sich das laut vorliest, was gemacht werden soll, dann ist das eine große Erleichterung, wenn man diese Abkürzungen benutzt.

Immer wieder muss man zum Beispiel den Wert einer Variablen erhöhen. In einem konkreten Beispiel heißt das, dass die Variable „alter“ den Wert 17 hat, aber am nächsten Geburtstag muss die Variable auf 18 erhöht werden. Der Programmieranfänger schreibt das meistens so:

Beispiel 3.5.1 noch kein Shortcut

```
>>> alter = 17
>>> alter =  alter + 1
```

Klar, das löst das Problem. Auf den aktuellen Wert, den die Variable hat, wird 1 addiert und nach der Addition wird der erhöhte Wert wieder in der Variablen „alter“ gespeichert.

Die Abkürzung schreibt sich so:

Beispiel 3.5.2 Jetzt mit Shortcut

```
>>> alter =  17
>>> alter += 1
```

Das kann man sich selbst so vorlesen, wie vor diesem Beispiel!

Shortcuts gibt es für alle elementaren arithmetischen Operationen. Das wird in der folgenden Tabelle so aufgelistet.

Tabelle 3.6. Shortcut-Operatoren

Operation	lange Version	Shortcut
Addition	x = x + 2	x += 2
Subtraktion	x = x - 3	x -= 3
Multiplikation	x = x * 5	x *= 5
Division	x = x / 12	x /= 12

4. Richtig programmieren

4.1. Entwicklungsumgebungen

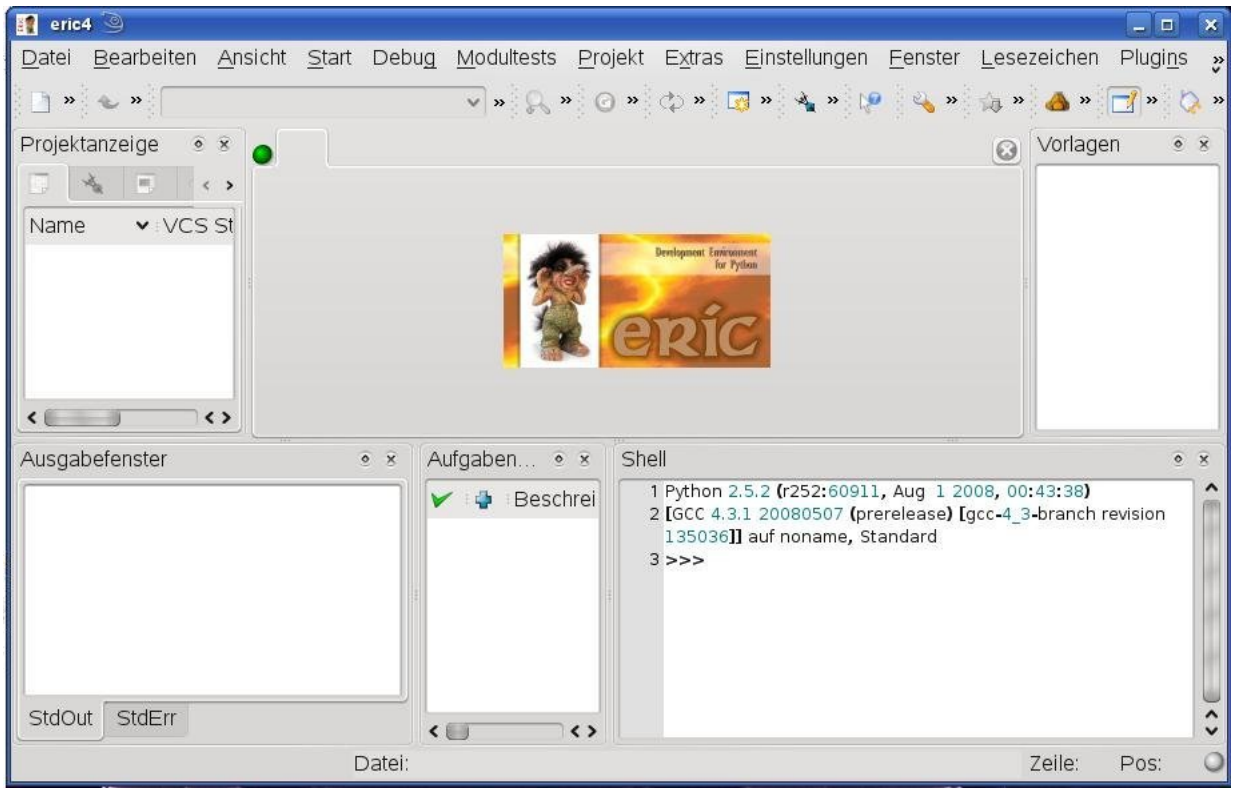
4.1.1. Die Entwicklungsumgebung Eric

Nachdem wir also eine ganze Weile jetzt mit IDLE gearbeitet haben, stellen wir fest: um einen oder vielleicht zwei oder drei Python-Befehle zu testen ist das nicht schlecht. Auch ein erfahrener Programmierer hat IDLE oft nebenher geöffnet, um einmal die Wirkungsweise eines Befehls auszuprobieren. Um größere Programme zu schreiben, ist IDLE aber nicht sehr geeignet. Da gibt es wirklich komfortablere Entwicklungsumgebungen.

Und wie heißt wohl der Nachfolger (oder die Erweiterung; oder Verbesserung) von IDLE? Na, blättern wir zum Anfang des vorigen Kapitels, dort findet sich doch ein kleiner Tipp. Hast Du es rausgekriegt?

Die mächtigere Entwicklungsumgebung ist „Eric“. Und sie sieht so aus:

Abbildung 4.1. Die IDE Eric



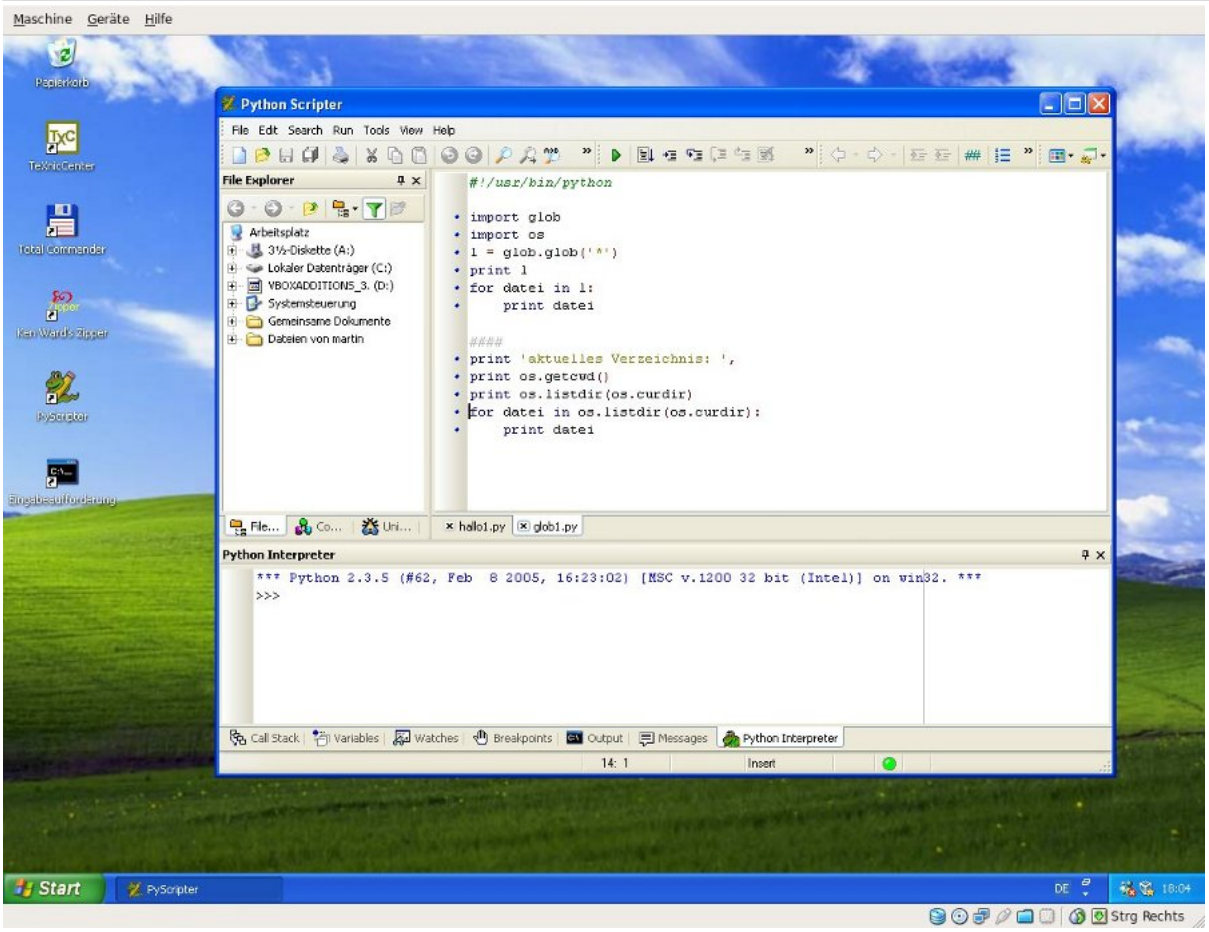
In „Eric“ können wir uns zum ersten Mal **richtig** daran machen, ein Programm zu schreiben. Das „richtig“ soll heißen, dass wir im Editor von Eric den Programmtext (auch Quelltext oder Quellcode genannt) schreiben, in einer Datei speichern, und jetzt außerhalb von Eric auch diese Datei aufrufen können.

4.1.2. IDE unter Windows

Die Installation von Eric unter Windows setzt eine Bibliothek voraus, die separat installiert werden muss.

Unter Windows ist deswegen eine andere IDE sehr verbreitet, der „PyScripter“. Der sieht so aus:

Abbildung 4.2. Die IDE PyScripter



Für kleinere Arbeiten existiert eine IDE unter Windows, die schlank und schnell ist und für Anfänger ausreichend, die „Geany“. Allgemein gilt hier: die IDE ist die beste, mit der ich am besten arbeiten kann. Und die Geschmäcker sind verschieden.

4.1.3. So sieht's aus

Programmier-IDEs haben alle einen sehr ähnlichen Aufbau. Es gibt immer 4 große Bereiche in einer solchen IDE, und bei manchen IDEs gibt es dann noch mehrere kleine Bereiche. Am oberen Rand der IDE ist eine Menu-Zeile, darunter eine Button-Zeile für häufig benötigte Operationen. Links darunter ist ein Bereich, in dem das Dateisystem abgebildet ist, so dass man hier navigieren und eine Datei auswählen kann. Rechts daneben ist der Editor-Bereich, in dem man seinen Quelltext schreibt. Für diesen Editor gibt es natürlich auch Buttons, mit denen man Editier-Operationen durchführen kann. Es ist aber sehr sinnvoll, sich die wichtigsten Tastatur-Befehle zu merken, denn das geht meistens viel schneller![1] Unter diesen beiden Bereichen ist dann der Ausgabe-Bereich, in den die Programme ihre Ausgabe schreiben. Unter Python ist dieser Ausgabebereich eine interaktive Python-Shell, in der man auch einzelne Befehle eingeben und testen kann.

4.1.4. Der Programm-Rahmen

Weil Python-Programme von der Syntax her portabel sind, das heißt ohne Veränderung des Quellcodes auf einem beliebigen Rechner (auf dem natürlich der Python-Interpreter installiert ist!) unter ei-

[1](Ein Bild, das diese veranschaulicht, findet man unter <http://www.kotzendes-einhorn.de/blog/wp-content/uploads/2011/02/guttenberg-tastatur1-600x440.jpg>.)

nem beliebigen Betriebssystem lauffähig sind, sollten wir uns hier bemühen, das Programm auch so zu schreiben, dass das gewährleistet ist.[2] Dazu gewöhnt man sich an, die unter Unix und MacOS übliche „sha-bang"-Zeile einzufügen, die dem Betriebssystem sagt, mit welcher Art von Programm (hier also: mit dem Python-Interpreter) die Datei zu bearbeiten ist und wo sich der Interpreter befindet:

Beispiel 4.1.1 Sha-Bang

```
#!/usr/bin/python
```

Der ist unter Unixen normalerweise im Verzeichnis /usr/bin . Aber leider nicht immer, deswegen ist die bessere Methode die, dass man das Betriebssystem auffordert, den Interpreter selber zu suchen:

Beispiel 4.1.2 Programmaufruf mit Umgebungsvariable

```
#!/usr/bin/env python
```

Hier wird das env-Kommando benutzt, das in der Umgebung (dem Environment) des Betriebssystems nach dem Interpreter sucht.

Und diese Zeile wird von den DOS-ähnlichen Betriebssystemen (also Windows) großmütig ignoriert, stört also in keinem Fall.

4.2. Das absolute Muss: Hallo Ihr alle

Es gibt ernsthafte Stimmen, die behaupten, dass es Unglück bringt, zum Beispiel 3 Tage lang Linsen mit Spätzle oder dass Bayern München schon wieder Deutscher Fußballmeister wird oder dass man die Blumen für den Hochzeitstag, an die man glücklich gedacht hat, geklaut bekommt, wenn man nicht als erstes Programm eine Begrüßung an den Rest der Welt schreibt: das berühmte „Hallo world". Also dann, das sieht in Python so aus:

Beispiel 4.2.1 Hallo world

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

print 'hallo world'
```

So einfach! Es ist keine Einbindung von Bibliotheken nötig (denke daran: batteries included!), keine Tricks, die den Anfänger verwirren.

In der Zeile nach dem sha-bang steht etwas, das wie ein Kommentar aussieht. Das ist aber eine Anweisung an den Interpreter, die ihm mitteilt, welche Kodierung das Programm benutzt. Hier ist die Unicode-Kodierung „utf8" eingestellt. Unter Windows [3] ist es oft sinnvoller, die Kodierung auf iso8859-1 oder latin1 umzustellen.

[2] Zu beachten hierbei ist allerdings, dass DOS (und damit Windows), Unix (und damit Linux) und MacOS verschiedene Zeichen benutzen, um den Zeilenvorschub zu kennzeichnen. Aber jedes der Betriebssysteme hat irgendein Hilfsmittel, um die Zeilenende-Zeichen zu übersetzen.

[3] wie immer zeitlich ein bißchen hinterher

WICHTIG

Unter Python 3.x hat sich das geändert. Hier wird als Standard-Kodierung „utf8“ angenommen, das heißt, dass man diese Zeile nur noch schreiben muss, wenn man auf einem System arbeitet, das noch nicht Unicode benutzt.[a]

[a] Also hoffentlich bald überhaupt nicht mehr.

4.3. Die ersten einfachen Programme

In einem Saftladen soll eine einfache Rechnung geschrieben werden. Das Programm dazu hat die Preise für die 3 Getränke, die zur Auswahl stehen, fest eincodiert. Vom Benutzer werden jetzt die konsumierten Getränke abgefragt, dann wird der Rechnungsbetrag ermittelt und ausgegeben.

Eine Eingabe durch den Benutzer wird über den Python-Befehl „input“ gemacht. Die Syntax lautet: variable = input(”Text zur Eingabeaufforderung”) „input“ nimmt Zahlen ordentlich entgegen, Texte hingegen müssen in Anführungsstriche geschrieben werden. Wer Texte benutzerfreundlicher bearbeiten will, sollte „raw_input“ nehmen, das Texte ohne Anführungsstriche entgegennimmt (dafür werden aber Zahlen nicht als Zahlen entgegengenommen, sondern als Text).

Beispiel 4.3.1 Saftladen

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

print "Saftladen\n\n"
prApfelschorle = 0.70
prBananenmilch = 1.20
prOrangensaft = 0.80

anzAschorle = input('Wieviel Apfelschorle? ')
anzBmilch = input('Wieviel Bananenmilch? ')
anzOsaft = input('Wieviel Orangensaft? ')

preis = anzAschorle * prApfelschorle + anzBmilch * prBananenmilch
        + anzOsaft * prOrangensaft
print 'Summe: ', preis
```

Im zweiten Programm werden zwei Minuten-Zahlen von Benutzern eingegeben, und diese werden in die üblichen Einheiten (x Stunden, y Minuten) umgerechnet:

Beispiel 4.3.2 Rechnen mit Uhrzeiten

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

min1 = input('erste Minuten-Zahl: ')
min2 = input('zweite Minuten-Zahl: ')

sum = min1 + min2
std = sum / 60
min = sum % 60

print std,' Stunde(n) ',min,' Minuten'
```

Teil III.

Texte und andere Daten

5. Texte

And now for something completely different

(John Cleese [1])

5.1. Grundlegendes zu Texten

Python kennt nicht den Datentyp des Zeichens; ein Zeichen ist einfach eine Zeichenkette der Länge 1. Für Zeichenketten ist der englische Ausdruck „strings“ üblich.

Strings sind unveränderlich (engl. immutable), d.h. sie können nicht am ursprünglichen Speicherplatz verändert werden. Die wichtigste Regel für Zeichenketten ist, dass Zeichenketten in Anführungszeichen geschrieben werden. Das ist bei Anfängern in vielen Programmiersprachen einer der häufigsten Fehler. Wir werden aber sehen, dass Python da hilfreich ist, so dass viele der üblichen Fehler schon beim Schreiben eines Programmes abgefangen werden.

In Python ist es egal, ob man einfache Anführungszeichen (auf der Tastatur über dem Lattenzaun # zu finden) oder doppelte Anführungszeichen (auf der Tastatur über der 2 zu finden) benutzt.

Ein Beispiel dazu:

```
>>> zKette = "Martin"
```

ist gleichwertig zu

```
>>> zKette = 'Martin'
```

Die Art der Anführungsstriche ist bedeutsam, wenn der Text selber Anführungsstriche bzw. Apostrophe enthält. Es funktioniert also

```
>>> zKette1 = "Zeig mir, wie's geht"
```

bzw.

```
>>> zKette2 = 'Gasthaus zum "Goldenen Ochsen" '
```

Dreifache Anführungszeichen haben eine besondere Bedeutung: sie begrenzen mehrzeiligen Text. Das wichtigste Beispiel (in Python) ist also:

```
>>>sinn = '''Always look
on the bright
side of life'''
```

Wenn wir das so in IDLE eingeben, ist es interessant zu sehen, wie das wieder ausgegeben wird. Es ist wieder ein Unterschied ob man diesen Spruch ausgibt, indem man einfach den Variablennamen

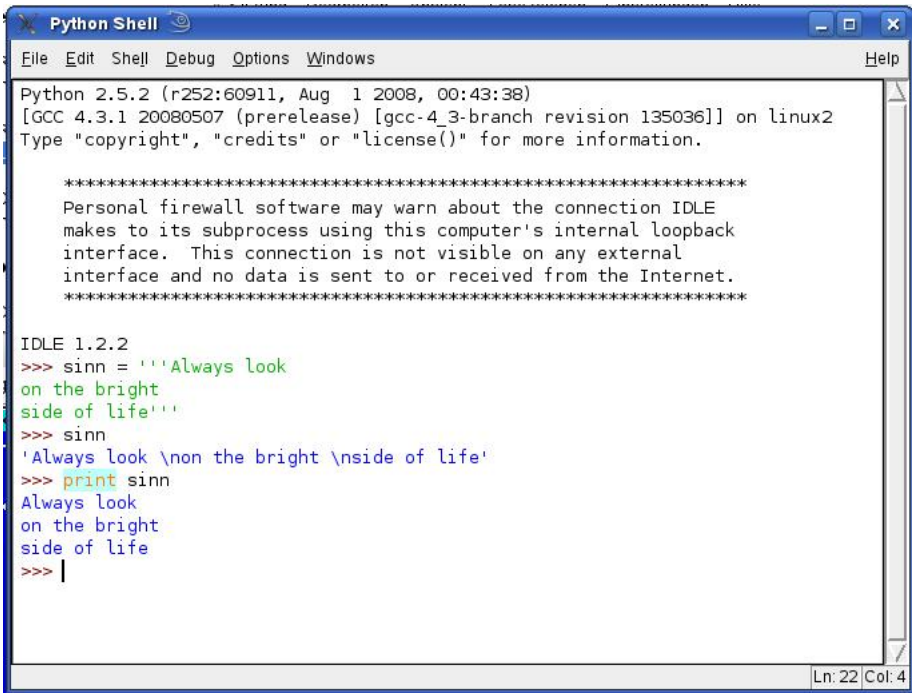
```
>>> sinn
```

[1] Ansage *in:* Monty Python's Flying Circus

eingibt, oder ob man IDLE ausdrücklich anweist, „sinn“ auszudrucken:

```
>>> print sinn
```

Abbildung 5.1. Texte in der IDE Idle



Man achte auch auf die Anführungsstriche in der Ausgabe!!

Die seltsame Zeichenkombination `\n` taucht hier zum ersten Mal auf. Das muss erklärt werden. In jedem Computer-System gibt es eine Anzahl Steuerzeichen. So nennt man Zeichen mit einer besonderen Bedeutung, die für die Steuerung des Rechners, genauer des Ausgabemediums, das in der Regel der Bildschirm ist, benötigt werden. Das wichtigste Steuerzeichen ist wohl das Zeichen für den Zeilenvorschub. Es wird mit `\n` codiert.

Beispiel 5.1.1 Zeilenvorschub (Steuerzeichen)

```
>>> vierZ = 'Das\nsind\n4\nZeilen'
>>> vierZ
'Das\nsind\n4\nZeilen'
>>> print vierZ
Das
sind
4
Zeilen
```

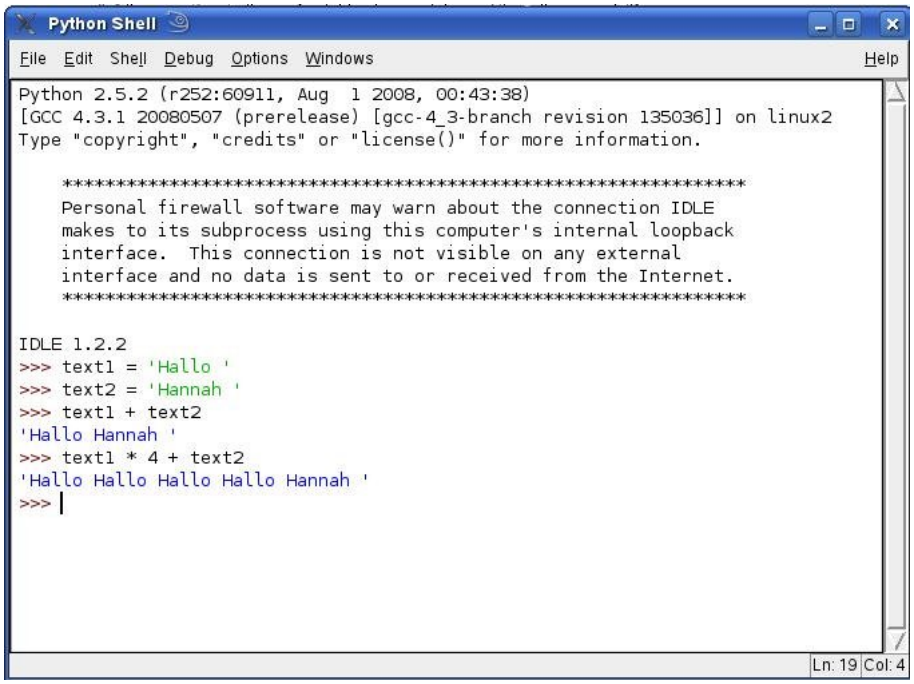
Die wichtigsten Steuerzeichen stehen in der folgenden Tabelle:

Python kann auch mit Texten rechnen. Was dabei herauskommt, wenn man Texte addiert oder einen Text mit einer Zahl multipliziert, sieht man hier

Tabelle 5.1. Steuerzeichen

Zeichen	Bedeutung
\a	Piepser (alarm)
\b	ein Zeichen zurück (backspace)
\f	Seitenvorschub (forward 1 page)
\n	Zeilenvorschub (new line)
\r	Wagenrücklauf (return)
\t	Tabulator

Abbildung 5.2. Multiplikation von Texten



So kann man also ganz schnell mal den Bildschirm von IDLE löschen:

```
>>> print 40*"\n"
```

5.2. Operationen auf Texten

And the first one now
Will later be last

(Bob Dylan[2])

Texte, also Zeichenketten, sind aus einzelnen Zeichen zusammengesetzt. Die dafür übliche Datenstruktur ist das Feld, auf englisch „array“. Ein Feld ist dadurch gekennzeichnet, dass jedes Element des Feldes eine Hausnummer hat. Zu beachten dabei ist, dass Informatiker oft auch Mathematiker sind, weswegen niemand von Hausnummer, sondern jeder von „Index“ redet und die erste Hausnummer, also der erste Wert für den Index, nicht die „1“ ist, sondern die „0“. Wenn wir also eine typische Zeichenkette betrachten

```
>>> FilmDerWoche = 'Das Leben des Brian'
```

dann kann man das als Feld so betrachten:

[2]The times they are a-changing *auf:* The times they are a-changing

Tabelle 5.2. Zeichenkette als Feld betrachtet

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Wert	D	a	s		L	e	b	e	n		d	e	s		B	r	i	a	n

Selbstverständlich kann man somit auch jedes einzelne Element der Zeichenkette ansprechen:

Beispiel 5.2.1 Elemente eines String

```
>>> FilmDerWoche = 'Das Leben des Brian'
>>> print FilmDerWoche[2]
s
>>> print FilmDerWoche[5]
e
```

Damit hat man schon das wichtigste Prinzip für die Bearbeitung von Texten verstanden! Man kann jeden Text als Aneinanderreihung von Zeichen auffassen, und jedes einzelne Zeichen herausfiltern, indem man den Namen der Zeichenkette gefolgt von der Hausnummer in eckigen Klammern angibt.

Dieses Prinzip wird jetzt aufgebohrt: man will ja oft nicht nur einen Buchstaben herausfiltern, sondern einen ganzen Bereich. Der technische Begriff dafür in Python lautet „Slicing“. Der gewünschte Teilbereich wird einfach in eckigen Klammern in der Form `Anfangs-Element : End-Element` angegeben, wobei zu beachten ist, dass das Anfangs-Element mit ausgewählt wird, das Endelement das erste Element ist, das nicht mehr ausgewählt wird.

Beispiel 5.2.2 Slicing

```
>>> FilmDerWoche = 'Das Leben des Brian'
>>> print FilmDerWoche[5:9]
eben
>>> print FilmDerWoche[2:6]
s Le
```

Eine wichtige Methode von Zeichenketten ist das Splitten. Eine Zeichenkette kann so an einem bestimmten Zeichen aufgespalten werden. Das Ergebnis ist eine Liste von Zeichenketten (siehe weiter hinten bei Listen). Das als Standard eingestellte Trennzeichen ist das Leerzeichen. Das soll an einem Beispiel verdeutlicht werden.

Beispiel 5.2.3 Splitting

```
>>> name = 'Martin Schimmels Oberndorf'
>>> name.split()[1]
'Schimmels'
>>> name.split()[-1]
'Oberndorf'
>>> name.split()[-2]
'Schimmels'
>>>
```

Hier wird also wirklich an den Leerzeichen getrennt. **Das Trennzeichen kommt dann in keiner der Teillisten vor.** Die einzelnen Teile kommen in eine Liste, und damit kann man die einzelnen Elemente der Liste mit ihrer Hausnummer ansprechen (Beachte: das Zählen fängt bei 0 an!!). Wenn man als Index eine negative Zahl benutzt, wird von hinten gezählt!

Weitere Operationen auf Texten sind das Finden und das Ersetzen. Erinnere Dich an das oben gesagte: Texte sind unveränderbar. Ein Ersetzen ändert den Inhalt der Variablen nicht. Aber das veränderte Stück Text kann einer (anderen) Variablen zugewiesen werden. Das sieht so aus:

Beispiel 5.2.4 Suchen und ersetzen

```
1
2  >>> name = 'Martin Schimmels Oberndorf'
3  >>> name.find('Schi')
4  7
5  >>> name.replace('Oberndorf', 'Rottenburg')
6 'Martin Schimmels Rottenburg'
7  >>> name
8  'Martin Schimmels Oberndorf'
9
```

In Zeile 3 wird als Ergebnis des Suchens die Zahl 7 ausgegeben. Das heißt, dass das Programm den gesuchten Text an der 7. Stelle des Namens gefunden hat. In Zeile 5 wird der Name ausgegeben mit der Ersetzung, aber in Zeile 7 sieht man, dass der Wert der Variablen `name` noch unverändert ist.

Auch die Untersuchung, ob ein Textstück in einem Text enthalten ist, kommt relativ häufig vor. Das geschieht über das Schlüsselwort **in**.

Beispiel 5.2.5 Enthaltensein

```
1
2  >>> name = 'Martin Schimmels Oberndorf'
3  >>> if 'mm' in name:
4  >>>    print 'gefunden'
5        gefunden
6
```

Weitere Methoden, mit denen man mit Texten arbeiten kann, werden hier unten als Befehl in eine Shell eingegeben. Die Ausgabe spricht für sich selbst.

Beispiel 5.2.6 Mehr mit Texten

```
1
2  >>> name = 'Martin Schimmels Oberndorf'
3  >>> name.upper()
4  'MARTIN SCHIMMELS OBERNDORF'
5  >>> name.lower()
6  'martin schimmels oberndorf'
7  >>> name.center(40)
8  '       Martin Schimmels Oberndorf       '
9  >>> name.center(40,'*')
10 '*******Martin Schimmels Oberndorf*******'
11
```

Weiter oben in diesem Kapitel wurden die Steuerzeichen angesprochen. Während es in der Unix-Welt fast keine Probleme mit Backslashes gibt, hat Microsoft leider dieses Zeichen als Pfad-Trenner missbraucht.[3] Das bringt Probleme mit sich, wenn man aus einem Programm heraus auf einen vollqualifizierten Dateinamen zugreifen muss, also auf etwas in der Art

```
C:\Benutzer\Texte\Liebesbriefe
```

Da gibt es die Möglichkeit, den Backslash durch einen Doppelbackslash zu maskieren. Das macht ein Programm nicht unbedingt lesbar. Die andere Möglichkeit ist, einen Text, der wie der obige Backslashes

[3] ganz zu schweigen von dem mittleren Blödsinn, den uns manche weismachen wollen, dass in Dateinamen, Verzeichnisnamen etc. Leerzeichen und Sonderzeichen auftreten dürfen

enthält, als „raw string“ zu codieren. Raw strings schalten die Interpretation des Backslashes als Steuerzeichen aus und interpretieren ihn als normales Zeichen. Das geschieht durch ein Voranstellen eines „r“. Beispiel:

```
verzeichnis = r'C:\Benutzer\Texte\Liebesbriefe'
```

5.3. Kodierungen

Die Umsetzung von Zeichen in Maschinenzahlen (die berühmten 01000111 - Folgen) kann selbstverständlich auf verschiedene Art erfolgen. Dies ist eine Frage der Vereinbarung. Übliche Vereinbarungen sind der ASCII [4] , der EBCDIC [5] , beide aus der Steinzeit der Datenverarbeitung. Das Problem dieser alten Kodierungen ist, dass sie nur eine begrenzte Menge von Zeichen darstellen können. Aus diesem Grund hat man sich Kodierungen überlegt, die mehr Zeichen aufnehmen können. Zuerst hat man beim ASCII, der ursprünglich nur 128 Zeichen aufnehmen konnte, den Platz verdoppelt. Mit westeuropäisch-nordamerikanischer Arroganz dachte man, damit alle Probleme gelöst zu haben: 26 Großbuchstaben, 26 Kleinbuchstaben, ein paar Satzzeichen, die 10 Ziffern und dann für Französisch noch ein paar Vokale mit Akzenten, für Deutsch noch ein paar Umlaute, da reichen doch 256 Zeichen.

Schnell hat man dann aber gemerkt, dass es so nicht geht. Kyrillisch, griechisch, türkisch, alle diese Sprachen haben wieder eigene Zeichen. Also hat man für Sprachfamilien (wie zum Beispiel für die nordeuropäischen Sprachen oder für die südosteuropäischen Sprachen) eigene Kodierungen erstellt, die dann normiert wurden unter „iso8859-x“. Dabei war x eine Zahl zwischen 1 und 15. Immerhin konnte man dann in jeder Sprache die üblichen Zeichen darstellen, aber wenn man die Sprache (bzw. die Sprachfamilie) verließ, musste man auch eine neue Kodierung auswählen.

Die Lösung heißt Unicode. Hier hat man Platz geschaffen, um auch Sprachen mit wesentlich mehr Zeichen (wie zum Beispiel Chinesisch) aufzunehmen und dabei alles in eine Kodierung zu packen.

5.4. Formatierung von Zeichenketten

Ziele der Formatierung von Zeichenketten sind:

1. Die Ausgabe soll schön, ordentlich, übersichtlich, lesbar ... sein
2. Der Ausgabe-Befehl soll variabel sein.

Die Regeln für die formatierte Ausgabe sind nicht schwer zu merken: das alleinstehende Prozentzeichen trennt die Formatierungsanweisung von den zu formatierenden Variablen. Diese Syntax lehnt sich an die Syntax der C-ähnlichen Sprachen an, wo eine solche Formatierung beim **printf**-Befehl verwendet wird. Hier noch formal:

```
print FORMATIERUNGSANWEISUNG % (VAR1, VAR2, etc.)
```

Nehmen wir an, wir wollen einen Gruß ausgeben, in den ein Name eingefügt werden soll.

Beispiel 5.4.1 Formatierungsanweisungen

```
>>> du = 'Hannah'
>>> print 'Guten Tag, %s, wie geht es Dir?' % du
Guten Tag, Hannah, wie geht es Dir?
```

Zuerst wird die Variable `du` deklariert, dann wird ein print-Befehl ausgegeben, der mit dem Text „Guten Tag,“ beginnt. Es folgt der Platzhalter für die Variable, versehen mit einem Formatierungsbefehl: das erste `%`-Zeichen steht für die erste zu formatierende Variable; das darauffolgende `s` gibt an, dass es sich bei der zu formatierenden Variablen um einen String, eine Zeichenkette, handelt. Es folgt ein

[4] ASCII: American Standard Code for Information Interchange
[5] Extended Binary Coded Decimals Interchange Code

weiteres Stück Text, worauf sich das `%`-Zeichen, das Formatierungsanweisung von zu formatierenden Variablen trennt, anschließt. Da wir in diesem Beispiel nur eine Variable haben, steht dort also nur die Variable `du`.

ACHTUNG

In Python ab Version 3.x ist die Formatierung mit Hilfe der C-ähnlichen Syntax, also der oben beschriebenen „%-Anweisungen“ entweder verpönt oder verboten (je nach Version). Hier funktioniert die Formatierung mittels der **format**-Methode von Strings.

Ein weiteres Beispiel folgt, diesmal mit zwei Variablen:

Beispiel 5.4.2 Formatierung mit 2 Parametern

```
>>> mann = 'Romeo'
>>> frau = 'Julia'
>>> print 'Berühmte Liebespaare: %s und %s' % (mann, frau)
Berühmte Liebespaare: Romeo und Julia
>>> m = 'Loriot'
>>> f = 'Evelyn Hamann'
>>> print 'Berühmte Liebespaare: %s und %s' % (m, f)
Berühmte Liebespaare: Loriot und Evelyn Hamann
```

Das ganze wird interessant, wenn man auch Zahlen formatiert ausgeben will. Hier folgt eine Liste der Format-Codes:

Tabelle 5.3. Format-Codes

Code	Bedeutung
%s	Zeichenkette (oder beliebiges anderes Objekt)
%i	Integers (ganze Zahlen)
%f	Fließkommazahl
%e	Fließkommazahl in Exponentialschreibweise

Die Format-Codes für Zahlen können noch Optionen bekommen.

Tabelle 5.4. Format-Option-Codes

neue Option	Beispiel	Erläuterung
. (dez.Punkt)	%4.2f	Die Zahl wird in einer Feldbreite von 4 Zeichen davon 2 Nachkommastellen ausgegeben.
0 (vorangestellt)	%06.2f	Die Zahl wird in einer Feldbreite von 6 Zeichen davon 2 Nachkommastellen und mit führenden Nullen ausgegeben.
- (Minuszeichen)	%-6.2f	Die Zahl wird in einer Feldbreite von 6 Zeichen davon 2 Nachkommastellen linksbündig ausgegeben.
+ (Pluszeichen)	%+6.2f	Die Zahl wird in einer Feldbreite von 6 Zeichen mit 2 Nachkommastellen und mit Vorzeichen ausgegeben.

5.5. Reguläre Ausdrücke

Reguläre Ausdrücke, auf englisch „regular expressions“, deswegen auch manchmal einfach mit „re“ abgekürzt, sind Ausdrücke, die Regeln gehorchen! Reguläre Ausdrücke sind eigentlich eine Program-

miersprache in der Programmiersprache, oder wie ein Kollege formuliert hat: die einzig wichtige Programmiersprache sind Reguläre Ausdrücke. Angenommen, Du suchst in einem Text nach einem Herrn Maier oder Herrn Meier oder Herrn Majer ... also genau das, was täglich passiert: wie schreibt sich der Herr bloß? Wenn Du jetzt nur suchen könntst nach einem Herrn, dessen Nachname mit einem „M“ anfängt, worauf einer der Buchstaben „a“ oder „e“ folgt, darauf einer der Buchstaben „i“, „y“ oder „j“, und dessen Nachname mit „er“ endet! Das kannst Du! Denn oben hast Du gerade eine Regel festgelegt, nach der der Ausdruck „Nachname“ aufgebaut sein soll. Versuchen wir es also:

Hier soll nur ein kurzer Anriss des Themas gemacht werden. Wer sich weiter informieren will, dem sei das Buch von [Friedl] empfohlen.

5.5.1. Allgemeines zu regulären Ausdrücken

Reguläre Ausdrücke in Python beginnen sinnvollerweise mit einem „r“, der Schreibweise für „raw strings“. Raw strings sind besondere Strings, die von Python so genommen werden, wie sie geschrieben sind. In diesem Fall dient das dazu, dass Backslashes nicht als Steuerzeichen in Escape-Folgen interpretiert werden. Der Raw string wird dann in Anführungsstriche eingeschlossen. Dieser raw string ist das Muster, nach dem wir suchen, der reguläre Ausdruck.

Damit wir diesen Ausdruck aufbauen können, benötigen wir außer den Regeln, nach was wir suchen die Vereinbarungen, was einzelne Zeichen in einem regulären Ausdruck für eine Bedeutung haben. Hier folgen zuerst einmal die Regeln, die wir für unser Problem benötigen:

- ein beliebiges Zeichen des aktuellen Zeichensatzes bedeutet genau dieses Zeichen. M bedeutet M, 3 bedeutet 3.
- beliebige Zeichen in eckigen Klammern bedeuten, dass eines der Zeichen in der eckigen Klammer genommen werden darf. [ae] bedeutet also, dass an dieser Stelle entweder ein a oder ein e stehen darf.

Das reicht schon, um unser Problem zu lösen. Zuerst muss aber ein Modul eingebunden werden, das sich mit regulären Ausdrücken befasst. Das geschieht mit der Anweisung „import re“. Leider muss hier vorgegriffen werden, denn Module werden erst in einem späteren Kapitel behandelt.

Beispiel 5.5.1 Reguläre Ausdrücke

```
>>> import re
>>> mText = "Unser Kandidat ist Herr Mayer.
                                  Er ist nicht zu verwechseln mit Frau Meyer"
>>> re.findall(r"M[ae][ijy]er",mText)
['Mayer', 'Meyer']
```

Notwendig ist der Import des Moduls re

Tabelle 5.5. Reguläre Ausdrücke: Zeichen

Zeichen	Bedeutung
Buchstabe, Zahl	genau dieses Zeichen
[abc]	eines der Zeichen in der eckigen Klammer
[a-z]	eines der Zeichen zwischen a und z
.	ein beliebiges Zeichen (außer Zeilenvorschub)
*	0 bis unendlich viele Wiederholungen des vorigen Zeichens / der vorigen RE
+	1 bis unendlich viele Wiederholungen des vorigen Zeichens / der vorigen RE
?	0 bis 1 Wiederholungen des vorigen Zeichens / der vorigen RE
^	Anfang der Zeichenkette
$	Ende der Zeichenkette
\d	[0-9]
\w	[0-9a-zA-z_]
\s	Leerzeichen, Tabulator, Zeilenvorschub

Tabelle 5.6. Reguläre Ausdrücke: Methoden

Name	Wirkung
re.compile(Muster)	gibt eine RE zurück
re.findall(Muster, String)	findet alle Vorkommen von Muster in String
re.sub(Muster, Ersetzung, String)	ersetzt alle Muster in String durch Ersetzung

5.5.2. Beispiel für Zahlen

In einem zweiten Beispiel soll eine Zahleingabe auf korrekte Schreibweise überprüft werden. Korrekt soll in unserem Fall heißen, dass eine Zahl eine Ganzzahl wie 17 oder eine Dezimalzahl mit maximal 2 Nachkommastellen sein soll. Ferner soll erlaubt sein, dass vor der Zahl ein Plus- oder ein Minuszeichen stehen darf. Nicht erlaubt ist das, was Mathematiker (vor allem aus dem englischsprachigen Raum) gerne machen, nämlich eine Dezimalzahl zwischen 0 und 1 ohne die 0 vor dem Komma zu schreiben, also zum Beispiel .75, so wie das ein Taschenrechner (oder auch Python) akzeptiert. Dazu ist zuerst die Zahl, die man eingibt, in einen Text umzuwandeln, denn reguläre Ausdrücke untersuchen Texte. Der reguläre Ausdruck wird Stück für Stück aufgebaut.

korrekteZahl = re.compile("'[0-9]+"') akzeptiert eine Ganzzahl die aus den Ziffern 0 bis 9 besteht (`[0-9]`), von denen mindestens eine existieren muss (`+`).

korrekteZahl = re.compile("'[0-9]+\."') akzeptiert eine Zahl, die aus den Ziffern 0 bis 9 besteht und durch einen Dezimalpunkt abgeschlossen wird. Der Dezimalpunkt muss durch einen Backslash maskiert werden.

korrekteZahl = re.compile("'([-+]?[0-9]+\.[0-9]{0,2})"') akzeptiert eine Dezimalzahl, die aus den Ziffern 0 bis 9 besteht und nach dem Dezimalpunkt wenigstens 0 und höchstens zwei weitere Ziffern aus der Menge 0 bis 9 enthält (`[0-9]{0,2}`). Aber das reicht noch nicht für eine gute Prüfung, denn ist noch erlaubt, dass ein Dezimalpunkt steht ohne Dezimalziffern danach.

Zusätzlich wird hier noch festgelegt, dass eine Zahl eine ganze Zeile ausfüllen muss. Das `^` bedeutet, dass `re` am Beginn der Zeile anfängt zu arbeiten, das `$`, dass `re` am Ende der Zeile aufhört.

```
korrekteZahl = re.compile(r'''(^[-+]?[0-9]+(\.[0-9]{1,2})?$)''')
```

Der zweite Teil der Zahl, nämlich das, was hinter dem Dezimalpunkt stehen darf, ist hier durch runde Klammern geblockt worden, so dass dieser Teil als ein Ausdruck aufgefasst wird (`(\.[0-9]{0,2})?`). Für diesen Block wird dann wieder ein Quantor benutzt, nämlich das `?`, das die Bedeutung hat: entweder kein Mal oder ein Mal. In diesem Fall also darf eine mit einem Punkt beginnende und durch 2 Ziffern beendete Zeichenfolge kein Mal oder ein Mal auftreten.

5.5.3. Ein komplizierterer Text

Dri Chinisin mit dim Kintribiss

—Kinderlied

Ein klassisches und deswegen ganz wichtiges Beispiel (hör Dich mal im Kindergarten Deines Wohnortes um!) ist der philosophische Text

Drei Chinesen mit dem Kontrabass
saßen auf der Strasse und erzählten sich etwas
kam ein Polizist "ei was ist denn das"
"Drei Chinesen mit dem Kontrabass"

in dem alle (unterschiedlichen) Vokale und Diphtonge durch ein und denselben Vokal ersetzt werden sollen. Dieses uralte Menschheitsproblem wird locker gelöst mit dem Programm

Beispiel 5.5.2 Drei Chinesen ... (reguläre Ausdrücke)

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4   # importiere das Modul "Reguläre Ausdrücke"
5   import re
6
7   vokalNeu = raw_input('Welcher Vokal soll es denn sein? ')
8
9   text = '''Drei Chinesen mit dem Kontrabass
10  saßen auf der Strasse und erzählten sich etwas
11  kam ein Polizist "ei was ist denn das"
12  "Drei Chinesen mit dem Kontrabass" '''
13
14  # regulären Ausdruck aufbauen, das ist ein "raw string"
15  suchString = re.compile(r"""(ei|au|ä|a|e|i|o|u)""")
16  neutext = suchString.sub(vokalNeu,text)
17  print neutext
18
```

Beachte dabei, dass die Diphtonge vor den Vokalen stehen müssen. Wenn die Diphtonge erst danach stünden, würde etwa in „ei“ zuerst das „e“, dann das „i“ ersetzt, wodurch aus dem „ei“ ein „oo“ würde (wenn man als Ersetzungs-Vokal das „o“ angibt).

Und das kommt dabei raus, wenn man als Ersetzungs-Vokal das „o“ angibt:

Beispiel 5.5.3 Dro Chonoson

```
Dro Chonoson mot dom Kontroboss
soßon of dor Strosso ond orzohlton soch otwos
kom on Polozost "o wos ost donn dos"
"Dro Chonoson mot dom Kontroboss"
```

Mit re.compile wird ein regulärer Ausdruck aufgebaut, der hier alternativ alle Vokale und Diphtonge enthält. Und der Rest ist wieder klar. (Hoffentlich!)

Eine Suche per **re.match** gibt ein Match-Object zurück. Jetzt kann es vorkommen, dass man genau mit diesem Objekt weiter arbeiten muss. Was nun? Hier kommt die Hilfe zur Selbsthilfe: man weist das Ergebnis des **match**-Aufrufs einer Variablen (z.B. der Variablen `ergMatch`) zu und ruft die Hilfe für dieses Objekt mit **help(ergMatch)** auf! Dann sieht man unter anderem, dass das Objekt `ergMatch` ein Attribut `string` hat. Das kann man ja mal ausdrucken!

6. Strukturierte Daten

6.1. Überblick

An strukturierten Daten unterscheidet man:

1. Listen
2. Dictionaries
3. Tupel

Der Unterschied zu einfachen Daten ist einfach: bei einfachen Daten hat eine Variable einen Namen und einen einzelnen Wert als Inhalt. Dieser Wert kann eine Zahl oder ein Text sein (und später werden wir sehen: noch mehr). Bei strukturierten Daten ist der Inhalt einer Variablen eine Menge von Werten. Eine Liste etwa kann eine Liste von Zahlen, eine Liste von Vornamen, eine Liste von Pizza-Angeboten Deines Lieblingsitalieners usw. sein. Ein Dictionary kann ein einsprachiges Wörterbuch sein, ein zweisprachiges Wörterbuch, ein Telefonbuch usw.

6.2. Listen

6.2.1. Definition von Listen und Listenelemente

Die einfachste Datenstruktur sind Listen. Listen sind geordnete Sammlungen von ... irgendwas. Das kann sogar wild durcheinander gehen: Zahlen, Texte, sogar wieder andere Listen. Das wichtigste daran ist, dass Listen-Elemente durchnumeriert werden.

WICHTIG

Beachte dabei: man fängt bei 0 an zu zählen!

Listen werden in eckige Klammern geschrieben, Listen-Elemente durch Kommata voneinander getrennt. Wenn man ein einzelnes Element einer Liste bearbeiten (oder anzeigen oder ...) möchte, geschieht das, indem man den Listennamen, gefolgt von der „Hausnummer“ in eckigen Klammern, angibt.

Hier folgen zwei Beispiele von Listen. Zuerst einmal eine typische Liste, nämlich eine Einkaufsliste:

Beispiel 6.2.1 Eine Einkaufsliste

```
einkaufsListe = ['Brot', 'Butter', 'Milch', 'Salz', 'Senf']
```

Listenelemente kann man einzeln ansprechen. Hier wird das erste Element der obigen Liste ausgegeben (denke daran: Python fängt bei 0 an zu zählen):

Beispiel 6.2.2 Listenelemente

```
>>> einkaufsListe[1]
'Butter'
>>> print einkaufsListe[0]
Brot
```

Und dann eine Liste mit ganz verschiedenartigen Elementen:

Beispiel 6.2.3 Eine einfache Liste

```
meineListe = ['Martin', 43, 'Mathematik']
```

Diese Liste hat drei Elemente, zwei Texte und eine Zahl. Und auch hier kann man die Listenelemente einzeln aufrufen:

Beispiel 6.2.4 Listenelemente

```
>>> meineListe[1]
43
>>> print meineListe[0]
Martin
```

Mit Listen kann man zum Glück noch viel mehr anstellen. Man kann Listen verketten. Das geschieht durch das Plus-Zeichen **+**

Beispiel 6.2.5 Verkettung von Listen

```
>>> meineListe = ['Martin', 43, 'Mathematik']
>>> kurzeListe = ['Karl','Egon','Uwe','Sepp']
>>> meineListe + kurzeListe
['Martin', 43, 'Mathematik','Karl','Egon','Uwe','Sepp']
```

Man kann auch eine Liste mit einer Zahl multiplizieren:

Beispiel 6.2.6 Rechnen mit Listen

```
>>> meineListe * 2
['Martin', 43, 'Mathematik', 'Martin', 43, 'Mathematik']
```

Man kann die Länge einer Liste feststellen:

Beispiel 6.2.7 Länge einer Liste

```
>>> print len(meineListe)
3
```

Immer wieder ist es aber auch wichtig, zu wissen, an welcher Stelle ein bestimmtes Element in einer Liste steht. Dazu dient die Operation **index** gemacht.

Beispiel 6.2.8 Position eines Listenelements

```
>>> print meineListe.index('Mathematik')
2
```

6.2.2. Veränderung von Listen

Etwas an eine bestehende Liste anzuhängen ist auch nicht schwer (aber es sieht prinzipiell anders aus als das Feststellen der Länge. Mehr dazu bei Klassen):

Beispiel 6.2.9 Verlängerung einer Liste

```
>>> meineListe.append('Lehrer')
>>> meineListe
['Martin', 43, 'Mathematik', 'Lehrer']
```

Entsprechend kann man auch etwas aus einer Liste entfernen, vorausgesetzt, man weiß, wo das gesuchte Element in der Liste steht.

Beispiel 6.2.10 Löschen eines Elements aus einer Liste

```
>>> meineListe
['Martin', 43, 'Mathematik', 'Lehrer']
>>> del meineListe[1]
>>> meineListe
['Martin', 'Mathematik', 'Lehrer']
```

Oben bei Zeichenketten als Feld haben wir schon den Begriff „slicing“ kennengelernt. Das funktioniert natürlich auch bei Listen.

Beispiel 6.2.11 Teile einer Liste

```
>>> langeListe = ['Martin', 43, 'Mathematik', 'Lehrer',
                                    'Banane', 1954, 'Senf']
>>> langeListe[2:5]
['Mathematik', 'Lehrer', 'Banane']
```

Und etwas, was oft sehr hilfreich ist:

Beispiel 6.2.12 Umkehrung einer Liste

```
>>> langeListe.reverse()
>>> print langeListe
['Senf', 1954, 'Banane', 'Lehrer', 'Mathematik', 43, 'Martin']
```

6.2.3. Tricks mit Listen

Ganz zu Anfang dieses Kapitels habe ich erwähnt, dass die Elemente einer Liste etwas beliebiges sein können. Darauf warten natürlich die Mathematiker schon lange: eine Matrix ist einfach eine Liste von Listen!

Beispiel 6.2.13 Matrix

```
>>> m1 = [[1, 0, 1],[-1,2,0],[0,-2,1]]
```

Allerdings ist die Darstellung bisher noch nicht so schön, damit müssen wir uns leider gedulden bis zum Kapitel über die Schleifen. (Wer spickeln möchte: Schleifen)

Es ist oft sehr hilfreich, dass die Elemente einer Liste durchnumeriert werden, manchmal ist es aber auch lästig. Nehmen wir einmal an, dass wir in einer Liste Daten zu einer Person, sagen wir Vornamen, Nachnamen und Schuhgröße speichern. So zum Beispiel:

Beispiel 6.2.14 Drei Listen

```
ich = ['Martin',  'Schimmels',  43]
hannah = ['Hannah',  'Schimmels',  40]
leni = ['Leni',  'Rein',  39]
```

Und nehmen wir weiter an, dass wir diese 3 Personen in einer weiteren Liste speichern wollen:

Beispiel 6.2.15 Eine Liste von Listen

```
allePersonen = [ich,  hannah,  leni]
```

Dann kann man den Vornamen der zweiten Person (denke daran: man fängt bei 0 an zu zählen! Die 2. Person in der Liste hat den Index 1!!) natürlich so holen:

Beispiel 6.2.16 Ein Element einer Liste von Listen (eigentlich ein Matrix-Element)

```
allePersonen[1][0]
```

Das ist aber gegen alle Gebote der Schönheit, Nachvollziehbarkeit, Transparenz. Besser wird das dadurch, dass man genau hinschaut und analysiert, wo was steht. Denn in der Liste jeder der Personen steht der Vorname immer an erster Stelle, hat also den Index 0. Also baut man sich ein Tupel, das als Inhalt die Begriffe Vorname, Nachname und Schuhgröße hat, und damit kann man jetzt viel eleganter auf die Vornamen zugreifen. Hier kommt ein kleiner Programmausschnitt:

Beispiel 6.2.17 Ein Trick bei der Listen-Bearbeitung

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   ich = ['Martin',  'Schimmels',  43]
6   hannah = ['Hannah',  'Schimmels',  40]
7   leni = ['Leni',  'Rein',  39]
8
9   allePersonen = [ich,  hannah,  leni]
10
11  vorname,  nachname,  schuhgroesse = range(3)
12
13  print allePersonen[0][vorname]
14
15  for einer in allePersonen:
16    print einer[vorname]+' '+einer[nachname]
17
```

Hier wird also in Zeile 11 ein Tupel mit den Zahlen 0,1,2 (Der Begriff range(3) tauchte bisher noch nicht auf. Der kommt erst im Kapitel über Schleifen; er bedeutet: alle ganzen Zahlen von 0 (inclusive) bis

3 (exclusive)) aufgebaut, das den Variablen Vorname, Nachname und Schuhgroesse zugeordnet wird. Zeile 11 bedeutet aufgeschlüsselt, dass der Variablen `vorname` der Wert „0“, der Variablen `nachname` der Wert „1“ und der Variablen `schuhgroesse` der Wert „2“ zugewiesen wird. Damit kann man auf die Teile der Liste, die eine Person beschreibt, über diese Begriffe zugreifen. Das ist doch viel besser zu lesen!

6.2.4. Kopie einer Liste

Immer wieder benötigt man von einer Liste eine Kopie. Was liegt näher, als folgendes zu versuchen:

Beispiel 6.2.18 Kopie einer Liste (so klappt es nicht!)

```
>>> meineListe = ['Apfel','Birne', 'Kohl', 'Gurke']
>>> kopie = meineListe
>>> print 'meine Liste = ', meineListe
>>> print 'Kopie = ', kopie
```

Das ergibt

Beispiel 6.2.19 Kopie einer Liste

```
meine Liste = ['Apfel','Birne', 'Kohl', 'Gurke']
Kopie = ['Apfel','Birne', 'Kohl', 'Gurke']
```

Und so wollten wir das doch … könnte man meinen!!! Aber Vorsicht! Hier wurde gar keine Kopie angelegt, wie man bei den nächsten Anweisungen sieht:

Beispiel 6.2.20 Kopie einer Liste (erster Versuch):

```
>>> meineListe[2] = 'Sauerkraut'
>>> print 'meine Liste = ', meineListe
>>> print 'Kopie = ', kopie
```

Und man liest

Beispiel 6.2.21 Listenkopie: so wollte ich es nicht!

```
meine Liste =  ['Apfel', 'Birne', 'Sauerkraut', 'Gurke']
Kopie =  ['Apfel', 'Birne', 'Sauerkraut', 'Gurke']
```

also etwas, was man gar nicht wollte. Was ist passiert? Durch die Einführung der Variablen `kopie` wurde dem Speicherbereich, der bisher den Namen `meineListe` hatte ein zweiter Name gegeben, aber tatsächlich keine Kopie erstellt. Das wird als „flache Kopie“ bezeichnet. Wenn ich wirklich eine Kopie erstellen will, dann muss ich das per Slicing machen: alle Elemente der ursprünglichen Liste in eine neue Liste kopieren. Aber das ist zum Glück nicht so schwer. Wir erinnern uns: `liste1[1:3]` bedeutete „alle Elemente der Liste vom ersten (inclusive) bis zum dritten (exclusive)“. Entsprechend bedeutet `liste1[2:]` „alle Elemente vom zweiten bis zum letzten“, `liste1[:3]` „alle Elemente vom Anfang bis zum dritten“, folglich `liste1[:]` „alle Elemente vom ersten bis zum letzten“.

Beispiel 6.2.22 Kopie einer Liste: aber jetzt!!

```
>>> meineListe = ['Apfel','Birne', 'Kohl', 'Gurke']
>>> kopie = meineListe[:]
```

Damit erstellt man ein neues Objekt, in das vom ersten bis zum letzten Element alles aus dem alten Objekt reinkopiert wird. Das kann man jetzt sehen, wenn man wieder aus dem Kohl in der einen Liste ein Sauerkraut macht.

Beispiel 6.2.23 Kopie einer Liste: na endlich hat es geklappt!

```
>>> meineListe[2] = 'Sauerkraut'
>>> print 'meine Liste = ', meineListe
>>> print 'Kopie = ', kopie
meine Liste =  ['Apfel', 'Birne', 'Sauerkraut', 'Gurke']
Kopie =  ['Apfel', 'Birne', 'Kohl', 'Gurke']
```

6.3. Dictionaries

A cat's meow and a cow's moo
You know I could recite them all

(Bob Dylan[1])

Dictionaries gibt es in vielen Programmiersprachen (nur heißen sie dort anders, nämlich „hashes" oder „assoziative Arrays"). Das charakteristische für Dictionaries ist, dass hier die Elemente nicht mehr durchnumeriert werden wie in Listen, sondern Werte unter einem Schlüssel abgelegt werden. Jeder Dictionary-Eintrag ist also ein Paar von Informationen, wobei die erste Information „Schlüssel" (engl. key), die zweite Information „Wert" (engl. value) genannt wird. Dictionaries werden in geschweifte Klammern geschrieben. Im ersten Dictionary soll tatsächlich ein Wörterbuch abgelegt werden.

Beispiel 6.3.1 Ein Dictionary (sogar ein Wörterbuch!)

```
englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh', 'sheep':'Schaf'}
>>> englDeutsch['dog']
'Hund'
>>> englDeutsch['cow']
'Kuh'
```

Der Schlüssel eines der Paare ist also hier das englische Wort für ein bestimmtes Tier, der Wert ist die deutsche Übersetzung dafür. Ein Element eines Dictionary wird also nicht mehr durch eine Hausnummer gefunden, sondern über den Schlüsselbegriff. Wichtig ist es manchmal, alle Schlüssel oder alle Werte eines Dictionary zu bekommen:

Beispiel 6.3.2 Schlüssel und Werte eines Dictionary

```
>>> englDeutsch.keys()
['sheep', 'dog', 'cow', 'cat']
>>> englDeutsch.values()
['Schaf', 'Hund', 'Kuh', 'Katze']
```

Eine kleine Anwendung von Dictionaries folgt, wenn Schleifen behandelt worden sind.

[1] Quinn the eskimo *auf:* Basement Tapes

6.4. Tupel

Tupel? Warum gibt es die? Denn Tupel sind eigentlich nur Listen, allerdings sind sie unveränderlich. Tupel werden in runde Klammern geschrieben. Und außerdem haben Tupel keine Methoden. Aber das sind schon die relevanten Unterschiede. Meistens wird man sich für Listen entscheiden, wenn man Objekte in einer Struktur ablegen will.

Eine ganz schöne Anwendung, und für Programmierer, die mit anderen Programmiersprachen groß geworden sind, ungewöhnlich, ist das Vertauschen von 2 Objekten. Der Pseudocode für eine solche Vertauschung sieht so aus (und so muss das auch in fast allen Programmiersprachen codiert werden):

```
temporaeresDing = ding1
ding1 = ding2
ding2 = temporaeresDing
```

In Python sieht der Programm-Code so aus, wie sich das der naive Programmierer vorstellt. Dabei wird benutzt, dass Python bei Zuweisungen die runden Klammern von Tupeln nicht benötigt:

```
ding1, ding2 = ding2, ding1
```

Wenn doch alles auf der (Programmierer-)Welt so einfach wäre!!

6.5. Zusammenfassung

Hier folgen noch mal die wichtigsten Eigenschaften der strukturierten Daten in einer Tabelle.

Tabelle 6.1. Überblick Strukturierte Daten

Name	andere Bezeichnung	Schreibweise	Daran erkennt man's!	veränderbar?
Liste	Array	L = ['Hans', 'Eva','Paul', 'pia']	eckige Klammern	ja
Dictionary	Hash	D = {'vorname':'Martin','nachname':'Schimmels', 'schuhgroesse':43, 'auto':'ja',}	geschweifte Klammern	ja
Tupel		t = (12, 'Monate', 3, 'Jahre')	runde Klammern	nein

6.6. Übungen zu Listen, Dictionaries ...

1. Schreibe ein Python-Programm, das eine Eiskarte darstellt. Die verschiedenen Angebote Deiner Eisdiele sollen in einem Dictionary abgelegt werden, das als Schlüssel den Namen des Eisbechers und als Wert den Preis enthält.
2. Eine Liste von Staaten soll erstellt werden, in der die Listenelemente wieder Listen sind. Gespeichert werden sollen der Ländername, die Hauptstadt und die Landessprache. Wie speichert man es sinnvoll ab, wenn ein Land wie z.B. die Schweiz mehrere Landessprachen hat?
3. Ein Adressbuch soll angelegt werden. Mit Adressen ist nicht etwas wie „martin@web.de“ gemeint, sondern so etwas altmodisches, was man auf Briefumschläge geschrieben hat, wie zum Beispiel:

   Frau
   Mathilde Weber
   Primus-Truber-Straße 123
   72072 Tübingen
   Deutschland

Das bietet sich an als ein Dictionary von Dictionaries. Überlege Dir sinnvolle Ausgaben dieser Datenstruktur.

Teil IV.

Strukturen

7. Programmstrukturen

Don't think twice, it's alright

(Bob Dylan[1])

7.1. Gute Programme, schlechte Programme

Bevor wir jetzt beginnen, richtige Programme zu schreiben, die eine gewisse Struktur haben, sollten wir uns zurücklehnen und uns vornehmen: wir schreiben nicht nur richtige Programme, sondern gute Programme. Wenn man ja nur wüßte, was ein gutes Programm ist!! Dass ein Programm die Aufgabe erfüllt, zu der man es geschrieben hat, ist eine Selbstverständlichkeit: ein solches Programm ist noch nicht gut, es funktioniert.

Damit es gut wird, muss es anpassbar sein, vielseitig verwendbar, gut lesbar, leicht änderbar, schön. Das alles wollen wir schaffen? Klar, und zwar gleich von Anfang an!

Deswegen kommt hier zuerst eine Regel, die uns etwas verbietet, richtig knallhart verbietet, denn so etwas geht gar nicht: **Ein Programm ist schlecht, wenn sich eine Zeile wiederholt.** Das soll auch positiv formuliert werden: **Damit ein Programm gut ist, darf eine Anweisung nur einmal auftauchen.** Das hört sich brutal an, aber es ist wirklich so: eine Anweisung darf nur einmal auftauchen! Wenn auch nur eine Anweisung mehrmals auftaucht, hat man bei der Konzeption, der Denkarbeit, bevor man sich an den Computer setzt und etwas eintippt, einen Fehler gemacht: einen Denkfehler.

Die Python-Programmierer haben dazu auch etwas veröffentlicht: „The Zen of Python". Das kann man nachlesen, indem man in der Python-Shell den Befehl **import this** eingibt. Do it!

7.2. Eins nach dem anderen: Die Sequenz

Bisher war Programmieren eher langweilig. Wir haben einzelne Befehle aufgeschrieben, und jeden dieser Befehle ausführen lassen. Eine solche Folge von Befehlen wird in der Programmierung als Sequenz bezeichnet. Trotzdem ist dies natürlich schon richtiges Programmieren. Ausreichend viele Programme tun schon etwas sehr vernünftiges, nur indem sequentiell Anweisungen abgearbeitet werden

Fazit zum Thema sequentielle Programme: Besser als nichts, aber nicht viel besser! Trotzdem könnte man dazu ein bißchen üben!

7.2.1. Aufgaben zu Sequenz

1. Falls Du es nicht mehr auswendig weißt, weil die Fahrschule schon zu lange hinter Dir liegt: informiere Dich, wie man den Anhalteweg eines Autos berechnet. Schreibe dann ein Programm, in das eine Geschwindigkeit eingegeben wird. Das Programm soll dann den Reaktionsweg, den Bremsweg und den Anhalteweg ausgeben.
2. Informiere Dich über die Herzfrequenz beim Training für eine optimale Fettverbrennung und für einen maximalen Konditionsaufbau. Schreibe dann ein Programm, in das man sein Alter eingibt. Das Programm soll den maximalen Puls für einen Menschen Deines Alters sowie die optimalen Frequenzen für Konditionsaufbau und Fettabbau ausgeben.
3. Der Body-Mass-Index soll durch ein Programm berechnet werden. Die Eingabe soll über „variable = input ..." erfolgen. Der BMI errechnet sich als Gewicht in kg geteilt durch das Quadrat der Größe in Metern.
4. Ein Umrechnungsprogramm soll geschrieben werden, das Meter in Fuß umrechnet. (1 Fuß = 30,48 cm)
5. Ein Umrechnungsprogramm Liter in Pint soll geschrieben werden.
6. Ein Programm soll sechs Zahlen einlesen und den Mittelwert dieser sechs Zahlen ausgeben.

[1] Don't think twice, it's alright *auf:* The Freewheeling Bob Dylan

7. In ein Programm soll der Radius eines Kreises eingegeben werden. Das Programm soll den Umfang und die Fläche des Kreises ausgeben.
8. Länge und Breite eines Rechtecks sollen eingegeben werden. Das Programm soll den Umfang, die Fläche und die Länge der Diagonalen des Rechtecks ausgeben.

7.3. Wenn . . . dann: Die Alternative

Die Sequenz gehört zu den Elementen der „Strukturierten Programmierung“. Weitere Elemente sind

- die Auswahl (auch „Alternative“)
- die Wiederholung (auch „Iteration“)

Alternative bedeutet, dass hier in eine Sequenz eingegriffen wird und von zwei Möglichkeiten eine ausgewählt wird.[2] Die Auswahl wird getroffen anhand einer oder mehrerer Bedingungen. Das soll in dem folgenden Beispiel (das nicht in einer Programmiersprache formuliert ist, sondern umgangssprachlich) demonstriert werden:

Beispiel 7.3.1 Entscheidung (if - then) (Pseudocode)

```
WENN die Person männlich ist:
   begrüße sie mit Herr Müller
SONST:
   begrüße sie mit Frau Müller
```

Die Bedingung fragt hier das Geschlecht der Person ab. Die Auswahl wird getroffen zwischen der Anrede „Herr“ und „Frau“.

Eine solche Schreibweise, bei der Wörter der Alltagssprache benutzt werden, aber eine Syntax, die sich an die Logik einer Programmiersprache anlehnt, wird „Pseudocode“ genannt. Und zu Psesudocode sagt Lutz [3]: „Because Python's syntax resembles *executable pseudocode*, it yields programs that are easy to understand, change, and use long after they have been written“ Der Code in Python sieht dann so aus:

Beispiel 7.3.2 Entscheidung (if - then) (Python)

```
if geschlecht == 'm':
  print 'Guten Tag, Herr Müller'
else:
  print 'Guten Tag, Frau Müller'
```

Zu den Vergleichsoperatoren siehe weiter vorne im Buch.

7.3.1. Blöcke und Einrückungen

An der Schreibweise des obigen Beispiels können wir aber gleich eine wesentliche Eigenschaft von Python lernen. Die Bedingung (also hier die Frage, ob das Geschlecht der Person männlich ist) wird auf ihren Wahrheitsgehalt geprüft. Die Bedingung kann erfüllt sein, das heißt, die Frage wird mit „ja“ beantwortet oder anders gesagt, die Bedingung bekommt den Wert „wahr“, oder die Bedingung ist nicht erfüllt, das heißt, die Frage wird mit „nein“ beantwortet oder anders gesagt, die Bedingung bekommt den Wert „falsch“. Die Aktion (oder die Aktionen), die in dem einen beziehungsweise im anderen Fall ausgeführt wird, ist von dem Wert der Bedingung abhängig. Man spricht von den „DANN-Anweisungen“ bzw. von den „SONST-Anweisungen“

[2] Alternative bedeutet tatsächlich nur eine Auswahl aus zwei Möglichkeiten; in der Umgangssprache (aber leider oft auch in der Fachsprache) wird Alternative oft als „Auswahl unter mehreren Möglichkeiten“ beschrieben.

[3] [Programming Python], Seite 5

Solche Abhängigkeiten müssen dem Programm, das wir schreiben wollen, mitgeteilt werden. Eine in vielen Programmiersprachen verwendete Methode ist es, den abhängigen Teil in besondere Zeichen, oft in Klammern, einzufassen. Python geht einen anderen Weg: In Python wird alles, was von etwas anderem abhängig ist, eingerückt. Dabei muss man beachten, dass alles, was von dem selben Programmteil abhängig ist, um die selbe Länge eingerückt wird. Noch dazu muss man aufpassen, dass man nicht Tabulatoren und Leerzeichen vermischt. Das soll hier durch eine Erweiterung des obigen Beispiels deutlich gemacht werden (und das Beispiel erklärt sich selber):

Beispiel 7.3.3 Verschachtelte Entscheidungen (Pseudocode)

```
WENN die Person männlich ist:
   WENN es vor 12 Uhr ist:
      begrüße sie mit 'Guten Morgen, Herr Müller'
   SONST:
      begrüße sie mit 'Guten Tag, Herr Müller'
SONST:
   WENN es vor 12 Uhr ist:
      begrüße sie mit 'Guten Morgen, Frau Müller'
   SONST:
      begrüße sie mit 'Guten Tag, Frau Müller'
```

7.3.2. Beispiel für eine Mehrfach-Entscheidung

Da sich Programmiersprachen an die englische Sprache anlehnen, sind die Schlüsselwörter der englischen Sprache entnommen. Das obige umgangssprachliche Beispiel ließe sich in Python so formulieren. Beachte dabei, dass als Abschluss der Bedingung so wie im umgangssprachlichen Text ein Doppelpunkt steht.

Beispiel 7.3.4 Verschachtelte Entscheidungen (Python)

```
if geschlecht == m:
  if stunde < 12:
    print 'Guten Morgen, Herr Müller'
  else:
    print 'Guten Tag, Herr Müller'
else:
  if stunde < 12:
    print 'Guten Morgen, Frau Müller'
  else:
    print 'Guten Tag, Frau Müller'
```

Wie sieht es aber aus, wenn ich mehrere Möglichkeiten zur Auswahl habe? Guten Morgen, guten Tag, guten Abend, gute Nacht!

Beispiel 7.3.5 Mehrere Möglichkeiten

```
if stunde < 10:
  print 'Guten Morgen'
else:
  if stunde < 18:
    print 'Guten Tag'
  else:
    if stunde < 21:
      print 'Guten Abend'
    else:
      if stunde <24:
        print 'Gute Nacht '
      else:
        print 'Wie bitte? Was sagt man denn nach Mitternacht? '
```

Das oben stehende Beispiel sieht aber nicht schön aus[4]. Die in zwei aufeinanderfolgenden Zeilen stehenden

```
  else:
    if
```

geben ja eine logische Ebene an. Der `else`-Zweig hat als erstes Konstrukt eine weitere Bedingung. Die auf dieses `else` folgende Anweisung ist, zusammen mit der oben drüber stehenden Anweisung und den darunter stehenden im Prinzip eine Auswahl unter mehreren Möglichkeiten. Deswegen gibt es in Python ein zusammengezogenes `else if`, nämlich das `elif`. Das obige Beispiel sieht besser so aus:

Beispiel 7.3.6 Verschachtelte Entscheidungen mit elif

```
if stunde < 10:
  print 'Guten Morgen'
elif stunde < 18:
  print 'Guten Tag'
elif stunde < 21:
  print 'Guten Abend'
elif stunde <24:
  print 'Gute Nacht '
else:
  print 'Wie bitte? Was sagt man denn nach Mitternacht? '
```

Es ist im Übrigen guter Stil, und man ist damit auch auf der sicheren Seite, wenn man eine solche Mehrfachauswahl mit einem `else` abschließt, damit auch alle vorher nicht ausdrücklich aufgelisteten Fälle behandelt werden.

7.3.3. Ein Trick für verschachtelte Entscheidungen

Und auch das ist noch nicht die elegante Variante. Leider steht aber in Python keine `switch`- oder `case`-Anweisung zur Verfügung wie in vielen anderen Programmiersprachen. Dafür gibt es aber Dictionaries. Das soll eine Mehrfach-Auswahl ersetzen? Doch, das geht wirklich!

[4] Wie oft habe ich das wohl schon geschrieben? Aber es ist wirklich so: wenn etwas nicht schön aussieht, dann ist es auch meistens nicht schön programmiert, und dann kann man das Programm (auf jeden Fall in Python!) ästhetischer schreiben, damit besser lesbar. Und das ist auch meistens die bessere Lösung.

Beispiel 7.3.7 Verschachtelte Entscheidungen getrickst mit Dictionary

```
preise = {'Milch':1.20, 'Apfelsaft':1.50, 'Orangensaft':1.60, 'Cola':2.20}

getraenk = raw_input('Bitte Getränk eingeben (Milch/Apfelsaft/
                                              Orangensaft/Cola)'
print preise[getraenk]
```

ist gleichwertig zu

Beispiel 7.3.8 Verschachtelte Entscheidungen zu Fuß

```
getraenk = raw_input('Bitte Getränk eingeben (Milch/Apfelsaft/
                                              Orangensaft/Cola)'
if getraenk == 'Milch':
  print 1.20
elif getraenk == 'Apfelsaft':
  print 1.50
elif getraenk == 'Orangensaft':
  print 1.60
elif getraenk == 'Cola':
  print 2.20
```

Die erste der beiden oben stehenden Varianten besticht durch ihre Kürze. Die zweite Variante ist dafür einfacher auf eine fehlerhafte Eingabe zu erweitern:

Beispiel 7.3.9 Verschachtelte Entscheidungen zu Fuß mit Fehlerbehandlung

```
getraenk = raw_input('Bitte Getränk eingeben (Milch/Apfelsaft/
                                              Orangensaft/Cola)'
if getraenk == 'Milch':
  print 1.20
elif getraenk == 'Apfelsaft':
  print 1.50
elif getraenk == 'Orangensaft':
  print 1.60
elif getraenk == 'Cola':
  print 2.20
else:
  print 'fehlerhafte Eingabe'
```

Um das mit Hilfe eines Dictionary zu erledigen, muss man die „get"-Methode von strukturierten Daten benutzen.

Beispiel 7.3.10 das selbe mit Dictionary

```
preise = {'Milch':1.20, 'Apfelsaft':1.50, 'Orangensaft':1.60, 'Cola':2.20}

getraenk = raw_input('Bitte Getränk eingeben (Milch/Apfelsaft/
                                              Orangensaft/Cola)'
print preise.get(getraenk, 'falsche Eingabe')
```

Die `get`-Methode erlaubt als zweiten Parameter einen Default-Wert, der hier auf „falsche Eingabe“ gesetzt wurde.

7.3.4. Übungen zu Auswahl

1. Punktprobe: In das Programm sollen Werte für die Steigung m und den y-Achsenabschnitt b einer Geraden sowie die x-Koordinate und die y-Koordinate eines Punktes eingegeben werden. Das Programm soll dann überprüfen, ob der Punkt auf der Geraden liegt und eine entsprechende Meldung ausgeben.
2. Zwei-Punkte-Form der Geradengleichung: In das Programm sollen vom Benutzer die x-Koordinaten und y-Koordinaten von 2 Punkten eingegeben werden. Das Programm soll die Geradengleichung in der Form y = mx + b ausgeben.
3. Die Signum-Funktion gibt den Wert 1 zurück, wenn die eingegebene Zahl positiv ist, den Wert 0, wenn die eingegebene Zahl 0 ist und den Wert -1, wenn die eingegebene Zahl negativ ist. Schreibe ein Programm!
4. Eine einfache Prüfziffer für 7-bit-Zeichen (den Standard-ASCII) könnte so funktionieren: wenn die Anzahl der Einsen ungerade ist, setze das 8. bit auf 1, wenn die Anzahl der Einsen gerade ist, auf 0.
5. Mitternachtsformel (oder für Nordlichter: die allgemeine Lösungsformel für quadratische Gleichungen) ist zu programmieren. Eingabe der Koeffizienten a, b, c der quadratischen Gleichung $ax^2 + bx + c = 0$ in das Programm mit Hilfe von „variable = input . . . “. Ausgabe von Diskriminante, Anzahl der Lösungen und gegebenenfalls der Lösungen.
6. Das Programm zur Berechnung des Body-Mass-Index aus Kapitel 4 soll dahingehend verbessert werden, dass es eine Bewertung ausgibt. Dabei gilt, dass Frauen mit einem BMI kleiner oder gleich 19 und Männer mit einem BMI kleiner oder gleich 20 untergewichtig sind, Frauen mit einem BMI größer oder gleich 24 und Männer mit einem BMI größer oder gleich 25 übergewichtig sind.
7. Ein Programm soll ausgeben, ob ein Jahr ein Schaltjahr ist. Beachte dabei, dass die durch 100 teilbaren Jahre keine Schaltjahre sind, außer wenn sie durch 400 teilbar sind.
8. Für die Berechnung des Wochentages hat C.F. Gauss folgende Formel aufgestellt (dabei ist die seltsame eckige Klammer, bei der der obere Haken fehlt, das Symbol für die Gauss'sche Klammerfunktion, die bedeutet: größte ganze Zahl kleiner oder gleich als das, was in dieser Klammer steht): $w = (d + \lfloor 2,6 \cdot m - 0,2 \rfloor + y + \lfloor \frac{y}{4} \rfloor + \lfloor \frac{c}{4} \rfloor - 2c) \mod 7$

   Die Variablen haben folgende Bedeutung:

   - d: Tagesdatum (1 bis 31)
   - m: (Monatsnummer (1 bis 12) - 2) mod 12
   - y: Die beiden letzten Stellen der Jahreszahl (für 2007 wäre diese 07 also 7)
   - c: Die beiden ersten Stellen der Jahreszahl (für 2007 wäre diese 20)
   - w: Wochentag gemäß unten angeführter Tabelle

7.4. Schleifen

God knows when
But you're doin' it again

(Bob Dylan [5])

Lies noch einmal nach, was weiter oben über gute Programme geschrieben wurde. Ein Programm der Art (das ist kein Python-Programm, sondern nur Pseudocode!)

Beispiel 7.4.1 so nicht!

```
programmieranweisung_fuer_den_wert_1
programmieranweisung_fuer_den_wert_2
programmieranweisung_fuer_den_wert_3
```

[5] Subterranean Homesick Blues*auf:*Bringing It All Back Home

geht also gar nicht. Mehrmals die selbe Anweisung, nur mit verschiedenen Werten: das muss zu vereinfachen sein.

Jetzt geht es also um die Wiederholung von einer Anweisung oder einer ganzen Gruppe von Anweisungen. Endlich lohnt sich das Programmieren!

Eine solche Wiederholung, Schleife genannt, kann prinzipiell auf zwei Arten realisiert werden:

- Man gibt die Anzahl der Wiederholungen an. In diesem Fall spricht man von einer Zählschleife. In der Programmierung nennt man das eine „for-Schleife".
- Man weiß nicht genau, wie oft etwas wiederholt werden soll, sondern man kann nur angeben, wann man mit der Wiederholung aufhören soll. Anders gesagt, man kennt die Abbruchbedingung für die Wiederholung. Dies wird als „while-Schleife" programmiert.

7.4.1. Zähl-Schleifen

Behandeln wir also zuerst die Zählschleife. In Python stellt man sich diese Art von Schleifen so vor: hier bekommst Du (die Schleife) eine Menge von Dingen, und mit jedem dieser Dinge mach etwas. Das kann eine Menge von Zahlen, eine Menge von Namen, eine Menge von irgendwelchen Objekten sein.

Bei einer Zählschleife gibt es eine Variable, die Buch darüber führt, wie oft etwas gemacht werden soll und wo man denn schon steht. Wenn die Menge von Dingen Zahlen sind, wird eine solche Variable „Zähler" genannt. Einem solchen Zähler muss man mitteilen, was der Anfangswert der Zählung sein soll, was der Endwert sein soll und mit welcher Schrittweite gezählt werden soll. Der einfachste Fall in Python ist

Beispiel 7.4.2 Zählschleife

```
>>> for zahl in range(4):
      print zahl
```

Das englische Wort „range" bedeutet Bereich, und hier ist also die Menge von Dingen der Bereich der natürlichen Zahlen bis 4. Hier finden wir wieder die Einrückung, die wir aus dem Kapitel über die Alternative kennen. Der print-Befehl ist abhängig von der Zählung; also muss er eingerückt werden und das, wovon abhängig ist, wird durch einen Doppelpunkt abgeschlossen. Das sollte man jetzt, sofern man vor seinem Rechner sitzt, selber probieren.

Beispiel 7.4.3 Einfache Zählschleife

```
>>> for zahl in range(4):
  print zahl
0
1
2
3
```

WICHTIG

Zwei Sachen muss man hier beachten. Zum einen startet die Zählung bei 0. Aber das sind wir ja schon fast gewohnt. Zum anderen gilt hier wie beim Slicing von Zeichenketten, dass der Anfangswert mit ausgegeben wird, der Endwert jedoch nicht.

Jetzt probieren wir gleich etwas mehr aus, indem wir der `range` 2 Werte mitgeben. Werte werden durch Kommata voneinander getrennt, wohingegen das deutsche Dezimalkomma in englischsprachigen Ländern zum Dezimalpunkt wird.

Beispiel 7.4.4 Zählschleife mit Anfang und Ende

```
>>> for zahl in range(5,8):
  print zahl
5
6
7
```

Man sieht: der erste Wert gibt den Anfang der Zählung, der zweite das Ende der Zählung an.

Nur Mut: `range` bekommt jetzt 3 Werte beigefügt

Beispiel 7.4.5 Zählschleife mit Anfang, Ende und Schrittweite

```
>>> for zahl in range(3,20,4):
  print zahl
3
7
11
15
19
```

Der dritte Wert gibt also die Schrittweite an. Also können wir endlich der NASA behilflich sein, und das Countdown-Programm schreiben:

Beispiel 7.4.6 Countdown

```
>>>  for zahl in range(10,0,-1):
  print zahl
10
9
8
7
6
5
4
3
2
1
```

Da nicht nur Zahlen gezählt werden können, wie wir inzwischen wissen, sondern auch Listen(-Elemente), kann man auch über Listen schleifen:

Beispiel 7.4.7 Schleifen über Listen

```
>>> kurzeListe = ['Karl','Egon','Uwe','Sepp']
>>> for wort in kurzeListe:
  print wort
Karl
Egon
Uwe
Sepp
```

Zählschleifen können aber auch benutzt werden bei Strukturen wie Zeichenketten, denn eine Zeichenkette ist (siehe oben bei String als Feld) im Prinzip nichts anderes als ein Feld, das durchnummeriert ist, und das man deswegen abzählen kann.

Beispiel 7.4.8 Schleifen über Zeichenketten

```
>>> name = "Martin"
>>> for buchstabe in name:
  print buchstabe
M
a
r
t
i
n
```

Das ist gut, aber noch besser wäre es, wenn alles in einer Zeile stünde. Auch das ist einfach zu realisieren, denn ich muss Python nur mitteilen, dass es nach einem Aufruf von **print** keinen Zeilenvorschub machen soll. Dies geschieht dadurch, dass man hinter den print-Befehl ein Komma einfügt.

Beispiel 7.4.9 Schleifen über Zeichenketten

```
>>> name = "Martin"
>>> for buchstabe in name:
  print buchstabe,
M a r t i n
```

Und hier noch ein Beispiel von der Ostalb:

Beispiel 7.4.10 Schleife mit 2 Variablen

```
>>> rest = "bele"
>>> anf = "ABCD"
>>> for buchst in anf:
  print buchst+rest

Abele
Bbele
Cbele
Dbele
```

Es fehlt aus dem Kapitel über strukturierte Daten noch das Beispiel, wie man eine Matrix schön darstellt.

Beispiel 7.4.11 Darstellung einer Matrix

```
#!/usr/bin/python

def matrixDarst(m):
  print ' '
  for zeile in m:
    spTupel = ()
    anzSpalten = len(zeile)
    for spalte in zeile:
      spTupel = spTupel + (spalte, )
    print "%+6.4f\t"*anzSpalten % spTupel


m = [[1, 0, 1], [-1, 2, 0], [0, -2, 1]]
matrixDarst(m)
```

Der Trick dabei ist, die Liste in ein Tupel zu verwandeln, das man an einen Formatstring übergibt. Dabei wird der Formatierungsbefehl genau so oft wiederholt, wie es Anzahl Spalten gibt.

Somit können wir ein kleines Quiz-Programm bauen, das Dictionaries benutzt. Es gibt den englischen Begriff aus und fragt die deutsche Übersetzung ab:

Beispiel 7.4.12 Ein kleines Quiz

```
#!/usr/bin/python
# -*- coding: utf8 -*-
englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh', 'sheep':'Schaf'}
punkte = 0
for tier in englDeutsch:
  print "Wie heisst  ", tier, " auf deutsch?"
  dTier = raw_input()
  if englDeutsch[tier] == dTier:
    punkte = punkte + 1
    print "Gut!!"
  else:
    print "so nicht"

  print "Du hast ", punkte, " richtige Antworten"
```

7.4.2. While-Schleifen

Oft soll aber nicht „gezählt geschleift“ werden, sondern man soll einen Block von Anweisungen solange wiederholen, bis eine bestimmte Bedingung erreicht ist. Der Programm-Code für solche Schleifen sieht wie folgt aus:

Beispiel 7.4.13 While-Schleife

```
>>> while (eingabe <> 'Ende'):
      machWas
```

Das hier beschriebene Programm-Segment ist ein typischer Fall für eine While-Schleife. Einen Programmablauf kann man so steuern:

Beispiel 7.4.14 Menu-Auswahl

```
while eingabe <> 'e':
  eingabe = raw_input('Was willst Du? m für mehr, w für weniger, e für Ende')
  if eingabe == 'm':
    machmehr
  elif eingabe == 'w':
    machweniger
  elif eingabe == 'e'
    pass # für e muss man gar nichts machen; wenn das Programm zum
         # nächsten Mal zur Überprüfung der While-Bedingung kommt,
         # wird die Schleife einfach nicht mehr durchlaufen
  else:
    print 'fehlerhafte Eingabe; nur m, w, e sind erlaubt'
```

Ein anderer klassischer Fall für eine While-Schleife ist das Lesen einer Datei. Da bisher noch keine Datei-Behandlung besprochen wurde, wird der Programmcode nicht in Python, sondern in Pseudocode wiedergegeben:

```
while nochEtwasVorhanden:
  liesEinZeichen
  if keinZeichenMehr:
    dannHoerHaltAufMitLesen
```

7.4.3. Verschiedene Probleme und ihre Lösung mittels Schleifen

Bei vielen dieser Probleme ist eine Menge von Dingen gegeben, die mit Hilfe einer Schleife durchsucht werden muss, wobei aus dieser Menge bei einigen dieser Probleme etwas herausgefiltert werden soll. Die Menge ist oft als eine Liste, manchmal als ein Dictionary gegeben.

7.4.3.1. Die Anzahl der Elemente einer Liste bestimmen

Nein, ich will hier niemanden ver...kackeiern. Natürlich ist in Python eine Funktion eingebaut, die genau die Länge einer Liste zurückgibt, denn das ist ja gerade das, was man unter der Anzahl der Elemente einer Liste versteht. Diese Funktion wird in dem unten stehenden Prográmmchen ganz am Anfang aufgerufen; schaun wir mal, ob wir das mittels einer Schleife auch rauskriegen. Diese Schleife zu programmieren hilft aber, viele ähnliche Probleme zu verstehen und zu lösen.

Beispiel 7.4.15 Länge einer Liste - zu Fuß berechnet

```
#!/usr/bin/python

liste = [7,12,5,23,1,14,18,33,4]
print len(liste)

zaehler = 0
for i in liste:
  zaehler += 1

print zaehler
```

Das ist wohl nicht so schwer zu verstehen! Vor Beginn der Schleife wird eine Variable Zähler deklariert und auf 0 gesetzt. Dann wird die Schleife Element für Element gelesen; bei jedem Schleifen-Durchlauf wird der Zähler um 1 hochgezählt.

7.4.3.2. Anzahl der Elemente einer Liste mit einer bestimmten Eigenschaft

Bei diesem Problem hilft die len-Funktion nicht mehr! Aber es ändert sich gar nicht so viel an dem vorigen Programm. Einzig innerhalb der Schleife wird gefragt, ob das aktuelle Element die Bedingung erfüllt, und nur wenn es das tut, wird der Zähler erhöht.

Beispiel 7.4.16 Anzahl der durch 3 teilbaren Zahlen in der Liste

```
#!/usr/bin/python

liste = [7,12,5,23,1,14,18,33,4]

zaehler = 0
for i in liste:
  if i % 3 == 0:
    zaehler += 1

print zaehler
```

Alles klar? Wenn nicht, lies noch mal die Einführung zu diesem Beispiel und die Erklärung zum vorigen Beispiel.

7.4.3.3. ... wie eben, aber mit Ergebnisliste

Also führen wir eine zweite Liste ein, die zu Beginn leer ist. Wenn die Bedingung erfüllt ist, wird mittels der append-Methode von Listen diese Liste erweitert.

Beispiel 7.4.17 Anzahl der durch 3 teilbaren Zahlen in der Liste mit Ergebnisliste

```
#!/usr/bin/python

liste = [7,12,5,23,1,14,18,33,4]

ergListe = []
zaehler = 0
for i in liste:
  if i % 3 == 0:
    zaehler += 1
    ergListe.append(i)

print zaehler
print ergListe
```

7.4.3.4. Das größte Element einer Menge finden

Sei also eine Liste von Elementen gegeben, für die es ein Ordnungskriterium gibt, so dass man also das größte Element bestimmen kann. Der Einfachheit halber ist dies eine Liste von natürlichen Zahlen, auf denen die übliche Größer-Relation definiert ist.

Beispiel 7.4.18 Größtes Element einer Liste

```
#!/usr/bin/python

liste = [7,12,5,23,1,14,18,33,4]
max = liste[0]
for i in liste[1:]:
  if i > max:
    max = i

print max
```

Die Logik ist wohl leicht zu erkennen: bevor die Schleife aufgerufen wird, wird das Maximum mit dem ersten (erinnere Dich: das ist das Element mit der Hausnummer 0) Element der Liste initialisiert. Dann wird die Schleife über die Liste ab dem zweiten Element durchlaufen. Jedes Element wird mit dem Maximum verglichen, und wenn das jeweils in Arbeit befindliche Element größer ist als das aktuelle Maximum, wird dieses aktuelle Element zum Maximum.

7.4.3.5. An welcher Stelle steht das größte Element einer Menge?

Im Gegensatz zum vorigen Beispiel interessiert nicht das größte Element, sondern nur die Position, an der dieses in der Liste steht.

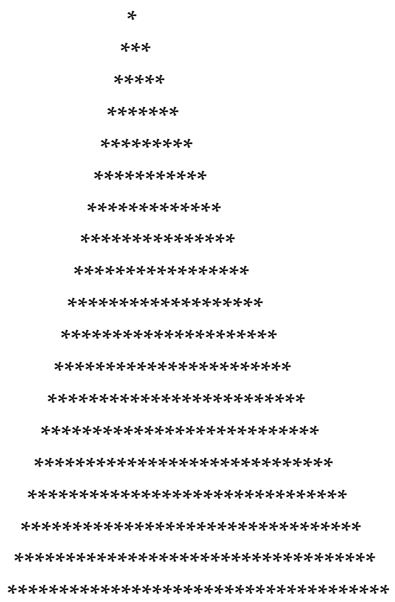
Beispiel 7.4.19 Position des größten Elements einer Liste

```
#!/usr/bin/python

liste = [7,12,5,23,1,14,18,33,4]
max = liste[0]
pos = 0
for i in range(1, len(liste)):
  if liste[i] > max:
    pos = i

print pos
```

Eine vollständig andere Logik für die Schleife muss hier angewendet werden. Vor Beginn des Schleifendurchlaufs wird wieder das Maximum mit dem ersten Element initialisiert, zusätzlich dazu wird die Position mit 0 besetzt. Es kann aber jetzt nicht mehr über die Liste als solche geschleift werden, sondern hier muss, da die Position interessiert, über die Zahl, die die Länge der Liste angibt, geschleift werden.

7.4.4. Übungen zu Schleifen

1. Ein Python-Programm soll eine Sternchen-Pyramide malen; das Ergebnis soll so aussehen:



2. Ein Programm soll die Quadratzahlen bis zu einer bestimmten Grenze ausgeben.
3. Aufgabe mit der Ausbreitung einer Epidemie:
   a) n = Bevölkerung (konstant)
   b) x(t) = Gesunde zur Zeit t
   c) y(t) = Kranke (= Ansteckende) zur Zeit t
   d) z(t) = Immune (= Genesene, Tote, Isolierte) zur Zeit t
   e) n = x + y + z
   f) x(t+1) - x(t) = -a * x(t) * y(t) (Abnahme der Gesunden proportional zu Anzahl der Gesunden und Anzahl der Kranken; Infektionsrate = a $>$ 0)
   g) z(t+1) - z(t) = b * y(t) (Zunahme der Immunen proportional zu Anzahl der Kranken; 0 $<=$ b = Immunisierungsrate $<=$ 1)
4. Quersumme einer Zahl ausrechnen; (beachte dabei: Typ-Konvertierung String-Element -$>$ Zahl)
5. Primzahlen mit dem Sieb des Eratostehnes berechnen
6. Fakultät einer Zahl berechnen
7. ggT und kgV ausrechnen mit Euklids Algorithmus
8. Newtonsches Näherungverfahren zur Berechnung von Nullstellen
9. Eine Zahl soll im Dezimalsystem eingegeben und in ein beliebiges anderes Stellenwertsystem umgerechnet werden.
10. Ein Programm soll geschrieben werden, das die Abrechnung für einen geliehenen Drucker macht. Dabei wurde mit dem Büromaschinenverleih vertraglich vereinbart, dass die fixen Kosten pro Monat 840,34 € betragen, die bereits ein Druckvolumen von 2000 Blatt beinhalten. Für jedes weitere gedruckte Blatt werden 0,038 € berechnet. Auf die bisher genannten Beträge muss noch die Mehrwertsteuer von 19 % aufgerechnet werden. Das Programm soll in Schritten von 500 Blatt (bis 10000 Blatt) die gesamten Kosten und die Kosten je Blatt ausgeben.
11. Ein nettes kleines Programm (bei dem man Schleifen und auch das „modulo-Rechnen“ anwenden kann) ist zu schreiben, das zwei Uhrzeiten addiert. Die Uhrzeiten werden jeweils als Listen übergeben, also die Uhrzeit 15:33:46 als [15,33,46].
12. Das Programm aus dem Kapitel über „Strukturierte Daten“, das in einer Eisdiele spielt, soll erweitert werden, so dass man eine Bestellung von mehreren Eisbechern aufnehmen kann (durchaus auch mehr als einen Eisbecher der selben Art) und danach eine Rechnung ausgegeben bekommt.

13. Das Programm aus dem Kapitel über „Strukturierte Daten“, das mit Daten von Staaten umgeht, soll so erweitert werden, dass in einem Programm alle Länder, die die selbe Landessprache haben, aufgelistet werden. Überlege Dir selbst, wie eine sinnvolle Ausgabe aussehen könnte.
14. Schreibe ein Programm, das in einem Dictionary Staaten mit ihren Hauptstädten speichert. In einem kleinen Rätsel sollen jetzt die Hauptstädte abgefragt werden. Je richtiger Antwort bekommt man einen Punkt, das Ergebnis soll am Ende des Rätsels angezeigt werden
15. Das vorige Programm soll lernen!! Wenn eine Hauptstadt falsch eingegeben wurde, das aber eine Hauptstadt eines Staates ist, soll ein Staat-Hauptstadt-Paar dem Dictionary hinzugefügt werden und das Rätsel von neuem starten, solange, bis alles fehlerfrei beantwortet ist.
16. Das Programm aus dem Kapitel über „Strukturierte Daten“, das Adressen behandelt, soll aufgebohrt werden, so dass ein schönes Adressbuch ausgegeben wird.
17. Das „Raus-“Geld soll gestückelt werden! Das Programm soll z.B. bei einem Rechnungsbetrag von 20,12 € und einer Bezahlung mit einem 100€ -Schein ausgeben, dass man einen 50€ -Schein, einen 20€ -Schein, einen 5€ -Schein, zwei 2€ -Münzen etc. herausbekommt.
18. Für eine gegebene Funktion (hier: eine ganzrationale Funktion) soll eine Wertetabelle in einem Intervall mit einer einzugebenden Schrittweite ausgegeben werden.
19. Die Sierpinski-Pfeilspitze kann man dadurch erzeugen, dass man im Pascal'schen Dreieck die ungeraden und die geraden Zahlen verschiedenfarbig darstellt. Schreibe ein Programm, das die Sierpinski-Pfeilspitze auf der Kommandozeile (shell) malt. Dabei soll „schwarz“ durch ein „x“ und weiß durch ein Leerzeichen dargestellt werden.
20. Heutzutage kommt man nicht mehr umhin, in einem Text über Programmierung auch auf Ver- und Entschlüsseln zu sprechen. Schreibe ein Programm, das eine monoalphabetische Verschlüsselung eines Textes liefert, so wie sie der Legende nach auf Caius Iulius Caesar zurückgeht. Caesar soll damals einfach um eine bestimmte Zahl im Alphabet weitergezählt haben. Beispiel: Wenn diese Zahl die 3 war, dann wurde aus A ein D, aus H ein K etc.

    Dabei sei der Text, wie damals zu Cäsars Zeit, in reinen Großbuchstaben gegeben. Der verschlüsselte Text soll auch nur Großbuchstaben enthalten.

    Dazu werden zwei Funktionen benötigt: ord(Zeichen) liefert die Nummer des Zeichens zurück, chr(Zahl) liefert das Zeichen mit der Nummer Zahl zurück.

    Ein Tip: geh in IDLE, schau nach, was A für eine Hausnummer hat!
21. Eine verbesserte Methode ist die Verschlüsselung mit Schlüsselwort. Dabei verschiebt man nicht jeden Buchstaben um den selben Wert, sondern denkt sich ein Schlüsselwort aus. Hier wird das beispielhaft mit einem sehr kurzen Schlüsselwort, dem Wort „BAD“, gemacht.

    B ist der zweite Buchstabe des Alfabets, A der erste, D der vierte. Es gibt also folgende einfache Zuordnung:

Tabelle 7.1. Schlüsselwort

Schlüssel	B	A	D
Wert	2	1	4

Wenn jetzt der Text „EIN GEHEIMER BRIEF“ verschlüsselt werden soll, dann geschieht das folgendermaßen (der Einfachheit halber ignoriere ich in diesem Beispiel die Leerzeichen):

a) Der erste Buchstabe des Textes, das E wird mit dem Buchstaben B, dem ersten Buchstaben des Schlüsselwortes, verschlüsselt. B hat den Wert 2, also wird von E aus zwei Buchstaben weitergezählt.
b) Der zweite Buchstabe des Textes, das I wird mit dem Buchstaben A, dem zweiten Buchstaben des Schlüsselwortes, verschlüsselt. A hat den Wert 1, also wird von I aus ein Buchstabe weitergezählt.
c) Der dritte Buchstabe des Textes, das N wird mit dem Buchstaben D, dem dritten Buchstaben des Schlüsselwortes, verschlüsselt. D hat den Wert 4, also wird von N aus vier Buchstaben weitergezählt.
d) Der vierte Buchstabe des Textes, das G wird mit dem Buchstaben B, dem ersten Buchstaben des Schlüsselwortes, verschlüsselt (denn da das Schlüsselwort nur 3 Buchstaben hat, geht es hier wieder von vorne los). B hat den Wert 2, also wird von G aus zwei Buchstaben weitergezählt.

e) usw.

Tabelle 7.2. Verschlüsselung

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Klartext	E	I	N	G	E	H	E	I	M	E	R	B	R	I	E	F
Schlüssel	B	A	D	B	A	D	B	A	D	B	A	D	B	A	D	B
Geheimtext	G	J	R	I	F	L	G	J	Q	G	S	F	T	J	I	H

7.5. Funktionen

7.5.1. Allgemeines zu Funktionen

Oft ist eine bestimmte Anweisungsfolge nicht für einen Wert, sondern für mehrere Werte zu durchlaufen. Es ist also praktisch, dieser Anweisungsfolge einen Namen zu geben, unter dem sie aufgerufen werden kann. Ideal ist es, wenn man beim Aufruf dieser Anweisungsfolge unter einem vorher festgelegten Namen noch einen oder mehrere Werte mitgeben kann. Das ist das Prinzip einer Funktion. Die Werte, die man einer solchen Funktion mitgibt, werden „Parameter“ genannt.

Dadurch gewinnt man. Man gewinnt Zeit: ein Stück Code muss nur einmal geschrieben werden. Man gewinnt Freunde: jemand anderer kann den Code benutzen, wenn er gut funktioniert. Man gewinnt Ordnung: das, was ich in einer Funktion codiert habe und von dem erwiesen ist, dass es funktioniert, kann ich „wegräumen“, so dass ich es nicht immer wieder lesen muss.

Funktionen kennt wahrscheinlich jeder aus der Mathematik. Dort sieht eine Funktion zum Beispiel so aus: $f(x) = 2x^2 - 3x + 5$ Die Funktion hat einen Namen, nämlich „f“, es gibt eine Variable, das „x“, das sozusagen in die Funktion eingegeben wird, und es gibt einen (Rückgabe-)Wert, der aus der Funktion herausgegeben wird, nämlich das, was auf der rechten Seite des Gleichheitszeichens ausgerechnet wird, wenn man für x einen Wert eingibt. Als Beispiel: wenn man in die oben notierte Funktion den Wert „2“ eingibt, berechnet die Funktion 2*4 - 3*2 +5 und gibt als Ergebnis 7 zurück. Wer etwas mehr Mathematik gemacht hat als die normale Oberstufen-Mathematik, der weiß auch, dass es Funktionen gibt, die mehr als eine Variable haben.

In einigen Programmiersprachen werden solche abgeschlossenen Blöcke von „funktionierenden Anweisungen“ auch Unterprogramme oder Prozeduren genannt, während Funktionen Unterprogramme sind, die einen Wert zurückgeben. In Pascal etwa gibt es verschiedene Schlüsselwörter für Prozeduren und Funktionen. In anderen Programmiersprachen hingegen werden die Begriffe gleichbedeutend benutzt.

7.5.2. Definition und Aufruf von Funktionen

7.5.2.1. Einfache Funktionen

Eine Funktion hat

- einen Funktionsnamen
- einen Funktionskopf
- einen Funktionskörper
- und manchmal einen Rückgabewert (den Funktionswert)

Für den Funktionsnamen gelten alle Regeln, die für Variablennamen (siehe dort) gelten. Es ist guter Stil, wenn man Funktionen als Namen ein Verb gibt, denn Funktionen tun etwas. Auch hier ist ein Vergleich mit der Mathematik interessant, denn auch da gibt es Funktionen. In Mathematik heißen Funktionen fast immer „f“, während in der Programmierung der Funktionsname schon Aufschluss darüber geben sollte, was die Funktion macht. In Mathematik berechnet eine Funktion immer einen Wert. In der Programmierung macht eine Funktion irgendwas.

Der Funktionskopf besteht aus dem reservierten Wort „def“, gefolgt vom Funktionsnamen und Parametern, die in Klammern mitgegeben werden. Selbst wenn einer Funktion keine Parameter mitgegeben werden, muss ein leeres Paar Klammern geschrieben werden. Der Funktionskopf endet mit einem

Doppelpunkt, und inzwischen ist bekannt, was nach dem Doppelpunkt gemacht werden muss: die Anweisungen, die zur Funktion gehören, der Funktionskörper, müssen eingerückt werden. Auch hier folgt gleich ein einfaches Beispiel.

Beispiel 7.5.1 Funktions-Definition

```
>>> def gruss():
      print "Hallo und Guten Tag"
```

Damit hat man eine Funktion definiert, und es ist klar, was sie leistet. Jetzt kann man diese Funktion aufrufen:

Beispiel 7.5.2 Funktionsaufruf

```
>>> gruss()
Hallo und Guten Tag
```

Diese Funktion hat also keinen Parameter (keine Variable, die etwas in die Funktion reinbringt) und keinen Rückgabewert, das heißt, es kommt auch nichts aus der Funktion heraus.

Wie vorher schon erwähnt, ist es besonders hilfreich, wenn eine Funktion nicht immer das selbe macht, sondern abhängig von einem mitgegebenem Wert etwas tut. Also bohren wir das obige Beispiel auf, so dass der Gruss etwas persönlicher wird:

Beispiel 7.5.3 Funktion mit Parameter

```
>>> def grussMitName(name):
      print "Hallo ", name, " und Guten Tag"
```

Dies wird jetzt aufgerufen mit

Beispiel 7.5.4 Funktionsaufruf mit Parameter

```
>>> grussMitName("Martin")
Hallo Martin und Guten Tag
```

7.5.2.2. Rückgabewerte und Parameterlisten

Funktionen können Werte zurückgegen. Das reservierte Wort, das einen Rückgabewert zurückgibt, heißt „return“. Es folgt das Beispiel dazu, und zwar sowohl die Definition als auch die Ausführung:

Beispiel 7.5.5 Funktion mit Rückgabewert

```
>>> def verdoppeln(zahl):
      return 2*zahl
>>> a = verdoppeln(3)
>>> print a
6
>>> b = verdoppeln(5.7)
>>> print b
11.4
```

Hier hingegen wird etwas in die Funktion reingegeben und es kommt auch ein Rückgabewert heraus. Dabei ist es der Funktion egal, welcher Art die Daten sind, die ihr übergeben werden:

Beispiel 7.5.6 nochmal: Funktion mit Rückgabewert

```
>>> c = verdoppeln("so was ")
>>> print c
so was so was
```

Wie weiter oben schon erwähnt wurde, kann eine Funktion auch mehrere Parameter bekommen.

Beispiel 7.5.7 Funktion mit mehreren Parametern

```
>>> def telNrDrucken(vorn, nachn, telnr):
      print vorn+' '+nachn
      print 'Telefon: ', telnr
>>> telNrDrucken('Martin','Schimmels', '12345')
Martin Schimmels
12345
```

Hier ist es natürlich wichtig, dass beim Aufruf der Funktion die Werte für die Parameter in der selben Reihenfolge übergeben werden, wie sie die Funktion in ihrem Funktionskopf deklariert hat, also in diesem Beispiel in der Reihenfolge Vorname, Nachname, Telefonnummer.

Es ist allerdings möglich, von dieser eben beschriebenen Regel abzuweichen, indem man beim Aufruf einer Funktion nicht nur den Wert für einen Parameter mitgibt, sondern im Aufruf der Funktion dem Parameternamen einen Wert zuweist. Das obige Beispiel lässt sich also auch so schreiben:

Beispiel 7.5.8 Parameterübergabe mit Parameternamen

```
>>> def telNrDrucken(vorn, nachn, telnr):
      print vorn+' '+nachn
      print 'Telefon: ', telnr
>>> telNrDrucken(telnr='12345', vorname='Martin',nachname='Schimmels', )
Martin Schimmels
12345
```

Hier wird also die Reihenfolge, wie sie in der Parameterliste des Funktionskopfes vorgegeben ist, nicht eingehalten, sondern die Reihenfolge wird beliebig gewählt, dafür aber bei jedem Parameterwert mitgeteilt, welchem Parameter dieser Wert zugewiesen werden soll.

Oft weiß man, dass für einen Parameter fast immer ein bestimmter Wert übergeben wird; das wird in der Informatik als „Default-Wert“ bezeichnet. Dann ist es gut, wenn man in der Parameterliste einer Funktion schon einen Standardwert festlegen kann. Das bedeutet, dass man für diesen Parameter der Funktion einen Wert an die Funktion übergeben kann (dann wird der genommen), es aber auch sein lassen kann (dann wird der in der Parameterliste festgelegte Standardwert genommen). Das Beispiel dazu:

Beispiel 7.5.9 Parameter mit Vorgabewert

```
>>> def verstaerkung(text, wieOft=1):
      print text * wieOft
>>> verstaerkung('hallo')
hallo
>>> verstaerkung('hallo',3)
hallohallohallo
```

Und, kein Wunder, das funktioniert natürlich auch mit einer Zahl als Wert für den ersten Parameter:

Beispiel 7.5.10 Parameter mit Vorgabewert (numerisch)

```
>>> verstaerkung(3)
3
>>> verstaerkung(3,4)
12
```

Es gibt sogar die Möglichkeit, Funktionen zu schreiben, bei denen die Anzahl der Parameter unbekannt ist. In einem solchen Fall übergibt man der Funktion ein Tupel von Werten. Die spannende Frage ist, wie die Funktion dieses Tupel entgegennimmt. Dies geschieht dadurch, dass dem Parameternamen in der Klammer der Funktionsdefinition ein „*“ vorangestellt wird. Das Beispiel dazu sieht so aus:

Beispiel 7.5.11 Unbekannte Anzahl Parameter

```
>>> def funktion1(para1, para2, *para3):
      print para1, para2, para3
>>> funktion1('Max', 'Senf', 'Hof', 'Isar', 'Pils', 'Auto')
Max Senf ('Hof', 'Isar', 'Pils', 'Auto')
```

Den ersten beiden Parametern, beide ohne einen „*“ in der Funktionsdefinition, werden also die ersten beiden Werte des Tupels zugewiesen. Dem letzten Parameter, dieser mit „*“, wird der gesamte Rest zugewiesen. Dieser gesamte Rest ist somit wieder ein Tupel, das aus diesem Grund in Klammern geschrieben wird, wobei die einzelnen Teile des Tupels in Anführungsstriche gesetzt und durch Kommata voneinander getrennt sind.

Mit diesem Wissen ist es aber auch einfach, die Parameter, auch die aus dem Tupel, einzeln auszugeben:

Beispiel 7.5.12 Nochmal: Unbekannte Anzahl Parameter

```
>>> def funktion1(para1, para2, *para3):
      print para1, para2,
      for x in para3:
        print x,
>>> funktion1('Max', 'Senf', 'Hof', 'Isar', 'Pils', 'Auto')
Max Senf Hof Isar Pils Auto
```

Oft soll eine Funktion ausdrücklich einen Wert zurückliefern. Dies geschieht durch den „return“-Befehl.

Beispiel 7.5.13 Funktion mit Rückgabewert

```
  def quadratFlaeche(seitenLaenge):
    return seitenLaenge * seitenLaenge
```

7.5.2.3. Gültigkeitsbereiche und Namensräume

An dieser Stelle kommt der Begriff *Namensraum* ins Spiel. Der Namensraum ist der Bereich, in dem eine Variable (oder ein anderer Teil eines Programms) bekannt und gültig ist. Das wird deswegen auch der Gültigkeitsbereich genannt. Das war bis jetzt noch kein Problem, weil unsere Programme noch nicht in kleinere Einheiten unterteilt waren.

Eine Funktion ist aber ein kleines „Programm im Programm“, womit eine solche Unterteilung in kleinere Einheiten geschieht. Damit gilt auch, dass eine Variable, die erst innerhalb einer Funktion zum ersten Mal benutzt wird, dadurch, dass ihr ein Wert zugewiesen wird, außerhalb der Funktion nicht bekannt ist. Das wird in der Programmierung eine *lokale Variable* genannt. Ein einfaches Beispiel, um das zu demonstrieren, sieht so aus:

Beispiel 7.5.14 Lokale Variable

```
#!/usr/bin/python

def doofFunktion():
  nr = 378
  print 'Nummer IN der Funktion:',  nr

doofFunktion()
print 'Nummer NACH der Funktion:',  nr
```

Die Variable `nr` wird hier in der Funktion definiert und gleich ausgegeben. Nach Beendigung der Funktion wird nochmals versucht, diese lokale Variable auszugeben. Hier ist die Ausgabe des Programms:

Beispiel 7.5.15 Lokale Variable

```
Nummer IN der Funktion: 378
Nummer NACH der Funktion:
Traceback (most recent call last):
  File "./namensraumFkt.py", line 8, in module
    print 'Nummer NACH der Funktion:',  nr
NameError: name 'nr' is not defined
```

Wie man sieht, wird der `print`-Befehl nach Beendigung der Schleife nicht ausgeführt, weil die Variable `nr` außerhalb der Funktion nicht bekannt ist, sondern das Programm bricht mit einer Fehlermeldung ab.

7.5.3. Funktionen wiederverwenden

Eine Funktion ist ein Stück Code, das funktioniert! Nicht nur, dass eine Funktion einen klar abgegrenzten „Tätigkeitsbereich“ hat, eine Funktion sollte so flexibel sein, dass sie mit verschiedenen Werten arbeiten kann und es sollte sicher gestellt sein, dass die Funktion auch vollkommen richtig arbeitet.

Wenn das der Fall ist, dann will man so ein kleines Schmuckstück natürlich auch mehrmals benutzen. Das könnte man natürlich so machen, dass man den Code einfach per „copy and paste“ in das neue

Programm nochmals einfügt. Aber das kann Probleme aufwerfen[6], denn unter Umständen fällt einem beim 15. Mal, dass man die Funktion benutzt, auf, dass man doch noch eine Kleinigkeit ändern muss. Also alle vorigen 14 Versuche nochmals anschauen???

Viel sinnvoller ist es, eine solche Funktion nur an einer Stelle gespeichert zu haben und nur diese eine Stelle zu ändern, wenn denn etwas geändert werden muss. Die Technik, die wir dabei anwenden, ist der Import. Dabei wird über die Import-Funktion von Python alles (oder auch nur Teile) aus der abgespeicherten Datei in das neue Programm eingefügt. Das kann auf 3 verschiedene Arten geschehen:

1. mittels `import datei` wird die gesamte Datei importiert. Dabei bleiben die Funktion und eventuelle Variable im Namensraum der gespeicherten Datei. Die Funktion und eventuelle Variable müssen in diesem Fall voll-qualifiziert aufgerufen werden, wenn sie verwendet werden sollen, das heißt, dass der Dateiname vorangestellt werden muss: `datei.funktion()`
2. mittels `from datei import funktion` wird nur die Funktion importiert. Dabei wird die Funktion in den Namensraum der neuen Datei übernommen. Sie wird in diesem Fall über den Funktionsnamen (ohne irgendeinen Zusatz) aufgerufen.
3. mittels `from datei import *` werden alle Dinge aus der gespeicherten Datei importiert und in den Namensraum der neuen Datei übernommen. Von dieser Art des Imports wird abgeraten, da es hier sehr auf die Disziplin des Programmierers ankommt, damit es keine doppelten Benennungen von Funktionen oder Variablen kommt.

Das soll jetzt an einem Beispiel gezeigt werden. Ich nehme dazu die Funktion `verdoppeln` von oben

Beispiel 7.5.16 Funktion verdoppeln

```
def verdoppeln(zahl):
    return 2*zahl
```

Diese wird in einer Datei `textfunktionen.py` gespeichert. In meinem neuen Programm will ich diese Funktion benutzen. Zuerst kommt hier die erste der oben beschriebenen Arten des Aufrufs:

Beispiel 7.5.17 Aufruf der Verdoppelung (1.)

```
import textfunktionen

print textfunktionen.verdoppeln(13)
```

Als nächstes folgt der Import mit „from“:

Beispiel 7.5.18 Aufruf der Verdoppelung (2.)

```
from textfunktionen import verdoppeln

print verdoppeln(13)
```

7.5.4. Globale Variable

Über Gültigkeitsbereiche von Variablen ist bisher noch gar nichts gesagt worden, ganz einfach deswegen, weil das bei der Konzeption von Python keine Bedeutung hatte. Oder wie es der Vater von Python formuliert hat: „We're all consenting adults[7]“ Deswegen gilt: Variable sind immer überall gültig, sie sind global. Das zeigen wir an einem Beispiel:

[6] und dabei denken wir nicht nur an Karl-Theodor z. G.
[7] [Programming Python2] , Seite 285

Beispiel 7.5.19 eine globale Variable

```
#!/usr/bin/python
# -*- coding: utf8 -*-

globaleVar = 'gloooobal!!'

def eineFunktion():
  print 'in der ersten Funktion: '+globaleVar

print 'außerhalb jeder Funktion: '+globaleVar
eineFunktion()
```

Die Ausgabe sieht so aus:

Beispiel 7.5.20 Ausgabe von globaler Variablen

```
außerhalb jeder Funktion: gloooobal!!
in der ersten Funktion: gloooobal!!
```

Jetzt deklarieren wir eine zweite Funktion, innerhalb derer eine Variable mit dem selben Variablennamen definiert wird.

Beispiel 7.5.21 Globale Variable und eine Funktion mit lokale Variablen

```
#!/usr/bin/python
# -*- coding: utf8 -*-

globaleVar = 'gloooobal!!'

def eineFunktion():
  print 'in der ersten Funktion: '+globaleVar

def zweiteFunktion():
  globaleVar = 'aber eigentlich lokal!!'
  print globaleVar

print 'außerhalb jeder Funktion: '+globaleVar
eineFunktion()
zweiteFunktion()
print 'außerhalb jeder Funktion: '+globaleVar
```

Jetzt sieht die Ausgabe so aus:

Beispiel 7.5.22 Ausgabe von globaler und lokaler Variablen

```
außerhalb jeder Funktion: gloooobal!!
in der ersten Funktion: gloooobal!!
aber eigentlich lokal!!
außerhalb jeder Funktion: gloooobal!!
```

Das Ergebnis: die Variable `globaleVar` innerhalb der Funktion `zweiteFunktion` ist etwas ganz

anderes als die Variable außerhalb der Funktion. Dafür ist diese Variable aber nur innerhalb der „zweiten Funktion“ bekannt. Das bezeichnet man als „lokale Variable“. Außerhalb dieser Funktion wird wieder die globale Variable benutzt.

Man kann dieses Verhalten verändern, indem man einer Variablen innerhalb einer Funktion das Attribut „global“ zuordnet. Das sieht so aus:

Beispiel 7.5.23 Globale Variable und eine Funktion mit lokaler Variablen

```
#!/usr/bin/python
# -*- coding: utf8 -*-

globaleVar = 'gloooobal!!'

def eineFunktion():
  print 'in der ersten Funktion: '+globaleVar

def zweiteFunktion():
  globaleVar = 'aber eigentlich lokal!!'
  print globaleVar

def dritteFunktion():
  global globaleVar
  globaleVar = 'jetzt wieder global, weil in der 3. Fkt. "über-definiert"!!'

print 'außerhalb jeder Funktion: '+globaleVar
eineFunktion()
zweiteFunktion()
print 'außerhalb jeder Funktion: '+globaleVar
dritteFunktion()
eineFunktion()
print 'außerhalb jeder Funktion: '+globaleVar
```

Die Ausgabe steht hier unten:

Beispiel 7.5.24 Ausgabe von globaler und lokaler Variablen

```
außerhalb jeder Funktion: gloooobal!!
in der ersten Funktion: gloooobal!!
aber eigentlich lokal!!
außerhalb jeder Funktion: gloooobal!!
in der ersten Funktion: jetzt wieder global,
          weil in der 3. Fkt. "über-definiert"!!
außerhalb jeder Funktion: jetzt wieder global,
          weil in der 3. Fkt. "über-definiert"!!
```

7.5.5. Rekursive Funktionen

Oft ist von innerhalb einer Funktion dieselbe Funktion, eventuell mit einem anderen Wert für den Parameter, nochmals aufzurufen. Das wird als „Rekursion“ bezeichnet. Eines der klassischen Beispiele (oder vielleicht das Beispiel dazu überhaupt) ist die Berechnung der Fakultät einer Zahl. Zur Erinnerung: 5! ist die mathematische Schreibweise für 5-Fakultät und ist definiert als 5! = 1 * 2 * 3 * 4 * 5

Oder anders gesagt: 5! = 5 * 4! und 4! = 4 * 3!. Das heißt, um 5! auszurechnen, wär es geschickt, wenn man vorher 4! ausgerechnet hätte. Und so weiter, bis man bei der 1 angelangt ist.

Beispiel 7.5.25 Fakultät rekursiv

```
def fak_rek(zahl):
  if zahl == 1:
    return 1
  else:
    return zahl*fak_rek(zahl-1)
```

Und der Aufruf dieser Funktion sieht ganz einfach aus:

Beispiel 7.5.26 Fakultät rekursiv

```
zahl = 12
print fak_rek(zahl)
```

7.5.6. Funktionen als Parameter von Funktionen

Ja, geht das denn überhaupt? Und was soll das? Ich schreibe eine Funktion, und diese Funktion soll intern etwas machen was in einer anderen Funktion bereits gelöst ist. Aber nicht immer soll das selbe gemacht werden, sondern es können sich verschiedene Anforderungen ergeben.

Das soll an einem einfachen Beispiel gezeigt werden, das Text-Dateien bearbeitet. Als Beispiel-Text nehmen wir die Datei `testTextFktInFkt.txt`, die folgenden Text enthält:

Beispiel 7.5.27 Ein doofer Text mit vielen Pythons

```
Dieser Text ist so sinnvoll, wie die meisten Texte,
die aus der Ecke von Monty Python stammen.
Er soll nur demonstrieren,
dass man mit der Sprache Python
auch Monty Python zu Hilfe kommen kann.
```

In diesem Text sollen jetzt alle Zeilen, die „Python“ enthalten, ausgegeben werden, wobei das Wort „Python“ in Großbuchstaben geschrieben werden soll.

Die dazugehörige Funktion, die das mit jeder einzelnen Zeile macht, ist schnell geschrieben. Gespeichert wird sie in der Datei `grKlSchreiben.py`

Beispiel 7.5.28 Gross-Schreiben eines Begriffs in einer Zeile

```
#!/usr/bin/python
# -*- coding: utf8 -*-

def grossSchreiben(zeile,  wort):
  print zeile.replace(wort, wort.upper())
```

Auch der Rahmen für das zeilenweise Bearbeiten einer Textdatei ist kein Hexenwerk:

Beispiel 7.5.29 Rahmen für Bearbeitung einer Text-Datei

```
1
2  #!/usr/bin/python
3  # -*- coding: utf8 -*-
4
5  def textBearbeiten(????):
6    eingabeDatei = open('testTextFktInFkt.txt', 'r')
7    wort = 'Python'
8    for eineZeile in eingabeDatei.readlines():
9      if wort in eineZeile:
10       ????(eineZeile, wort)
11   eingabeDatei.close()
12
13 if __name__ == '__main__':
14   textBearbeiten(????)
15
```

Auch wenn Dateien erst später im Text (siehe Kapitel 13 „Dateizugriff“) behandelt werden, ist das verständlich. Die beiden letzten Zeilen starten dieses Programm, falls es direkt aufgerufen wird und nicht irgendwo importiert wird. Die Funktion `textBearbeiten` hat in der Parameterliste noch Fragezeichen, denn noch weiß diese Funktion nicht, was sie überhaupt für eine Aufgaben bekommt. Als erstes Aktion führt sie aber das Öffnen der Textdatei aus, das zu suchende Wort wird in der Variablen `wort` abgelegt, und dann wird die gesamte Datei mittels `readlines` in eine Liste eingelesen. Diese Liste wird in einer „for“-Schleife abgearbeitet. Wenn das gesuchte Wort in einer Zeile auftaucht, dann muss irgendwas mit dieser Zeile gemacht werden, aber da noch nicht klar ist, was, stehen hier auch noch Fragezeichen.

Doch! Es ist doch klar, was mit jeder Zeile gemacht werden soll, nämlich das, was schon oben in der Datei `grKlSchreiben.py` schon codiert wurde. Also muss ich doch nur noch ganz wenige Dinge ändern.

1. die Funktion `grossSchreiben` muss aus der Datei `grKlSchreiben.py` importiert werden, diesmal mit der Import-Option „as eineFunktion“, damit die Funktion, die importiert wird, innerhalb des Programms immer den selben Namen hat.
2. sie wird beim Aufruf von `textBearbeiten` mitgegeben
3. und im Kopf der Funktion `textBearbeiten` entgegengenommen
4. und dann aufgerufen, wenn das gesuchte Wort gefunden wurde.

Das Programm sieht also so aus:

Beispiel 7.5.30 Python wird gross geschrieben

```
1
2  #!/usr/bin/python
3  # -*- coding: utf8 -*-
4  from grklSchreiben import grossSchreiben as eineFunktion
5
6  def textBearbeiten(eineFunktion):
7    eingabeDatei = open('testTextFktInFkt.txt', 'r')
8    wort = 'Python'
9    for eineZeile in eingabeDatei.readlines():
10     if wort in eineZeile:
11       eineFunktion(eineZeile, wort)
12   eingabeDatei.close()
13
14 if __name__ == '__main__':
15   textBearbeiten(eineFunktion)
16
```

Das sieht umständlich aus? Es zahlt sich aber aus. Denn die Datei, die `grossSchreiben` enthält, heißt ja `grklSchreiben.py`. Schreib also dort eine weitere Funktion `kleinSchreiben`. Um diese neue Funktion auf die Text-Datei loszulassen, muss nur eine Stelle in dem Programm geändert werden, nämlich in der „import“-Zeile! Wow!

7.5.7. Übungen zu Funktionen

1. Ein Programm soll in zwei Funktionen Fläche und Umfang eines Rechtecks ausrechnen.
2. Ein Programm soll in zwei Funktionen Oberfläche und Volumen eines Quaders ausrechnen.
3. Ein Programm soll in drei Funktionen Mantelfläche, Oberfläche und Volumen einer quadratischen Pyramide berechnen.
4. Es sollen zwei Funktionen geschrieben werden, die jeweils das Minimum und Maximum eines Paares zurückgeben, dazu dann ein Programm, das diese beiden Funktionen testet.
5. In einem Programm soll das Newton’sche Näherungsverfahren für die Berechnung einer Nullstelle einer Funktion durchgeführt werden; hierbei sollen Funktion und Ableitungsfunktion als Funktionen im Programm geschrieben werden.
6. Ein Programm soll eine Dezimalzahl in eine römische Zahl umwandeln.

Teil V.

Modularisierung

8. Module

8.1. Mehr Mathematik

Im Kapitel über Zahlen wurden einfache Rechenaufgaben an Python gestellt, Aufgaben auf Grundschulniveau. Python soll uns natürlich auch bei komplizierteren mathematischen Problemen eine Hilfe sein. Probieren wir es also:

Beispiel 8.1.1 Fehler! Python kann kein Mathe!

```
>>> rad = 2.68
>>> umf = 2 * pi * rad
  Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    umf = 2 * pi * rad
  NameError: name 'pi' is not defined
```

Depp depperter! pi kennt doch jeder!
Der Grund dafür, dass sich Python hier weigert, unseren Befehl auszuführen, ist, dass Python außer dem Kern der Sprache noch eine ganze Menge Module enthält. Diese Module sind nicht ständig präsent, sondern sie müssen vielmehr vom Programmierer bei Bedarf zugeladen werden. Dies macht Sinn, denn somit wird nur das in ein Programm eingebaut, was wirklich benötigt wird, und das spart Zeit bei der Ausführung des Programms. Also laden wir uns mal das Mathematik-Modul hinzu, nein genauer: laden wir uns aus dem Mathematik-Modul das hinzu, was wir brauchen:

Beispiel 8.1.2 Import aus der Mathematik-Bibliothek

```
>>> from math import pi
>>> umf = 2 * pi * rad
>>> print umf
16.8389366232
```

Warum haben wir nicht das gesamten Mathematik-Modul hinzugeladen? Es gibt 2 wichtige Gründe: erstens packt man so wenig in ein Programm hinein, wie möglich, so viel, wie nötig, zweitens vereinfacht man sich so die Benennung von Variablen, Konstanten, Funktionen usw. Schauen wir einmal, was passiert, wenn man sich das ganze Mathematik-Modul hinzulädt:

Beispiel 8.1.3 Import der gesamten Mathematik-Bibliothek

```
>>> import math
>>> umf = 2 * pi * rad

Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    umf = 2 * pi * rad
NameError: name 'pi' is not defined
```

Es scheint nichts geschehen zu sein. pi ist immer noch nicht bekannt, obwohl jetzt alles aus „math“ in unser Programm importiert wurde. Aber wenn man ein ganzes Modul importiert, verbleiben alle

Variablen, Konstanten, Funktionen usw. im Namensraum des Moduls, das heißt, dass die Variablen und Funktionen des Moduls nur als Teile des Moduls bekannt sind. Ich muss also die Variable pi durch den vollständigen Namen ansprechen:

Beispiel 8.1.4 Qualifzierter Bezeichner aus einer Bibliothek

```
>>> umf = 2 * math.pi * rad
>>> print umf
16.8389366232
```

Vollständiger Name bedeutet, dass dem Konstantennamen pi hier der Modul-Name math vorangestellt wird, durch einen Punkt voneinander getrennt.

Bisher wurden 2 Methoden des Modul-Aufrufs angesprochen:

- Der Import von einzelnen Elementen eines Moduls mit

  `from modul import dingsbums`
- Der Import des gesamten Moduls mit

  `import modul`

Es gibt noch eine dritte Methode, die hier erwähnt, vor der aber auch sehr gewarnt wird.

```
from modul import *
```

importiert alles aus dem Modul, aber alles wird in den Namensraum des aufrufenden Programms übernommen. Das kann sehr gefährlich werden, wenn im aufrufenden Programm eine Variable oder eine Funktion den selben Namen hat wie im Modul.

Im Modul „math“ befindet sich natürlich noch viel mehr, aber was? Ganz einfach, fordern wir Hilfe an:

Beispiel 8.1.5 Hilfe zum Mathematik-Modul

```
>>> help(math)
Help on module math:

NAME
    math

FILE
    /usr/lib/python2.5/lib-dynload/math.so

MODULE DOCS
    http://www.python.org/doc/current/lib/module-math.html

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

    atan(...)
        atan(x)

        Return the arc tangent (measured in radians) of x.

    atan2(...)
        atan2(y, x)

        Return the arc tangent (measured in radians) of y/x.
        Unlike atan(y/x), the signs of both x and y are considered.

    ceil(...)
        ceil(x)

        Return the ceiling of x as a float.
        This is the smallest integral value >= x.

    cos(...)
        cos(x)

        Return the cosine of x (measured in radians).
```

Natürlich enden die mathematischen Funktionen hier nicht beim „cos“! (Aber mehr muss in diesen Text nicht rein, und wer mehr Mathematik braucht, weiß jetzt, wie man nachschlägt!) Aber damit kann man doch schon ganz ordentlich arbeiten.

Beispiel 8.1.6 Noch mehr Mathematik

```
>>> from math import cos, sin, pi
>>> x = 1.234
>>> sin(x)
0.94381820937463368
>>> sin(pi)
1.2246063538223773e-16
>>> cos(pi)
-1.0
```

8.2. Eigene Module

Das ist ja nett, dass jemand ein Mathematik-Modul zusammengestellt hat, denn das braucht man doch tagtäglich. Aber Du und ich, wir sind ja wenigstens genau so genial und fleißig und sozial, und deswegen wollen wir unsere eigenen Produkte, die wir so gut programmiert haben, dass wir oder jemand anderer sie wiederverwenden kann, für die Welt da draußen zur Verfügung stellen. Zuerst wollen wir sie uns selber aber zur Verfügung stellen, das heißt, wir wollen sie wiederbenutzbar machen. Was müssen wir dazu tun?

Ein Modul ist nichts anderes als ein Verzeichnis, dessen Inhalt vom Anwender (dem Programmierer also) benutzt werden kann. Dazu sind allerdings noch ein paar Kleinigkeiten zu beachten. Besonders wichtig ist, dass das Verzeichnis eine Datei `__init__.py` (das sind wieder mal 2 Unterstriche vor und 2 nach dem init) enthält. Aber machen wir das ganze mal schön der Reihe nach.

1. Gehen wir davon aus, dass wir in einigen Dateien verschiedene kleine Funktionen geschrieben haben, mit denen wir zufrieden sind, weil sie gut funktionieren. Also schieben wir diese Dateien (oder besser vielleicht eine Kopie davon) in ein eigenes Verzeichnis, das wir der Einfachheit halber „eigeneModule“ nennen wollen.

2. Unsere erste Datei heißt `fakBinko.py` und enthält zwei Funktionen, nämlich die Funktion `fak`, die die Fakultät einer Zahl ausrechnet, und die Funktion `binko`, die den Binomialkoeffizienten zweier Zahlen ausrechnet.

3. Unsere zweite Datei heißt `fibo.py` und enthält eine Funktion `fibo`, die die n-te Fibonacci-Zahl ausrechnet.

4. Jetzt erstelle ich eine Datei `__init__.py`, die aus den beiden vorher genannten Dateien die 3 Funktionen importiert. Der Inhalt der Datei `__init__.py` ist also:

Beispiel 8.2.1 Eine eigene Modul-Datei

```
from fakBinko import fak,  binko
from fibo import fibo
```

5. Jetzt ist alles hergerichtet. Ich kann jetzt in einer anderen Datei das Modul aufrufen. Das sieht so aus:

Beispiel 8.2.2 Aufruf des eigenen Moduls

```
#!/usr/bin/python
# -*- coding: utf8 -*-

import eigeneModule

print eigeneModule.fak(5)
print eigeneModule.fibo(5)
```

Wie man sieht, wird jetzt das ganze Modul importiert. Danach stehen die Funktionen unter

- `eigeneModule.fak()`
- bzw. unter `eigeneModule.fibo()`

zur Verfügung.

Das soll jetzt noch ein wenig mehr Benutzerfreundlichkeit bekommen. Das eigene Modul soll auch noch einen Hilfetext erhalten, den man sich bei Bedarf anschauen kann. Der Hilfetext ist einfach eine Zeichenkette in dreifachen Anführungszeichen. Also sieht die Modul-Datei `__init__.py` so aus:

Beispiel 8.2.3 Eine eigene Modul-Datei mit Hilfetext

```
#!/usr/bin/python
# -*- coding: utf8 -*-
'''
bisher vorhanden:
  fak(z): berechnet die Fakultät von z
  binko(n,k): berechnet n über k (Binomialkoeffizient
  fibo(z): berechnet die n-te Fibonacci-Zahl
'''
from fakBinko import fak,  binko
from fibo import fibo
```

Die Hilfe kann ich nach dem Import des Moduls aufrufen:

Beispiel 8.2.4 Aufruf der Hilfe zum eigenen Modul

```
>>> help(eigeneModule)

Help on package eigeneModule:

NAME
    eigeneModule

FILE
    /home/fmartin/bin/python/eigeneModule/__init__.py

DESCRIPTION
    bisher voranden:
      fak(z): berechnet die Fakultät von z
      binko(n,k): berechnet n über k (Binomialkoeffizient)
      fibo(z): berechnet die n-te Fibonacci-Zahl

PACKAGE CONTENTS
    fakBinko
    fibo
```

Dem, der unter Unix / Linux arbeitet, kommt diese Darstellung bekannt vor: so sehen alle Manuals[1] aus. Das ist doch einfach schön, außerdem schön einfach und es freut den Lernenden, weil er etwas wiedererkennt.

geschachtelte Struktur für Module muss noch geschrieben werden

[1] die man mit man(Befehl) aufruft

Teil VI.

Objektorientierte Programmierung

9. Klassen

9.1. Ist schon klasse, so eine Klasse!

Wie schon weiter oben bemerkt, ist Python eine objektorientierte Programmiersprache - nur hat das bis hierher noch niemand gemerkt (oder fast kein Leser). Wenn wir bisher eine Variable deklariert haben, dann dadurch, dass wir ihr einen Wert zugewiesen haben. Ohne uns darüber bewußt zu werden haben wir aber dadurch gleichzeitig eine Zuordnung zu einer Klasse gemacht.

Was ist nun aber Objektorientierung? Und wozu dient sie?

Bei der objektorientierten Programmierung geht es darum, die für ein Programm relevanten Daten möglichst „naturgetreu“ abzubilden. Das Mittel dazu ist die Schaffung von Klassen, einer Beschreibung von Eigenschaften gleichartiger Dinge. Diese gemeinsamen Eigenschaften werden in der Sprache der Objektorientierung „Attribute“ genannt. Diese gleichartigen Dinge haben aber nicht nur gemeinsame Eigenschaften, sondern „handeln“ auch gleich. Diese Handlungen von Elementen einer Klasse nennt man „Methoden“. Kurz gesagt:

- Attribute beschreiben Eigenschaften
- Methoden beschreiben Verhalten

Eine Klasse ist eine Vorlage für viele Objekte. In der Klasse wird beschrieben, wie jedes der zu erzeugenden Objekte aussehen soll und wie sich jedes der Objekte verhalten soll. Die Klasse selbst ist nur zu dieser einen Sache zu gebrauchen: eines oder mehrere Objekte nach ihrem Muster zu bauen. Man kann sich das vorstellen wie eine Ausstechform beim Plätzchenbacken. Die Form ist wirklich nicht geniessbar, weil sie aus Metall oder Plastik ist, aber was man damit aus dem Teig aussticht und backt: lecker, lecker.

Attribute und Methoden haben Namen. Man sollte sich von Anfang an darum bemühen, hier konsequente Schreibweisen einzuhalten. Wie für alle Variablennamen gilt auch für Attribute und Methoden, dass sie klein geschrieben werden. Bei längeren Namen bietet sich die Schreibweise mit Unterstrich (`ein_langer_Variablen_Name`) oder die Kamel-Schreibweise (`einLangerVariablenName`) [1] an.

Die wichtigste Regel in Python für den Umgang mit Objekten und ihren Attribute und Methoden folgt:

- Attribute eines Objektes werden mit `objekt.attribut`
- und Methoden eines Objektes mit `objekt.methode` angesprochen.

Viel wichtiger aber noch ist Konsequenz bei der Wahl des Namens. Wie aus dem vorvorigen Absatz hervorgeht, beschreiben Attribute Eigenschaften. Für einen Attributnamen nimmt man also Substantive. Die Werte der Attribute hingegen sind oft Adjektive: das Attribut „Farbe“, als Attribut natürlich klein `farbe` geschrieben, kann die Werte rot, grün, gelb etc. annehmen.

Methoden aber beschreiben, was mit dem Objekt passiert, meistens sogar genauer: was mit einem Attribut des Objekts passiert. Aus diesem Grund nimmt man für Methodennamen Verben, in Fortführung des obigen Beispiels also etwa eine Methode `farbeAendern`.

Das Thema „gute Programme“ (angefangen bei gute Programme) muss jetzt noch ergänzt werden. Ein Programm ist gut, wenn eine Klassen-Definition gut ist. Das ist der Fall, wenn eine Klasse viele kurze Methoden besitzt. Die Methodennamen sollten Verben sein, und ein Verb beschreibt **genau eine** Tätigkeit. Eine Methode ist also gut geschrieben, wenn sie eine Aktion ausführt; das bedeutet andersherum aber auch, dass eine Klassenbeschreibung, bei der eine Methode mehr als eine Tätigkeit durchführt, schlecht ist.

Wenn man sich an diese elementaren Regeln hält:

1. Substantive für Attribute
2. Verben für Methoden

[1] .. die so heißt, weil die Großbuchstaben wie Kamelhöcker nach oben herausstehen.

hat man für die objektorientierte Programmierung schon viel gewonnen und viele potentielle Fehlermöglichkeiten von vorneherein ausgeschaltet. Wenn man trotzdem irgendwann einen Namen hat, der nicht diesen Regeln entspricht, hilft es oft, sich in ein Objekt der Klasse hineinzuversetzen. „Ich bin ein T-Shirt. Ich habe eine Farbe. Diese Farbe kann ich ändern." Das hört sich zwar am Anfang sehr albern oder kindisch an, hilft aber, Strukturen zu erkennen.

Ein Beispiel für eine Klasse und die nach ihrem Muster erzeugten Objekte folgt hier: Stecker und Steckdose sehen (in Deutschland) gleich aus und funktionieren gleich. Es reicht also, dass ich einmal allgemein beschreibe, wie Stecker und Steckdosen aussehen und was ihre Fähigkeiten sind. Dann benutze ich diese Beschreibung, um einen Kühlschrank-Stecker, einen Computer-Stecker, einen Fön-Stecker zu konstruieren. Wenn meine allgemeine Beschreibung fehlerfrei ist, dann bin ich sicher, dass alle speziellen Stecker fehlerfrei funktionieren.

Wenn man diese allgemeine Beschreibung macht — vorausgesetzt, man macht das sinnvoll und richtig — erreicht man dadurch mehreres:

1. Es wächst zusammen, was zusammen gehört. (W. Brandt, aber in einem ganz anderen Zusammenhang!) Mit den Elementen einer Klasse kann man dann nicht mehr jeden Unfug anstellen, sondern nur noch das, was in der Klasse als erlaubte Aktionen festgelegt wurde. Und man kann einem Element einer Klasse auch keine Eigenschaften andichten, die nicht in der Klasse festgelegt sind.
2. Und damit ist viel Code wiederverwertbar. Hat man erst einmal eine Klasse ordentlich beschrieben und getestet, dann verhält die sich wie eine ganz nette schwarze Kiste: ich muss nicht mehr wissen, was sich im Inneren der Kiste abspielt, sondern ich kann darauf vertrauen, dass sich ein Objekt, das ich dieser Klasse zuordne, so verhält wie jedes andere Objekt dieser Klasse.

   Vor allem ist Code damit wartbar geworden. Der Programmier-Anfänger hat oft die Vorstellung, dass ein Programm einmal geschrieben und danach nie wieder angeschaut wird. Die Realität aber ist, dass ein Programm geschrieben wird, die elementaren Eigenschaften, die gefordert wurden abbildet, aber dass im Laufe der Zeit auffällt, dass z.B. Sonderfälle nicht behandelt wurden. Ebenso merkt man erst im Laufe der Zeit, dass ein Programm, das von einem Benutzer interaktiv bedient wird, nicht alle möglichen — damit sind vor allem die „unmöglichen" gemeint — Eingaben eines Benutzers abfängt. Auch mit etwas, was ich als Programmierer gar nicht erwartet habe (soooo doof kann doch kein Anwender sein), darf nicht dazu führen, dass etwas falsch bearbeitet wird oder gar das Programm abbricht

   Deswegen findet man oft verschiedene Versionen eines Programms, die sich dadurch unterscheiden, dass mögliche Fehler ausgemerzt wurden — und andere dafür auftauchen. Deswegen ist es wichtig, dass ein Programm nicht nur vom Programmierer zur Zeit des Programmierens verstanden wird, sondern auch noch Monate später und vielleicht von jemand ganz anderem gelesen werden kann.
3. Von einer Klasse kann man andere Klassen herleiten. Das bedeutet, dass eine Klasse viele Eigenschaften einer anderen Klasse erben kann. Und schon wieder hat man Zeit und Arbeit gespart.

Lies noch einmal nach, was weiter oben über gute Programme geschrieben wurde, und Du verstehst, dass Objektorientierung dazu beiträgt, Dinge einfach zu halten: einfach in der Bedeutung „leicht", aber auch einfach in der Bedeutung „nur einmal".

In der Theorie der objektorientierten Programmierung haben sich im Laufe der Zeit Vorgehensweisen und Richtlinien ergeben, die durch die folgenden Fachbegriffe im Zusammenhang mit objektorientierter Programmierung beschrieben werden.

1. **Vererbung**: ein Stecker ist noch sehr allgemein; spezielle Stecker sind Kaltgerätestecker, Monitor-Stecker, USB-Stecker. Alles, was prinzipiell Stecker auszeichnet, gilt für jeden dieser speziellen Stecker; aber es gilt für jeden dieser Stecker noch ein bißchen mehr. Das heißt, dass jeder spezielle Stecker die allgemeinen Eigenschaften vom allgemeinen Stecker erbt und noch ein paar spezielle Eigenschaften hinzugefügt werden.
2. **Kapselung**: alles, was zu einer Klasse gehört, soll (oder manchmal: kann) nur von Objekten dieser Klasse angesprochen werden. Von außen sind die Eigenschaften der Klasse „versteckt", und damit gerät man weniger in Gefahr, mit Variablen irgendeinen Unsinn zu machen.
3. **Polymorphie** heißt „Vielgestaltigkeit". Damit ist gemeint, dass sich eine Eigenschaft in verschiedenen Klassen verschieden zeigen kann. Ein ganz einfaches Beispiel haben wir schon weiter oben kennen gelernt, nur ist das gar nicht richtig aufgefallen. Das Zeichen + bezeichnet eine Eigenschaft von Zahlen, aber auch eine von Zeichenketten. In Verbindung mit Zahlen bedeutet es die

aus der Schule gewohnte Addition; in Verbindung mit Zeichenketten bedeutet es das Hintereinanderschreiben.

Die erste Klasse erstellen wir in IDLE. Das erlaubt es uns, wirklich interaktiv mit der Klasse herumzuspielen. Außerdem ist der Editor von IDLE da sehr hilfreich, wie man gleich sehen wird. Eine Klasse wird durch das Schlüsselwort `class` festgelegt. Bauen wir also die Klasse `Angestellter`, die als Attribute den Namen, das Alter, das Einkommen und die Berufsbezeichnung hat.

Beispiel 9.1.1 Klasse Angestellter

```
>>> class Angestellter:
      def __init__(self, name, alter, einkommen, berufsbezeichnung):
        self.name = name
        self.alter = alter
        self.einkommen = einkommen
        self.berufsbezeichnung = berufsbezeichnung
>>> ich = angestellter('Martin Schimmels', 54, 20000, 'Programmierer')
>>> ek = angestellter('Eckard Krauss', 39, 30000, 'Dozent')
>>> print ich.name
Martin Schimmels
>>> print ek.name
Eckard Krauss
```

Und hier folgen die Erläuterungen!
Eine Klasse ist wie eine Backform: ein Muster für das, was man herstellen will. Unser Muster für den Angestellten sagt uns, dass ein Angestellter immer einen Namen, ein Alter, ein Einkommen und eine Berufsbezeichnung hat. Die Methode, mit der man aus dem Muster ein tatsächliches Objekt konstruiert, heißt Konstruktor. Eine Klasse wird immer erstellt durch

Beispiel 9.1.2 Klassendefinition

```
class Pipapo:
   ????
   ????
```

Und nochmal: die Klasse macht nichts, ist nichts ... außer einem Muster. Objekte nach dem Muster werden erstellt durch

Beispiel 9.1.3 Objektdefinition

```
objekt1 = Pipapo()
objekt2 = Pipapo()
objekt3 = Pipapo()
```

Aber die 3 Objekte verhalten sich jetzt so, wie es in der Klassenbeschreibung (im Muster) vorgegeben wurde.

Der Konstruktor einer Klasse heißt in Python immer `__init__`. (Das sind zwei Unterstriche vor dem init und zwei nach dem init.) Wenn ich jetzt also ein Objekt der Klasse `Angestellter` erzeugen will, stelle ich eine Variable bereit, in diesem Fall die Variable `ich`, der ich mitteile, dass sie einen Angestellten darstellen soll. In Klammern werden die Attribute angegeben, also Name, Alter, Einkommen und Berufsbezeichnung. Damit wird jetzt der Konstruktor der Klasse aufgerufen. Der Konstruktor hat in Klammern ebenfalls die Attribute angegeben, damit er das, was übergeben wird, in Empfang nimmt. Zusätzlich hat der Konstruktor immer als ersten Parameter den Parameter `self`, der eine Referenz auf sich selbst ist, in diesem Fall also auf die Variable `ich`. Jetzt werden die übergebenen Parameterwerte

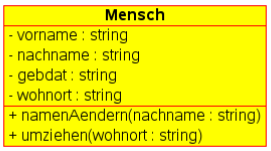
an Attribute des Objekts weitergegeben, was durch die Anweisung `self.name = name` geschieht. Damit wird jetzt dem Attribut `name` des Objekts `ich` der Wert „Martin Schimmels“ zugeordnet. Dass das tatsächlich funktioniert hat, sieht man einige Zeilen später, wenn der Wert von `ich.name` ausgegeben wird. Der ist tatsächlich vorhanden und lautet „Martin Schimmels“.

Objektorientierung geht heute - im ersten Jahrzehnt des 21. Jahrhunderts - nicht mehr ohne UML. UML ist die Abkürzung für „Unified Modelling Language“. Und genau das ist es: eine vereinheitliche Modellierungssprache für die Modellierung von Klassen (und anderem mehr. Aber das gehört jetzt nicht hierher!).

Modellieren wir also unsere zweite Klasse: Menschen!! (Aber abstrahieren wir soviel es geht, damit unser Modell einer Klasse überschaubar bleibt.) Menschen haben einen Vornamen, einen Nachnamen, ein Geburtsdatum. Geburtsdatum und Vorname sind unveränderlich (wenn man mal von Künstlern und Verbrechern absieht), der Nachname kann sich ändern. Menschen haben einen Wohnort, und der ändert sich (wahrscheinlich häufiger als der Nachname). So! Das reicht!

In UML sieht das Modell der Klasse `Mensch`, so wie es oben beschrieben wurde, so aus:

Abbildung 9.1. Klassendiagramm `Mensch`



So schön kann ich natürlich nicht von Hand zeichnen. Es gibt zum Glück jede Menge UML-Programme, die eine Klasse modellieren und das Ergebnis schön darstellen. Dieses Diagramm wurde mit dem Programm „Umbrello“ gezeichnet, das aus der Linux-Ecke kommt und „Open Source“ ist, also nichts kostet.

Eine kurze Beschreibung dessen, was man da sieht: im oberen kleinen Kasten steht der Name der Klasse. Vereinbarung: Klassennamen beginnen mit einem Großbuchstaben. Im zweiten Kasten stehen die Eigenschaften, in UML Attribute genannt, die durch ein vorangestelltes Minus-Zeichen markiert sind.

In unserem Fall sind die 3 Attribute allesamt Zeichenketten, auf englisch „strings“. Im dritten Kasten stehen die Aktionen, in UML Methoden genannt, gekennzeichnet durch ein vorangestelltes Plus-Zeichen. Hier ist Umbrello in Aktion, das nicht für eine bestimmte Programmiersprache entworfen wurde, sondern für viele Programmiersprachen einsetzbar ist. In fast allen anderen Programmiersprachen muss man bei der Deklaration einer Variablen angeben, welchen Typ diese Variable zum Inhalt haben soll, etwa, dass die Variable vom Typ „int“ (für integer, also ganze Zahlen) oder vom Typ „string“ (für Zeichenketten) sein soll. In Python ist das anders: durch die erste Zuweisung eines Wertes an eine Variable wird die Klasse festgelegt, zu der die Variable gehört. In unserem Fall muss man den Methoden jeweils eine Information mitgeben: der Methode `nachnamenAendern` den neuen Nachnamen, der Methode `umziehen` den neuen Wohnort. Das, was einer Methode mitgegeben wird, wird als Parameter der Methode bezeichnet.

Methoden sind im Prinzip nicht viel anders als Funktionen, mit dem Unterschied, dass sie an eine Klasse gebunden sind und nur von Objekten der Klasse benutzt werden können. Das kommt Dir wohl schon bekannt vor, denn oben haben wir den Begriff des Namensraums eingeführt. Damit kann dieser Sachverhalt auch anders beschrieben werden: der Namensraum einer Methode ist die Klasse. Ebenso gilt, dass ein Attribut eine Variable ist, die als Namensraum die Klasse hat.

Eigentlich sollte es überflüssig sein, das nochmals zu erwähnen, aber trotzdem: Attribute sind Variablen, Methoden sind Funktionen, also beginnen Attribut- und Methodennamen mit einem Kleinbuchstaben.

9.2. Objekte, Objekte, Objekte

Jede Klasse (für uns im Moment jedenfalls!) hat eine Methode, mit Hilfe derer nach dem Modell der Klasse ein Objekt erzeugt werden kann: den Konstruktor. Das ist wie im Sandkasten: ich habe ein Burg-Förmchen, Sand rein, glatt streichen, umkippen, und schon habe ich eine Burg.

Dabei machen wir es von Anfang an richtig! Die Klasse kommt in eine eigene Datei, die wir `clMensch.py` nennen werden. [2] Und machen wir es Schritt für Schritt, wobei wir nach jedem Schritt testen werden, ob alles richtig funktioniert.

Beispiel 9.2.1 Eine erste Klasse

```
class Mensch():
  def __init__(self,  vorname,  nachname,  gebdat,  wohnort):
    self.vorname = vorname
    self.nachname = nachname
    self.gebdat = gebdat
    self.wohnort = wohnort
```

Eine Klasse wird definiert durch das Schlüsselwort `class`, gefolgt vom Klassennamen, gefolgt von einem Paar runder Klammern, in denen eventuell Parameter dieser Klasse stehen. Unsere Klasse bekommt im ersten Schritt nur einen Konstruktor. Der Konstruktor wird bei uns mit 5 Parametern aufgerufen: der erste Parameter, `self`, ist Pflicht!!!!! Danach folgen die Parameter, in denen die Werte für jedes Attribut, das wir deklariert haben, dem Konstruktor mitgegeben werden. In den folgenden Zeilen wird dem Attribut, gekennzeichnet durch das vorangestellte `self`, der jeweilige Parameter zugewiesen. An einem Beispiel verdeutlicht: der Parameter `vorname` aus der runden Klammer hinter dem Konstruktor-Namen `__init__` wird dem Attribut der Klasse, dem self.vorname zugewiesen.

Man sieht: Methoden sehen fast aus wie Funktionen. Es sind auch Funktionen, die aber nicht von überall gesehen werden und von überall benutzt werden dürfen. Es sind Funktionen, die an Klassen bzw. an Objekte dieser Klassen gebunden sind.

Die wichtigste Regel im Umgang mit Klassen in Python wiederhole ich also nochmals:

- **alles, was zu einer Klasse gehört, egal ob Attribut oder Methode, wird geschrieben als**
  - `klassenname.attribut`
  - bzw. `klassenname.methode`

  Klassenname und Attribut bzw. Klassenname und Methode werden durch einen Punkt voneinander getrennt. Das Schlüsselwort `self` verweist immer auf das aktuelle Objekt der Klasse, oft auch die „Instanz der Klasse“ genannt.

Zum Test benötigen wir dann noch eine zweite Datei, `menschAufruf.py`, in der wir ein Objekt der Klasse erzeugen.

Beispiel 9.2.2 Aufruf der Klasse (Erzeugung eines Objektes)

```
#!/usr/bin/python
from clMensch import Mensch

ich = Mensch('Martin', 'Schimmels', '29.12.1954', 'Rottenburg')
```

Zuerst muss das Aufruf-Programm natürlich die Klasse importieren. Erst dann kann man ein Objekt konstruieren. Dazu vergebe ich einen Variablennamen und teile dem Programm mit, dass die Variable jetzt ein Platzhalter für ein Objekt der Klasse `Mensch` ist. Das geschieht durch den Zuweisungsoperator =. Dadurch wird der Konstruktor der Klasse aufgerufen, und der benötigt eine ganze Menge Parameter.

[2] Es ist eine gute Sache, sich anzugewöhnen, Klassen-Dateien immer mit „cl“ (oder „cl_“ oder „kl“ oder so ähnlich) anfangen zu lassen. Sollte man später beruflich in einem Unternehmen programmieren dürfen, wird es sowieso Vorgaben geben, wie Dateien zu benennen sind.

Der Parameter `self` muss nicht übergeben werden, aber die anderen vier: Vorname, Nachname, Geburtsdatum und Wohnort. So! Objekt erzeugt!!! (Und wenn ich das Programm jetzt laufen lasse, tut sich gar nichts.)

Also wird unser aufrufendes Programm in einem zweiten Schritt erweitert:

Beispiel 9.2.3 Erzeugung eines Objekts

```
#!/usr/bin/python
from clMensch import Mensch

ich = Mensch('Martin', 'Schimmels', '29.12.1954', 'Rottenburg')
print ich.vorname()
print ich.nachname()
```

So! Jetzt tut sich etwas: Vor- und Nachname werden angezeigt.

Das, was ich im oben stehenden Beispiel gemacht habe, ist eigentlich absolut verboten: ausserhalb der Klassen-Definition soll man nicht auf Attribute zugreifen. (Siehe dazu weiter unten bei XrefId[??] Sichtbarkeit.) Für den Zugriff auf Attribute soll man immer eigene Methoden schreiben. Hier (und das wiederholt sich für viele unserer Beispiele) schreiben wir also einfach eine Methode, die alle Attribute eines Objektes und damit das ganze Objekt anzeigt. Damit ändert sich unsere Klassendatei so:

Beispiel 9.2.4 Die erste Klasse mit einer richtigen Methode

```
class Mensch():
     def __init__(self, vorname, nachname, gebdat, wohnort):
          self.vorname = vorname
          self.nachname = nachname
          self.gebdat = gebdat
          self.wohnort = wohnort

     def anzeigen(self):
          print self.vorname+' '+self.nachname
          print self.gebdat
          print self.wohnort
```

Die Regel von oben wurde auf jeden Fall eingehalten: **Die Methode `anzeigen` führt genau eine Aufgabe aus (nämlich die, die durch den Methoden-Namen vorgegeben ist).** Also muss dazu nicht mehr gesagt werden. Nach der Erzeugung eines Objektes greife ich jetzt also nicht mehr auf die Attribute des Objektes zu, sondern auf eine Methode!

Beispiel 9.2.5 Aufruf der Methode anzeigen

```
#!/usr/bin/python
from clMensch import Mensch

ich = Mensch('Martin', 'Schimmels', '29.12.1954', 'Rottenburg')
ich.anzeigen()
```

Wie angekündigt, Schritt für Schritt, wird das Programm unserem Modell angepasst. Die Klasse braucht eine Methode `nachnamenAendern`.

Beispiel 9.2.6 Eine weitere Methode

```
class Mensch():
   def __init__(self,  vorname,  nachname,  gebdat,  wohnort):
      self.vorname = vorname
      self.nachname = nachname
      self.gebdat = gebdat
      self.wohnort = wohnort

   def anzeigen(self):
      print self.vorname+' '+self.nachname
      print self.gebdat
      print self.wohnort

   def nachnamenAendern(self,  nachname):
      self.nachname = nachname
```

Das sollte klar sein. Auf zum Test!

Das Aufruf-Programm muss jetzt natürlich noch geändert werden, damit man die Änderung in der Klasse sieht.

Beispiel 9.2.7 Aufruf der neuen Methode

```
#!/usr/bin/python
from clMensch import Mensch

ich = Mensch('Martin', 'Schimmels', '29.12.1954', 'Rottenburg')
ich.anzeigen()
ich.nachnamenAendern('Meier')
ich.anzeigen()
```

Und auch das funktioniert prima!

Und zum (vorläufigen) Schluß wird jetzt noch die Methode `umziehen` geschrieben und getestet. Zuerst also die Änderung an der Klasse Mensch:

Beispiel 9.2.8 Noch eine Methode

```
class Mensch():
   def __init__(self,  vorname,  nachname,  gebdat,  wohnort):
      self.vorname = vorname
      self.nachname = nachname
      self.gebdat = gebdat
      self.wohnort = wohnort

   def anzeigen(self):
      print self.vorname+' '+self.nachname
      print self.gebdat
      print self.wohnort

   def nachnamenAendern(self,  nachname):
      self.nachname = nachname

   def umziehen(self, wohnort):
      self.wohnort = wohnort

```

Und dann das Aufruf-Programm:

Beispiel 9.2.9 Aufruf der neuen Methode

```
  #!/usr/bin/python
  from clMensch import Mensch

  ich = Mensch('Martin', 'Schimmels', '29.12.1954', 'Rottenburg')
  ich.anzeigen()
  ich.nachnamenAendern('Meier')
  ich.anzeigen()

  ich.umziehen('Ammerbuch')
  ich.anzeigen()

```

9.3. Kapselung

Die bisherigen Beispiele haben aus Sicht der Objektorientierung noch einen gewaltigen Fehler. Auf Attribute kann man beim obigen Beispiel von außerhalb der Klasse zugreifen, das heißt, dass in einem Programm folgendes möglich ist:

Beispiel 9.3.1 Keine Geheimnisse (leider)

```
  #!/usr/bin/python
  from clMensch import Mensch

  ich = Mensch('Martin', 'Schimmels', '29.12.1954', 'Rottenburg')
  print ich.nachname

```

Auf Attribute sollte nur mittels Methoden der Klasse zugegriffen werden. Python hat hier eine Schwäche, denn anders als in anderen objektorientierten Sprachen wird bei der Deklaration einer Variablen weder ein Datentyp noch eine Sichtbarkeitsklausel angegeben.

Es gibt 3 Stufen der Sichtbarkeit:

1. public: Jeder darf von überall alles mit dem Attribut machen.
2. protected: nur die eigene Klasse und daraus abgeleitete Klassen dürfen auf ein solches Attribut zugreifen.
3. private: Die stärkste Sicherheit: hier darf nur die eigene Klasse auf das Attribut zugreifen.

In Python wird die Sichtbarkeitsstufe durch vorangestellte Unterstriche dargestellt. Das ist leider keine absolute Sicherheit gegen einen missbräuchlichen Zugriff auf ein Attribut. Die Regeln in Python lauten:

1. Ein Attribut ohne einen führenden Unterstrich ist public, also auch von außerhalb der Klasse lesbar und veränderbar.
2. Ein Attribut mit genau einem führenden Unterstrich ist protected; damit ist noch keine Sicherheit gegeben, aber der Programmierer tut damit kund, dass man auf dieses Attribut nicht von außerhalb zugreifen sollte.
3. Ein Attribut mit genau zwei führenden Unterstrichen ist private; dieses Attribut ist von außerhalb der Klasse nicht sichtbar und kann somit nur durch Methoden der Klasse gelesen und geschrieben werden.

Tabelle 9.1. Sichtbarkeit und Kapselung

Stufe	Beispiel
public	vorname
protected	_vorname
private	__vorname

Also sollte man dieses Prögrämmchen noch richtig machen (richtig im Sinne der Objektorientierung). Die Attribute werden „privat“ gemacht, so dass nur noch über die Methoden zugegriffen werden kann.

Beispiel 9.3.2 Private Attribute

```
1
2  class Mensch():
3     def __init__(self, vorname, nachname, gebdat, wohnort):
4        self.__vorname = vorname
5        self.__nachname = nachname
6        self.__gebdat = gebdat
7        self.__wohnort = wohnort
8
9     def anzeigen(self):
10       print self.__vorname+' '+self.__nachname
11       print self.__gebdat
12       print self.__wohnort
13
14    def nachnamenAendern(self, nachname):
15       self.__nachname = nachname
16
17    def umziehen(self, wohnort):
18       self.__wohnort = wohnort
19
```

Die 3 Attribute `vorname`, `nachname` und `wohnort` haben im Konstruktor zwei Unterstriche vorangestellt. In der Methode `anzeigen` und den beiden Methoden, die Wohnort und Nachname ändern, müssen natürlich dann auch die Attribute mit vorangestelltem Unterstrich angesprochen werden.

Das aufrufende Programm ändert sich gar nicht — außer, dass in der Zeile nach dem ersten Anzeigen ein Fehler eingebaut wird: es wird nämlich der Versuch gemacht, auf das Attribut `__nachname` von außerhalb der Klasse zuzugreifen.

Beispiel 9.3.3 Aufruf mit Fehler

```
1
2   #!/usr/bin/python
3   from clMensch import Mensch
4
5   ich = Mensch('Martin', 'Schimmels', '29.12.1954', 'Rottenburg')
6   ich.anzeigen()
7   print ich.__nachname                # Das geht schief! Wenn diese Zeile
8                                       # entfernt wird, läuft das Programm
9                                       # natürlich einwandfrei.
10
11  ich.nachnamenAendern('Meier')
12  ich.anzeigen()
13
14  ich.umziehen('Ammerbuch')
15  ich.anzeigen()
16
```

9.4. Vererbung (ohne Erbschaftssteuer)

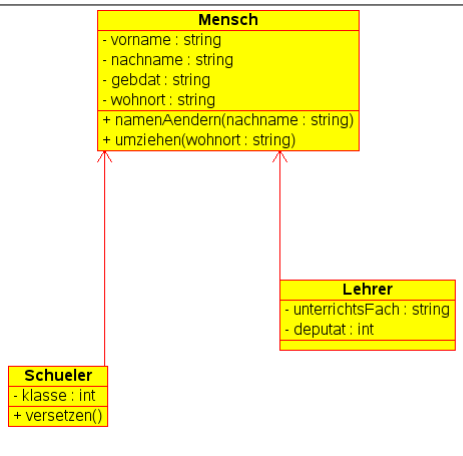
Richtig interessant, weil arbeitssparend, wird Objektorientierung aber erst, wenn Klassen ihre Eigenschaften vererben. Dazu bohren wir das Beispiel mit den Menschen ein wenig auf. Nehmen wir also typische Menschen an einer Schule, von der es wieder zwei Klassen gibt: die Schüler und die Lehrer. Die sind alle Menschen, haben also alle Eigenschaften der oben bearbeiteten Klasse Mensch, nämlich, wir erinnern uns, Vornamen, Nachnamen, Geburtsdatum und Wohnort.

Zusätzlich haben Schüler noch die Eigenschaft `Klasse`, das heißt sie gehen in beispielsweise die 11. Klasse. Außerdem können Schüler eine Aktion durchführen: die neue Methode heißt `versetzen`, und bedeutet, dass ein Schüler am Ende des Schuljahres von der 11. Klasse in die 12. Klasse wechselt (hoffentlich!). Um das Modell einfach zu halten und nachher auch die Methode `versetzen` schnell zu programmieren, ist die Klassenangabe nur eine Zahl zwischen 1 und 13 (keine Parallelklassen wie z.B. 12d oder 11/1).

Lehrer hingegen haben die zusätzliche Eigenschaft, dass sie ein „Unterrichtsfach“ unterrichten (um das Modell einfach zu halten, beschränken wir uns hier auf **ein** Fach). Außerdem hat ein Lehrer noch ein Deputat, d.h. eine bestimmte Anzahl Stunden, die er unterrichten muss.

Und so sieht das Modell in UML aus:

Abbildung 9.2. Klassendiagramm `Schueler`



Hier werden 3 Klassen abgebildet, und diese Klassen sind durch Pfeile miteinander verbunden. Der Pfeil zeigt immer auf die übergeordnete Klasse, von der die untergeordneten Klassen etwas (in diesem Beispiel: alles!) erben. Umbrello spricht hier von einer „gerichteten Assoziation“. Die Eigenschaften, die

die untergeordneten (in UML: abgeleiteten) Klassen erben, müssen dort nicht mehr aufgeführt werden. Alles Gute kommt von oben!

Die Realisierung in Python sieht so aus:

Beispiel 9.4.1 Abgeleitete Klasse

```
1
2  class Mensch():
3     def __init__(self,  vorname,  nachname,  gebdat,  wohnort):
4        self.vorname = vorname
5        self.nachname = nachname
6        self.gebdat = gebdat
7        self.wohnort = wohnort
8
9     def anzeigen(self):
10       print self.vorname+' '+self.nachname
11       print self.gebdat
12       print self.wohnort
13
14    def nachnamenAendern(self,  nachname):
15       self.nachname = nachname
16
17    def umziehen(self, wohnort):
18       self.wohnort = wohnort
19
20 class Schueler(Mensch):
21    def __init__(self, vorname,  nachname,  gebdat,  wohnort,  klasse):
22       Mensch.__init__(self,  vorname,  nachname,  gebdat,  wohnort)
23       self.klasse = klasse
24
25    def versetzen(self):
26       self.klasse += 1
27
```

Auch hier gibt es eine kurze Erläuterung: in die selbe Datei wie die Klasse `Mensch` schreibe ich die Klasse `Schueler` (keine Umlaute!!!). Um Python anzuweisen, dass es Schueler als eine Unterklasse von Mensch auffassen soll, wird dem Klassennamen in Klammern die Bezeichnung der Oberklasse mitgeteilt. Der Konstruktor nimmt wieder alle Attribute in Empfang, aber reicht gleich die Attribute, die die Klasse `Schueler` von der Klasse `Mensch` erbt, weiter an den Konstruktor von Mensch nach dem Motto: „Mach Du das doch, Du weißt doch schon, wie das geht.“ Einzig die Klasse muss hier initialisiert werden.

Die neue Methode ist wohl ganz leicht zu verstehen. Es wird der Kurzschluß-Operator für die Addition benutzt.

Jetzt brauchen wir wieder ein Aufruf-Programm, mit dem man die neue Klassen-Definition testen kann. Hier müssen zwei Klassen-Definitionen importiert werden, die durch Komma voneinander getrennt werden:

Beispiel 9.4.2 Aufruf der abgeleiteten Klasse

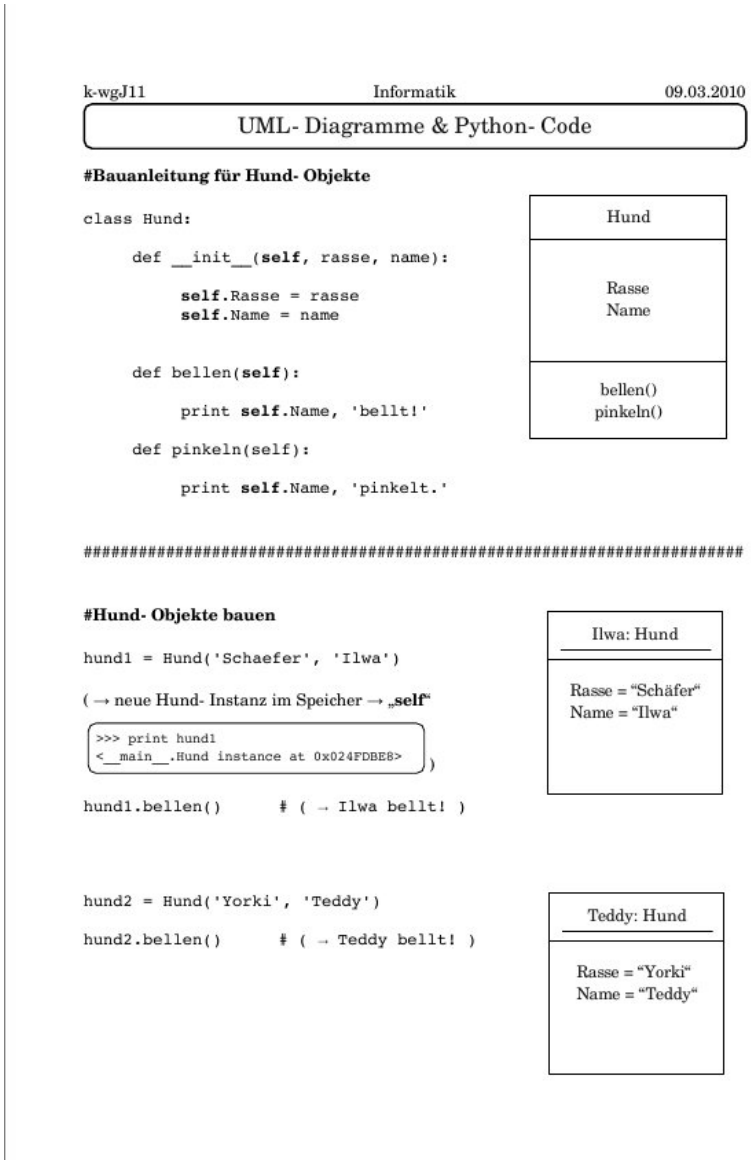
```
1
2   #!/usr/bin/python
3   from clMensch import Mensch, Schueler
4
5   ich = Mensch('Martin', 'Schimmels', '29.12.1954', 'Rottenburg')
6
7   ich.anzeigen()
8   ich.nachnamenAendern('Meier')
9   ich.anzeigen()
10  ###
11  # ab hier gibt es was neues
12  ###
13  thomas = Schueler('Thomas', 'Renz', '12.11.1989', 'Dettenhausen', 11)
14
15  thomas.anzeigen()
16
17  thomas.versetzen()
18  thomas.anzeigen()
19
```

Was noch zu tun ist:

- Der Schüler benötigt noch eine Methode `anzeigen`. Dabei sollte selbstverständlich die Methode `anzeigen` von `Mensch` benutzt werden, so weit das möglich ist.
- Die Klasse `Lehrer` muss noch geschrieben werden.

10. Arbeitsblätter zu Klassen (Danke, Jens!)

10.1. UML-Diagramme und Python-Code



k-wgJ11 Informatik 09.03.2010

UML- Diagramme & Python- Code

#Bauanleitung für Hund- Objekte

```
class Hund:

     def __init__(self, rasse, name):

          self.Rasse = rasse
          self.Name = name


     def bellen(self):

          print self.Name, 'bellt!'

     def pinkeln(self):

          print self.Name, 'pinkelt.'
```

Hund
Rasse
Name
bellen()
pinkeln()

```
########################################################################
```

#Hund- Objekte bauen

```
hund1 = Hund('Schaefer', 'Ilwa')
```

(→ neue Hund- Instanz im Speicher → „**self**“

```
>>> print hund1
<__main__.Hund instance at 0x024FDBE8>
```

)

```
hund1.bellen()      # ( → Ilwa bellt! )
```

Ilwa: Hund
Rasse = “Schäfer“
Name = “Ilwa“

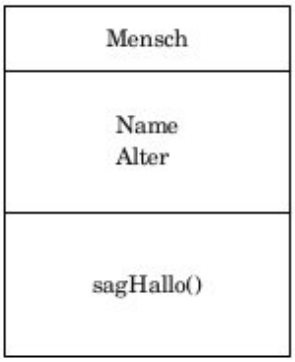
```
hund2 = Hund('Yorki', 'Teddy')

hund2.bellen()      # ( → Teddy bellt! )
```

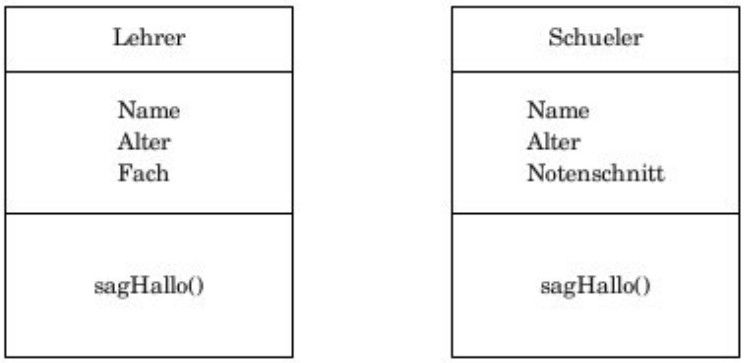
Teddy: Hund
Rasse = “Yorki“
Name = “Teddy“

10.2. Vererbung

k-wgJ11 Informatik 23.03.2010

Alle sind Menschen –





Python – Code:

```
class Schueler

    def __init__(self, name, alter):
```

10.3. Objektvariable vs. Klassenvariable

k-wgJ11 Informatik 26.03.2010

Objektvariablen vs. **Klassenvariablen** (& Modularisierung)

Während Objektvariablen zum Speichern von Attributen eines bestimmten Objekts einer Klasse verwendet werden, benötigen wir **Klassenvariablen** um Eigenschaften der gesamten Klasse zu speichern.

Beispiel: Anzahl bereits initialisierter Personen

Fachklasse

```
class Person:

     anzahl_personen = 0

     def __init__(self,name,email):

          self.name = name
          self.email = email

          Person.anzahl_personen += 1

     def ausgeben(self):

          print 'Ich bin eine Person und heisse',self.name
          print 'Mit mir sind wir nun',Person.anzahl_personen
```

Startklasse

```
from Person import *

p1 = Person('Jens Stephan','stephan@wilhelm-schickard-schule.de')

p1.ausgeben()


p2 = Person('Christoph Lederle','lederle@wilhelm-schickard-
schule.de')

p2.ausgeben()
```

Ausgabe

```
>>>
Ich bin eine Person und heisse Jens Stephan
Mit mir sind wir nun 1
Ich bin eine Person und heisse Christoph Lederle
Mit mir sind wir nun 2
```

10.4. Polymorphie

k-wgJ11 Informatik 26.03.2010

Polymorphie, oder „Was bin ich?“

```
class Person:

      anzahl_personen = 0

      def __init__(self,name,email):

            self.name = name
            self.email = email

            Person.anzahl_personen += 1

      def ausgeben(self):

            print 'Ich bin eine Person und heisse',self.name
            print 'Mit mir sind wir nun',Person.anzahl_personen
```

```
from Person import *

class Schueler(Person):
    anzahl_schueler = 0

    def __init__(self,name,email):
        Person.__init__(self, name, email)
        Schueler.anzahl_schueler += 1

    def ausgeben(self):

        print 'Ich bin ein Schueler und heisse',self.name
        print 'Mit mir sind wir nun',Person.anzahl_personen, 'Personen inkl.',
Schueler.anzahl_schueler, 'Schueler'
```

```
from Person import *
from Schueler import *

p1 = Person('Jens Stephan','stephan@wilhelm-schickard-schule.de')

p2 = Person('Christoph Lederle','lederle@wilhelm-schickard-schule.de')

p3 = Schueler('Annika Otto', 'annika@irgendwo.de')

p1.ausgeben()

p2.ausgeben()

p3.ausgeben()
```

```
>>>
Ich bin eine Person und heisse Jens Stephan
Mit mir sind wir nun 3
Ich bin eine Person und heisse Christoph Lederle
Mit mir sind wir nun 3
Ich bin ein Schueler und heisse Annika Otto
Mit mir sind wir nun 3 Personen inkl. 1 Schueler
```

10.5. Übungsaufgabe

k-wgJ11 Informatik 13.04.2010

Übungsaufgabe zu UML, Klassen, Vererbung, Attributen, Methoden (Parameter), Polymorphie und Klassenvariablen + KAPSELUNG

1. Erstellen Sie aus dem Text ein UML- Diagramm der benötigten Klassen.

   Polizisten haben einen Namen und eine Dienstnummer; sie können jemanden festnehmen („*Herr oder Frau ... , sie sind festgenommen!*“) und sie können sich ausweisen („*Guten Tag. Mein Name ist ... . Ich bin Polizist und habe die Dienstnummer ... .*“)

   Verdeckte Ermittler (Undercover) sind auch Polizisten, haben zusätzlich aber einen Decknamen. Wenn sie sich ausweisen sagen sie „*Guten Tag. Mein Name ist ... . Ich bin Verdeckter Ermittler und habe die Dienstnummer ... .*“

2. Erstellen Sie den Python- Code für jede Klasse in einer eigenen Datei und legen Sie zusätzlich eine Startklasse an, in der Sie ein Polizisten- und ein Undercover-Objekt erstellen (`p1`, `u1`), sowie beide sich ausweisen und Herrn XY festnehmen lassen.

3. Implementieren Sie für die Klasse Polizist eine Methode `zaehlen()`, mit der sich ausgeben lässt: „*Wir sind ... Polizisten.*“ und für die Klasse Undercover das Entsprechende.

4. Benutzen Sie die Methode `zaehlen()` in der Startklasse.

   Versuchen Sie auch folgendes: `print u1.Deckname`

Kapselung:

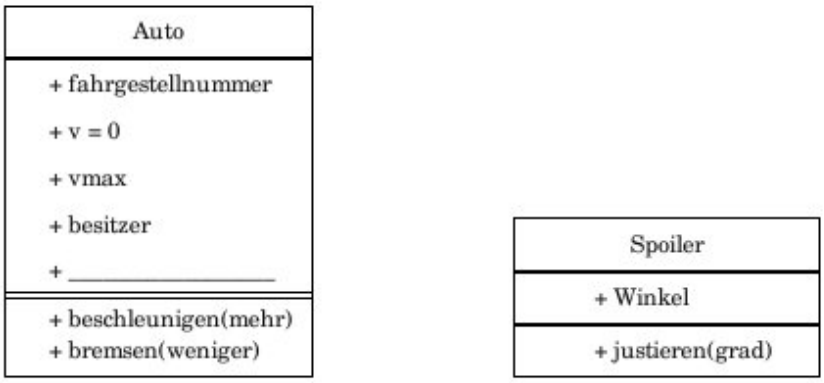

```
def getDeckname(self):
    return self.Deckname

def setDeckname(self,neuname):
    self.Deckname = neuname
```

10.6. Assoziationen Teil 1

k-wgJ11 Informatik 04.05.2010

Kommunikation von Objekt zu Objekt - „Assoziationen“

Abhängig von der Geschwindigkeit eines Autos soll dessen Heckspoiler automatisch seinen Neigungswinkel justieren um die Straßenhaftung zu optimieren.



Alle Methoden sollen über eine `print`- Anweisung berichten, was verändert wurde!

`beschleunigen`()
Diese Methode soll die Geschwindigkeit (v) des Autos um „mehr“ km/h erhöhen. Es ist sicherzustellen, dass die Höchstgeschwindigkeit nicht überschritten wird!

`bremsen`()
Diese Methode soll die Geschwindigkeit (v) des Autos um „weniger“ km/h reduzieren. Es ist sicherzustellen, dass die Geschwindigkeit nicht unter 0 km/h sinkt!

`justieren`()
Diese Methode soll den Neigungswinkel des Spoilers auf „grad“ Grad verändern.

10.7. Klassenarbeit

k-wgJ11 Informatik 11.05.2010

Klassenarbeit

Name: **Punkte:** von 24 **Notenp.:**

Aufgabe 1

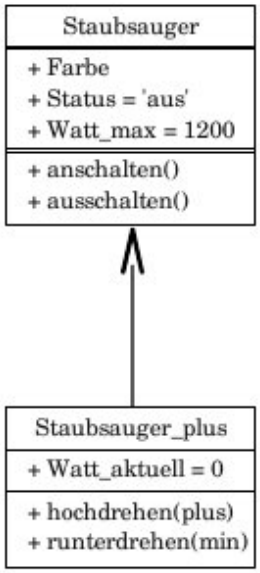
Erstellen Sie aus folgendem Sachverhalt ein UML-**Klassendiagramm** für die benötigten Klassen des betrachteten Auftragsverwaltungsprogramms. **(7)**

Für die Auftragsverarbeitung wird in einem Unternehmen zwischen Privatkunden und Firmenkunden unterschieden. Von allen Kunden werden Straße, Postleitzahl, Ort, Telefonnummer, Bankleitzahl und Kontonummer gespeichert. Allen Kunden kann eine Rechnung zugesandt werden.

Von Privatkunden wird zusätzlich der Name benötigt, von Firmenkunden die Firma (Name des Unternehmens) und ein Ansprechpartner. Firmenkunden können zu „Kundenpflegeterminen" eingeladen werden.

Aufgabe 2

Unser Unternehmen verkauft vollautomatische, per Internet steuerbare Bodenstaubsauger. Geben Sie den Python-Code für die folgende Klasse Staubsauger an. Die beiden Methoden sollen den Status des Geräts (an/aus) entsprechend verändern und ausgeben, dass das Gerät jetzt aktiv, bzw. nicht aktiv ist. **(10)**



Aufgabe 3

Unsere neueste Weiterentwicklung ist stufenlos regulierbar, aber noch etwas unausgereift.

Geben Sie den Code für die Methode „**hochdrehen**" an, die

- prüft, ob der Staubsauger_plus überhaupt angeschaltet ist
- läuft er, wird die aktuelle Wattzahl um „plus" Watt erhöht und ausgegeben, ansonsten wird eine Fehlermeldung ausgegeben

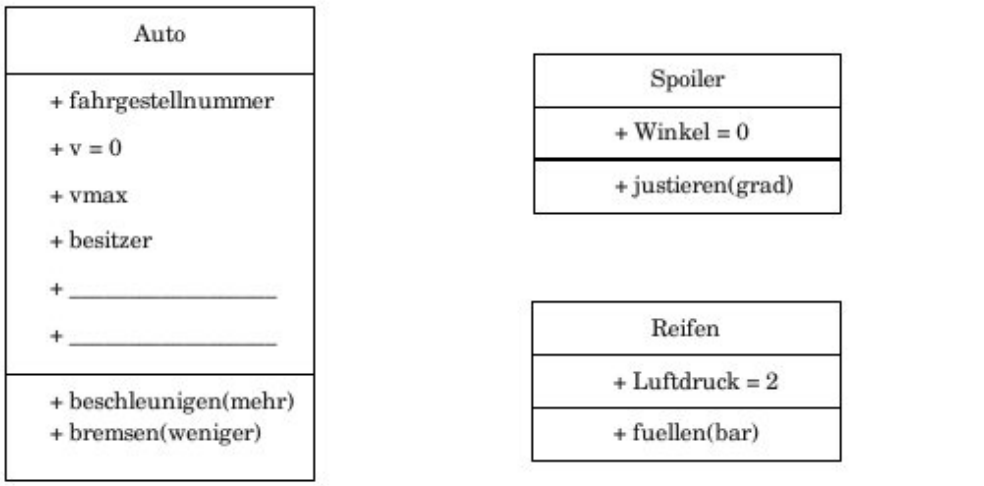
(7)

Viel Erfolg!

10.8. Assoziationen Teil 2

k-wgJ11 Informatik 11.06.2010

Kommunikation von Objekt zu Objekt - „Assoziationen“



- Assoziation mittels einer

Zu lang für ein Zeile ...

```
class Auto:

    def __init__(self, fahrgestellnummer, hoechstgeschwindigkeit,
besitzer, spoiler):

        self.Fahrgestellnummer = fahrgestellnummer
        self.Hoechstgeschwindigkeit = hoechstgeschwindigkeit
        self.Besitzer = besitzer
        self.Geschwindigkeit = 0
        self.Spoiler = spoiler

        self.Reifen =
```

10.9. Liste von Objekten

k-wgJ11 Informatik 29.06.2010

Zugriff auf eines der „n“ zugeordneten Objekte (Liste) in einer 1:n - Beziehung

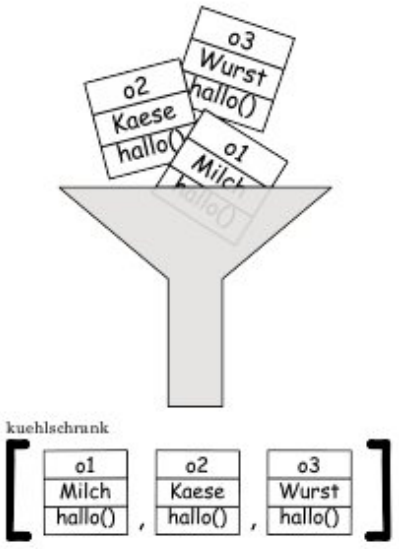
Klasse für unsere Objekte:

```
class IrgendeinObjekt:

     def __init__(self, name):

          self.Name = name

     def hallo(self):

          print 'Hallo vom Objekt mit Namen' ,  self.Name,'.'
```

IrgendeinObjekt
+ Name
+ hallo()

Startklasse:

```
from IrgendeinObjekt import *

o1 = IrgendeinObjekt('Milch')
o2 = IrgendeinObjekt('Kaese')
o3 = IrgendeinObjekt('Wurst')

kuehlschrank = []


kuehlschrank.append(o1)
kuehlschrank.append(o2)
kuehlschrank.append(o3)


for o in kuehlschrank:
     o.hallo()
```



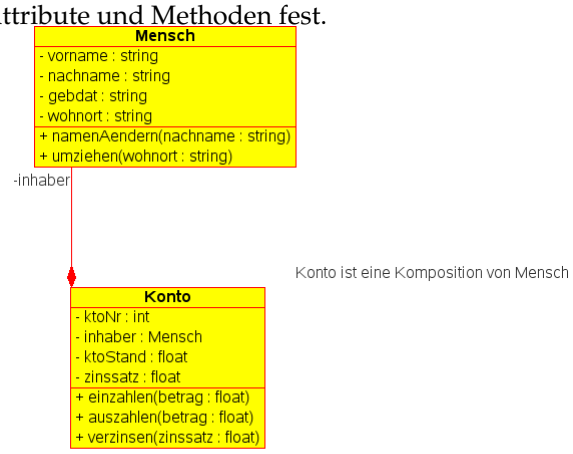
Ausgabe:

```
>>>
```

11. Ein etwas längeres Programm

11.1. Objektorientierter Entwurf

Stellen wir uns vor, wir sollten ein Programm schreiben, das Konten einer Bank verwaltet. Den Entwurf wollen wir wieder in einzelnen Schritten durchführen, wobei wir am Anfang sehr grob vorgehen, und später das Erarbeitete verfeinern. In diesem Sinn legen wir für die Klasse `Konto` zuerst nur die nötigsten Attribute und Methoden fest.



Das Besondere an einem Konto fällt (hoffentlich) sofort auf: der Inhaber des Kontos ist ein Mensch! (Als Aussage des alltäglichen Lebens ist das wirklich keine Weisheit. Als Aussage in unserem Klassen-Entwurf ist das etwas völlig Neues!) In der Sprache UML ausgedrückt: eines der Attribute der Klasse Konto ist ein Objekt der Klasse Mensch. Dies wird **Aggregation** genannt.

11.2. Objektorientierte Programmierung

11.2.1. Die Klasse `Konto`

Die Umsetzung der Klassendefinition in Python sieht so aus:

Beispiel 11.2.1 Die Klasse „Konto“

```
1
2  from clMensch import Mensch
3
4  class Konto(object):
5
6     def __init__(self, ktoNr, vorname, nachname, gebdat, wohnort,
7                        ktoStand,  zinssatz):
8        self.ktoNr = ktoNr
9        self.inhaber = Mensch(vorname, nachname, gebdat, wohnort)
10       self.ktoStand = ktoStand
11       self.zinssatz = zinssatz
12
13    def einzahlen(self, betrag):
14       pass
15
16    def auszahlen(self, betrag):
17       pass
18
19    def verzinsen(self, zinssatz):
20       pass
21
```

Das wird wieder in einer eigenen Datei gespeichert, in clKonto.

11.2.2. Das aufrufende Programm

Zuerst wird der Mensch importiert, denn den benötigen wir für den Inhaber. Der Konstruktor weist die eigentlichen Konto-Attribute Konto-Nummer, Konto-Stand und Zinssatz direkt zu. Die personenbezogenen Attribute werden nicht zugewiesen, sondern es wird ein Objekt der Klasse Mensch erzeugt, das heißt mit der Zeile `self.inhaber = Mensch(vorname, nachname, gebdat, wohnort)` wird der Konstruktor der Klasse Mensch aufgerufen. Die Methoden sind noch nicht ausprogrammiert, statt dessen steht nur ein Methodenrumpf da mit der leeren Anweisung `pass`, die nichts bewirkt.

Wieder benötigen wir ein Aufruf-Programm, mit dem wir diesen Klassen-Entwurf testen können:

Beispiel 11.2.2 Aufruf der Klasse Konto

```
#!/usr/bin/python

from clKonto import Konto

meinKonto = Konto('101010', 'Martin', 'Schimmels', '29.12.1954', 'Rottenburg',
                                100.00, 0.02)
print meinKonto.inhaber.vorname, meinKonto.inhaber.nachname,
                                '\n',  meinKonto.inhaber.wohnort
print 'Kontonr.:', meinKonto.ktoNr, ' Kontostand: ', meinKonto.ktoStand
```

11.2.3. Realisierung der Methoden „Einzahlen“ und „Auszahlen“

Einzahlungen und Auszahlungen müssen nun programmiert werden. Das sind aber sehr kurze Methoden.

Beispiel 11.2.3 Realisierung der ersten Methoden

```
1
2  from clMensch import Mensch
3
4  class Konto(object):
5
6     def __init__(self, ktoNr, vorname, nachname, gebdat, wohnort,
7                        ktoStand, zinssatz):
8        self.ktoNr = ktoNr
9        self.inhaber = Mensch(vorname, nachname, gebdat, wohnort)
10       self.ktoStand = ktoStand
11       self.zinssatz = zinssatz
12
13    def einzahlen(self, betrag):
14       self.ktoStand += betrag
15
16    def auszahlen(self, betrag):
17       self.ktoStand -= betrag
18
19    def verzinsen(self, zinssatz):
20       pass
21
```

Und das Aufruf-Programm ruft jede der beiden Methoden einmal auf und gibt jeweils anschließend den aktuellen Stand aus:

Beispiel 11.2.4 Aufruf der beiden neuen Methoden

```
1
2   #!/usr/bin/python
3
4   from clKonto import Konto
5
6   meinKonto = Konto('101010', 'Martin', 'Schimmels', '29.12.1954',
7                                 'Rottenburg', 100.00, 0.02)
8   print meinKonto.inhaber.vorname, meinKonto.inhaber.nachname,
9                                 '\n',  meinKonto.inhaber.wohnort
10  print 'Kontonr.:', meinKonto.ktoNr, ' Kontostand: ',
11                                meinKonto.ktoStand
12
13  meinKonto.einzahlen(111.23)
14  print meinKonto.inhaber.vorname, meinKonto.inhaber.nachname, '\n',
15                                meinKonto.inhaber.wohnort
16  print 'Kontonr.:', meinKonto.ktoNr, ' Kontostand: ', meinKonto.ktoStand
17
18  meinKonto.auszahlen(5.01)
19  print meinKonto.inhaber.vorname, meinKonto.inhaber.nachname, '\n',
20                                meinKonto.inhaber.wohnort
21  print 'Kontonr.:', meinKonto.ktoNr, ' Kontostand: ', meinKonto.ktoStand
22
```

11.2.4. Die Verzinsung

Es fehlt noch die Verzinsung. Auch hier muss wieder eine Vereinbarung für den Test und damit das ordentliche Funktionieren der Klasse Konto getroffen werden, denn eine an der Realität orientierte Verzinsungs-Methode können wir nicht testen. Wer will schon ein Jahr vor dem Rechner sitzen, und warten, bis der Zeitpunkt der Verzinsung gekommen ist!!!

Deswegen wird hier das Modell folgendermaßen abgeändert: jede Transaktion, sei es Einzahlung oder Auszahlung, wird gezählt, und nach 3 Transaktionen wird die Verzinsung angestoßen.

Beispiel 11.2.5 Die Verzinsung

```
1
2  from clMensch import Mensch
3
4  class Konto(object):
5     taZaehler = 0
6
7     def __init__(self, ktoNr, vorname, nachname, gebdat, wohnort,
8                        ktoStand,  zinssatz):
9        self.ktoNr = ktoNr
10       self.inhaber = Mensch(vorname, nachname, gebdat, wohnort)
11       self.ktoStand = ktoStand
12       self.zinssatz = zinssatz
13
14    def einzahlen(self, betrag):
15       self.ktoStand += betrag
16       self.taZaehler += 1
17       if self.taZaehler % 3 == 0:
18           self.verzinsen(self.zinssatz)
19
20    def auszahlen(self, betrag):
21       self.ktoStand -= betrag
22       self.taZaehler += 1
23       if self.taZaehler % 3 == 0:
24           self.verzinsen(self.zinssatz)
25
26    def verzinsen(self, zinssatz):
27       self.ktoStand += self.ktoStand*self.zinssatz
28
```

Hinzugekommen ist das Attribut `taZaehler`, das die Transaktionen zählt. Zu Beginn wird dieser Zähler auf 0 gesetzt, in den einzelnen Methoden wird er hochgezählt (Vorsicht! Es muss natürlich mit `self.taZaehler` angesprochen werden!) und nach dem Hochzählen wird abgefragt, ob er durch 3 teilbar ist. Wenn ja, wird die Verzinsung angestossen. Das Aufruf-Programm ändert sich vorerst kaum, da wir ja immer noch in der Test-Phase sind, es werden einzig die Befehle zum Einzahlen und zur Anzeige mehrmals kopiert, so dass man mehrere Transaktionen bekommt.

Es gibt natürlich auch hier noch einiges zu verbessern.

1. Zuerst einmal entspricht es keinem guten Stil, Attribute eines Objektes direkt anzusprechen; das sollte immer mittels einer Methode der Klasse geschehen.
2. Die Anzeige nach der Verzinsung gibt einen Kontostand mit mehr als 2 Dezimalstellen aus. Das verwirrt auch den wohlwollendsten Bankkunden auf Dauer.
3. Das Aufruf-Programm ist ein Skandal! Immer wieder eine Folge von Befehlen zu kopieren ist mit dem Berufsethos eines Programmierers nicht zu vereinbaren: da muss eine Schleife mit einer sinnvollen Abbruchbedingung her.
4. Dabei wird auch endlich ein ordentliches „Buchungsverfahren“ verwirklicht, das heißt, dass nicht mehr im Aufruf-Programm die Buchungen fest verdrahtet sind, sondern dass man über Tastatur-Eingaben Einzahlungen und Auszahlungen macht.

Das alles wird im nächsten Kapitel behandelt.

11.2.5. Die ersten Verbesserungen

Schreiben wir also zuerst eine Methode, die die Informationen über ein Konto ausgibt. (Hier wird nur die neue Methode ausgegeben, die als vorläufig letzte Methode in die Klasse eingefügt wird.)

Beispiel 11.2.6 Eine Ausgabe-Methode

```
def kontoStandAusgeben(self):
   print self.inhaber.vorname,  self.inhaber.nachname
   print 'geboren am', self.inhaber.gebdat, ', wohnhaft in ',
          self.inhaber.wohnort
   print 'Kontonummer: ', self.ktoNr, ', Kontostand: ', self.ktoStand
```

Das Aufruf-Programm ändert sich dadurch, es wird kürzer, weniger fehleranfällig, überschaubarer; kurz, es wird schöner!

Beispiel 11.2.7 Verbesserungen des Aufruf-Programms

```
1
2  #!/usr/bin/python
3
4  from clKonto import Konto
5
6  meinKonto = Konto('101010', 'Martin', 'Schimmels', '29.12.1954',
7                                  'Rottenburg', 100.00, 0.02)
8  meinKonto.kontoStandAusgeben()
9
10 meinKonto.einzahlen(111.23)
11 meinKonto.kontoStandAusgeben()
12
13 meinKonto.auszahlen(5.01)
14 meinKonto.kontoStandAusgeben()
15
16 meinKonto.einzahlen(100.00)
17 meinKonto.kontoStandAusgeben()
18
19 meinKonto.einzahlen(100.00)
20 meinKonto.kontoStandAusgeben()
21
```

Die zweite Verbesserung ist nur eine Kleinigkeit. Nur die neu geschaffene Methode `kontoStandAusgeben` muss an einer Stelle verändert werden. Siehe dazu im Kapitel über Zeichenketten den Abschnitt über Formatierung von Zeichenketten. Der Quelltext der veränderten Klassenmethode sieht so aus:

Beispiel 11.2.8 Verbesserung der Ausgabe-Methode

```
def kontoStandAusgeben(self):
  print self.inhaber.vorname,  self.inhaber.nachname
  print 'geboren am',  self.inhaber.gebdat, ', wohnhaft in ',
                self.inhaber.wohnort
  print 'Kontonummer: %12s  Kontostand: %+8.2f' %
                (self.ktoNr, self.ktoStand)
```

Jetzt kommt die wirklich interessante Veränderung. Es wird nicht mehr fest im Programm verdrahtet, wieviele Eingaben gemacht werden sollen, sondern der Benutzer entscheidet, wann das Programm beendet werden soll: machen wir noch eine Schleife drum!

Beispiel 11.2.9 Konten bearbeiten in einer Schleife

```
1
2  #!/usr/bin/python
3  # -*- coding: utf8 -*-
4  from clKonto import Konto
5
6  meinKonto = Konto('101010', 'Martin', 'Schimmels', '29.12.1954',
7                                  'Rottenburg', 100.00, 0.02)
8  meinKonto.kontoStandAusgeben()
9
10 def meld():
11    print "Konto-Verwaltung\ne für Einzahlung\na für Auszahlung\ns für Schluss"
12
13 eingabe = 'q'
14 while (eingabe <> 's'):
15    meld()
16    eingabe = raw_input('Bitte Eingabe:')
17    if eingabe == 'e':
18       betrag = input('Einzahlungsbetrag: ')
19       meinKonto.einzahlen(betrag)
20    elif eingabe == 'a':
21       betrag = input('Auszahlungsbetrag: ')
22       meinKonto.auszahlen(betrag)
23    elif eingabe == 's':
24       print 'und tschüß'
25    else:
26       print "Falsche Eingabe. nur a, e, s erlaubt"
27    meinKonto.kontoStandAusgeben()
28
```

Wieder könnte man sich entspannt zurücklehnen. Aber ganz zufrieden ist man nie, das ganze soll noch besser werden. (Ganz ehrlich: so könnten wir das Programm noch keiner Bank verkaufen!) Was zu tun ist? Das:

1. Das aufrufende Programm aus dem letzten Programm-Listing ist aus der Sicht des objektorientierten Programmierers nicht in Ordnung. Das, was in diesem Programm steht, ist eigentlich ein Menu für die Anwendung. Ein Menu ist aber wieder ein Objekt einer Menu-Klasse. Man könnte sich also überlegen, wie man eine solche Menu-Klasse modelliert, wie man ein Objekt dieser Klasse erzeugt und wie man dann dieses Objekt arbeiten lässt. Der Konjunktiv steht hier, weil das für dieses einfache Programm erst mal nicht nötig ist, um das Prinzip der Objektorientierung an dem Beispiel der Klasse `Konto` zu überdenken. In einem der nächsten Beispiele wird das genau durchgeführt, dass eine Klasse `Menu` erstellt wird. Ein bißchen Geduld also bitte!
2. Ein Konto ist ja ganz gut und schön, aber mit nur einem Konto wird keine Bank glücklich. Es sollten schon mehrere Konten möglich sein.
3. Wenn der Rechner ausgeschaltet wird, sind alle Informationen verloren. So geht das nicht. Die Daten sollten also gespeichert werden, (spätestens) wenn das Programm beendet wird.

Das alles wird im nächsten Kapitel behandelt.

11.2.6. Übungen zu Klassen

1. Es soll eine Klasse `Buch` geschrieben werden, die die Menge der Bücher in einer Leihbücherei beschreibt. Dazu soll das Buch nur die Attribute `Autor`, `Titel` und `ausgeliehen` sowie die Methoden `ausleihen` und `zurückgeben` haben. Ferner soll ein aufrufendes Programm geschrieben werden, das diese Klasse benutzt.
2. Eine Klasse `Bruch` soll geschrieben werden, die Brüche darstellt und die Methoden kürzen, addieren, subtrahieren, multiplizieren und dividieren enthält, außerdem ein aufrufendes Programm, mit dem man diese Klasse testen kann.

Teil VII.

Fehler und Ausnahmen

12. Fehler...

Because something is happening here
And you don't know what it is

(Bob Dylan [1])

12.1. .. macht jeder

Vor allem jeder Programmierer macht Fehler. Bei manchen Programmen ist es ganz in Ordnung, dass das Programm abbricht, wenn ein Fehler passiert. Das ist nicht weiter schlimm, wenn das Programm nicht gleich das ganze Betriebssystem mit ins Nirwana zieht — wer erinnert sich da noch an das alte MS-DOS?!?!

Ein Freund hat mir vor ein paar Tagen beschrieben, was in der Institution, für die er ehrenamtlich arbeitet, passiert ist. Diese Institution hat Mitglieder, Angestellte, Kosten, Einnahmen von Mitgliedern, Einnahmen aus Geschäften und viele andere Dinge, die — im 21. Jahrhundert selbstverständlich — auf einem Rechner bearbeitet werden. Wie jedes Unternehmen ist diese Institution darauf bedacht, dass diese Informationen erhalten bleiben, auch wenn auf den Rechnern etwas Unvorhergesehenes geschieht. Eine Datensicherung muss also eingerichtet werden.

Bei der Konzeption des Rechnersystems und der Software war das auch beachtet worden. Datensicherung sollte täglich geschehen, und diese Sicherungen, also verschiedene Versionen mit jeweils unterschiedlichem Datum, wurden eine bestimmte Zeit aufbewahrt. Bei einer Überprüfung des Rechnersystems, die gemacht wurde, weil Antwortzeiten inakzeptabel geworden waren, stellte der Überprüfende fest: seit 6 Monaten war keine Sicherung mehr geschrieben worden und eine ganze Gruppe von Daten wurde noch überhaupt nie gesichert. (Geplant war das alles, und Diejenigen, die das System erstellt haben, bestätigten, dass diese Sicherungen eingebaut worden waren, aber den Grund, warum es in der Realität nicht oder nicht mehr funktionierte, wusste mein Bekannter noch nicht.)

Wie kann man als Informatiker (oder Programmierer oder sonst irgendein IT-Fachmann) mit einem Programm(-System) in einem solchen Fall umgehen? Es gibt verschiedene Ansätze:

1. In dem oben geschilderten Fall ist die zuständige Person nach dem Motto vorgegangen: ich habe das programmiert (oder die Programme eingerichtet), ich habe es einmal (oder 10 mal oder noch öfter) getestet, es hat funktioniert, also wird es auch weiterhin immer funktionieren.

   Nichts zu tun: das ist, bei meinem Bekannten hat sich das bestätigt, für die für die Einrichtung des Systems zuständige Person die einfachste, für den Benutzer oder den Kunden aber sicher die im Zweifelsfall unbefriedigendste Lösung.

2. Die nächste Stufe ist das Minimum, was man von einem System erwarten sollte. Bei jedem Schritt einer solchen Datensicherung wird eine Meldung ausgegeben. Das könnte dann so aussehen:

   ===> Daten erfolgreich geschrieben

   ===> Tabelle `xyz` mit `ii` Datensätzen geschrieben

   oder:

   =X=X=> Fehler !!!

   =X=X=> Schreiben der Tabelle `xyz` nach `jj` Datensätzen abgebrochen

   OK, das sieht doch schon mal nicht schlecht aus. Es erfordert aber, dass der Anwender diese Informationen jeden Tag liest, und das sehr sorgfältig. Aber jeder weiß: jetzt hat das an 500 Tagen funktioniert, dann brauch ich ab jetzt nicht mehr so genau hinschauen. Das wird doch auch weiterhin stimmen.

   Dazu muss aber das Programm, das für die Datensicherung zuständig ist, bei jeder Aktion die korrekte Ausführung überprüfen, und das müsste der Programmierer tatsächlich Schritt für Schritt programmieren.[2] Nach jedem einzelnen der überprüften Schritte muss dann der Programmierer,

[1] Ballad of a thin man *auf:* Highway 61 revisited

[2] So hat man das wirklich vor nicht allzu langer Zeit gemacht. Eine extrem lästige Arbeit!

nach Anweisung der Fachabteilung, in sein Programm einbauen, was passieren soll: soll die gesamte Sicherung abgebrochen werden, soll eine Fehlermeldung ausgegeben werden und die Sicherung der weiteren Dateien versucht werden, müssen bereits gesicherte Daten wieder entfernt werden, damit keine Inkonsistenzen entstehen?

Diese Fehlerbehandlung „zu Fuß“ wird umfangreich und damit auf Dauer undurchsichtig, also auch schwer wartbar. Es ist auch fraglich, ob bei einer solchen Fehlerbehandlung in Handarbeit alle möglichen Fehlerquellen abgedeckt werden und vor allem keine möglicherweise in Zukunft auftretende Fehlerquelle nicht bedacht wird.

3. Um das zu verbessern und zu vereinfachen, ist in der Software-Technik das Prinzip der „Ausnahmebehandlung“ entstanden. Eine Ausnahme ist irgendetwas in einem Programmablauf, was der Entwickler nicht erwartet. Genauer gesagt: er erwartet nicht, dass es passiert, aber er weiß, dass es passieren könnte. Im schlimmsten Fall ist eine Ausnahme ein Fehler, der zu falschen Ergebnissen führt, vielleicht auch zu Datenverlust, in weniger schlimmen Fällen entsteht in einem Programm eine Situation, die eine weitere Ausführung unmöglich macht, aber noch andere Ausnahme(type)n sind denkbar.

Aber meistens soll ein Programm trotz eines Fehlers weiterlaufen, vor allem dann, wenn der Benutzer etwas macht, was der Programmierer so nicht vorgesehen hat. Nehmen wir an, der Programmierer hat ein Programm geschrieben, in dem der Benutzer sein Alter eingeben soll, und Otto (der aus Ostfriesland) sitzt vor dem Programm und gibt ein „ich sage meistens Papa zu ihm“. In der Otto-Show ist das ganz lustig, der Programmierer hatte aber erwartet, dass der Benutzer eine Zahl eingibt, weil mit Hilfe dieser Zahl, dem Alter in Jahren, etwas berechnet werden soll. Das Programm wird im Normalfall abbrechen und eine eher verwirrende Fehlermeldung ausgeben. Hier kommt dieses Beispiel, und was die Python-Shell daraus macht:

Beispiel 12.1.1 Fehler! (Gruß an Otto)

```
>>> alter =  input('Alter? ')
Alter? ich sage meistens Papa zu ihm
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<string>", line 1
ich sage meistens Papa zu ihm
         ^
SyntaxError: invalid syntax
```

Was man sich wünschen würde, wäre aber etwas in der Art:

Beispiel 12.1.2 Ein Fehler wird abgefangen

```
>>> alter =  input('Alter? ')
Alter? ich sage meistens Papa zu ihm
 Fehlerhafte Eingabe! Es ist das Alter in Jahren gefragt. Probier es nochmal!
>>> alter =  input('Alter? ')
```

Und man bekommt eine zweite Chance (und zur Not eine dritte und eine vierte ...)

Zu diesem Zweck hat Python eine eingebaute Fehlerbehandlung, die man aber besser als Ausnahmebehandlung bezeichnet. Das Prinzip ist immer das folgende:

1. Wenn man einen Programmbefehl schreibt, bei dem es möglich ist, dass ein Fehler passiert, teilt man dem Programm mit, dass es diesen Befehl nicht ausführen soll, sondern probieren soll, ob der Befehl ausführbar ist.
2. Wenn ja, dann wird der Befehl auch tatsächlich ausgeführt.

3. Wenn der Befehl nicht ausgeführt werden kann, wird eine Ausnahme geworfen.[3] Die Ausnahme ist ein Befehl, der ausgeführt wird, wenn der ursprünglich gewünschte Befehl nicht ausgeführt werden kann.
4. Es gibt dann noch die Möglichkeit, Befehle anzuschließen, die auf jeden Fall ausgeführt werden, egal, ob der ursprünglich gewünschte Befehl geklappt hat oder nicht.

Das sieht dann in Python so aus:

Beispiel 12.1.3 Try - Except

```
  >>> try:
        alter = input('Alter?')
      except:
        print 'keine Zahl'
...
  Alter?d
  keine Zahl
```

Hier wurde einfach blindlings mit `except` jeder x-beliebige Fehler abgefangen. Und das ist doch schon ganz gut, denn damit kann das Programm weiterlaufen und als Programmierer kann man sich überlegen, was in einem solchen Fall gemacht werden soll.

Dieses einfache Muster einer Fehlerbehandlung

Beispiel 12.1.4 einfache Fehlerbehandlung

```
  >>> try:
        Anweisung(en)
      except:
        Fehlermeldung
```

kann verbessert werden zu

Beispiel 12.1.5 Try - Except

```
  >>> try:
        Anweisung(en)
      except:
        Fehlermeldung
      else:
         weitere Anweisungen, falls keine Ausnahme aufgetreten ist
```

Das `else` ist in manchen Fällen sehr hilfreich. Nur so kann man sicherstellen, dass Anweisungen dann und nur dann ausgeführt werden, wenn keine Ausnahme aufgetreten ist.

Oft weiß man aber genauer, welche Art von Fehler gemacht werden könnte:

Beispiel 12.1.6 Korrekter Index in einer Liste

```
  >>> eineListe = ['nulltes Element', 'erstes Element', 'zweites Element']
  >>> eineListe[1]
  'erstes Element'
```

[3] Diese Sprechweise hat sich auch im Deutschen durchgesetzt: sie ist eine wörtliche Übersetzung von „to throw an exception“

So weit kein Problem. Aber was passiert, wenn man jetzt versucht, das siebte Element auszugeben?

Beispiel 12.1.7 Falscher Index in einer Liste

```
>>> eineListe = ['nulltes Element', 'erstes Element', 'zweites Element']
>>> eineListe[7]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Immerhin sagt Python mir hier, was falsch ist: der Index ist außerhalb des gültigen Bereichs. Es handelt sich also hier um einen Index-Fehler. Das kann man natürlich viel genauer abfangen:

Beispiel 12.1.8 Index-Fehler wird abgefangen

```
>>> try:
      eineListe[7]
    except IndexError:
      print 'Index außerhalb der erlaubten Grenzen'

Index außerhalb der erlaubten Grenzen
```

Hier wurde also der Index-Fehler ausdrücklich abgefangen.

Es ist gute Technik, Fehler immer mit einem ganz konkreten Fehlercode abzufangen. Mit einem einfachen `try ... except` ohne die Angabe eines Fehlercodes ist es vor allem in der Phase der Programmentwicklung ausgesprochen schwierig, einem Fehler auf die Spur zu kommen. Da ist es hilfreich, wenn der Fehler durch den Fehlercode eingegrenzt wird. Dazu kommt jetzt eine Tabelle der wichtigsten Fehlercodes.

Tabelle 12.1. Fehler-Konstanten

Konstante	Bedeutung
EOFError	Leseoperation über das Ende der Datei hinaus
IOError	Fehler bei Ein-/Ausgabe
TypeError	Falscher Typ des Parameters / der Variablen
ZeroDivisionError	Division durch 0
IndexError	Falscher Index (Versuch, über Ende einer Liste zu lesen)

Teil VIII.

Permanente Speicherung

13. Dateizugriff

13.1. Dateien allgemein

Dateien sind Speicherbereiche auf einem Speichermedium, die erstellt, geöffnet, geschlossen, bearbeitet und gelöscht werden können. Dafür ist das jeweilige Betriebssystem zuständig, das für jede dieser Aktionen einen Befehl bereitstellt. [1]

Jede Aktion mit Dateien fängt damit an, dass man die Datei öffnet. Damit erstellt man aus Sicht von Python ein Objekt, das eine Verbindung zu der real existierenden Datei auf dem Datenträger herstellt. Dabei muss man im Programm festlegen, was man nach dem Öffnen der Datei mit der Datei anstellen will: will man sie nur lesen, will man sie verändern, oder will man eine neue Datei erstellen (und damit eventuell eine bereits existierende Datei des selben Namens überschreiben). Der entsprechende Befehl in Python sieht also so aus:

```
meineDatei = open('meinDateiName.txt', 'r') # r steht für Lesen
meineDatei = open('meinDateiName.txt', 'w') # w steht für (Neu-)Schreiben
meineDatei = open('meinDateiName.txt', 'a') # a steht für Anhängen, also
                                            #   verändern
```

Damit ist klar: ab diesem Zeitpunkt ist `meineDatei` ein Objekt der Klasse `file`.

Diese Klasse hat einige Methoden, die jetzt für den Zugriff auf den Inhalt der Datei benutzt werden. Grundsätzlich gilt: was man aufmacht, muss man auch wieder zumachen. Man sollte jede Datei nach Gebrauch wieder schließen, und dazu dient die Methode `close`:

```
meineDatei.close()        # Die Methode close des Objektes wird
                          # aufgerufen
```

Schreiben wir also zwei Programme: zuerst eines, das eine Datei schreibt, dann ein zweites, das diese Datei wieder liest. Für das Schreiben steht die Methode `write` zur Verfügung.

Beispiel 13.1.1 Datei schreiben

```
#!/usr/bin/python
# -*- coding: utf8 -*-
# Schreib-Programm
neueDatei = open('liebeserkl.txt', 'w')
neueDatei.write('Ich liebe Python.\n Und natürlich meine Freundin.')
neueDatei.close()
```

Jetzt kann man auf Betriebssystemebene (oder mit einem Hilfsprogramm wie xxx-Commander) sehen, dass diese Datei erstellt worden ist.

Das Thema „Dateizugriff" wurde mit dem Schreiben begonnen, da das die einfachere Operation ist. Man muss sich dabei aber klar machen, dass der Dateizugriffsmodus „w" bedeutet, dass unbedingt geschrieben wird. Falls eine Datei des selben Namens bereits existiert, wird diese überschrieben. Das ist einfacher, weil es immer funktioniert, allerdings kann man da auch ganz schnell viel zerstören. Das Lesen ist schon allein deswegen komplizierter, weil das Programm nicht weiß, wieviel es zu lesen gibt.

[1] Ganz sicher bin ich mir nicht, ob das wirklich stimmt. In Unix-ähnlichen Betriebssystemen wird eine leere Datei durch den Befehl **touch dateiname** erstellt (JA! Das ist wirklich sinnvoll!!) Gibt es einen analogen Befehl unter Windows?

Erwartet man eine eher kleine Datei, ist es oft sinnvoll, die ganze Datei auf einmal einzulesen und dann das Gelesene in einer Schleife abzuarbeiten. Wenn man allerdings weiß, dass die zu lesende Datei vielleicht mehrere 100 MB groß ist, ist es sinnvoller, diese Datei Stück für Stück (das heißt günstigstenfalls Zeile für Zeile) einzulesen und jede Zeile einzeln zu bearbeiten. Im Vergleich zu der ersten Methode bedeutet das, dass hier die Schleife nicht innerhalb des Gelesenen sondern vor dem Lesen steht (oder wie der Programmierer sagt: man schleift über das Lesen).

Hier wird die erste Methode angewandt, um die Datei zu lesen, nämlich alles auf einmal. Die Python-Methode dazu heißt wieder genauso, wie man das erwartet: `readlines` (beachte den Plural) liest alle Zeilen der Datei.

Beispiel 13.1.2 Datei lesen (alles auf einmal)

```
#!/usr/bin/python
# -*- coding: utf8 -*-
# Lese-Programm
liebeDatei = open('liebeserkl.txt', 'r')
alleZeilen = liebeDatei.readlines()
print alleZeilen
liebeDatei.close()
```

Wenn man die Schleife um das Lesen herummachen will, benutzt man die Methode `readline`. Eigentlich ist das auch wieder logisch, dass man in diesem Fall nur eine Zeile liest, deswegen also der Singular. Dann sieht das Programm so aus:

Beispiel 13.1.3 Datei zeilenweise lesen

```
#!/usr/bin/python
# -*- coding: utf8 -*-

dat = open('liebeserkl.txt', 'r')
ende = False
while not(ende):
  zeile = dat.readline()
  if not(zeile):
    ende = True
  else:
    print zeile
dat.close()
```

Hier muss man gewährleisten, dass nicht versucht wird, über das Ende der Datei rauszulesen. Das würde einen Programmabsturz verursachen, also muss mit dem logischen Schalter `ende` das Dateiende abgefangen werden.

Seit Python Version 2.4 gibt es für den Dateizugriff „Iteratoren“, also Mechanismen, die „mitzählen“, wieviel man gelesen hat und die damit wissen, was als nächstes drankommt. Der Unterschied ist, dass Iteratoren mit Hilfe einer Exception[2] abgefangen werden, während beim `readline` am Ende der Datei einfach eine leere Zeichenkette in der entsprechenden Variablen (im Beispiel oben in der Variablen `zeile`) steht. Der Iterator, der immer auf die nächste Zeile zeigt, heißt, wen wundert es, `next`. Das Programm ändert sich leicht:

[2]Das Thema Fehler und Ausnahmen steht erst weiter hinten im Text.

Beispiel 13.1.4 Datei zeilenweise mit Iterator lesen

```
#!/usr/bin/python
# -*- coding: utf8 -*-

dat = open('liebeserkl.txt', 'r')
ende = False
while not(ende):
  try:
    zeile = dat.next()
    print zeile
  except StopIteration:
    ende = True
    break
dat.close()
```

13.2. Fortführung des Konten-Programms

Die erste Änderung hat noch nichts mit dem Dateizugriff zu tun, denn hier geht es darum, das Programm so zu ändern, dass man mehrere Konten bearbeiten kann. Dazu sollte das Anlegen eines Kontos nicht mehr fest im Programm kodiert sein, sondern interaktiv erfolgen. Also muss man den Konstruktor der Klasse `Konto` ändern.

Wahrscheinlich fällt hier wieder jedem auf, wie elegant der objektorientierte Ansatz ist. Um für alle möglichen Konten, die wir bearbeiten wollen, diese Änderung durchzuführen, muss man tatsächlich nur an einer Stelle eingreifen. Ich schreibe hier die alte Version des Konstruktors nochmals hin:

Beispiel 13.2.1 Der bisherige Konstruktor für Konten

```
1
2   def __init__(self, ktoNr,  vorname,  nachname,  gebdat,  wohnort,
3                  ktoStand,  zinssatz):
4     self.ktoNr = ktoNr
5     self.inhaber = Mensch(vorname,  nachname, gebdat,  wohnort)
6     self.ktoStand = ktoStand
7     self.zinssatz = zinssatz
8
```

und hintendran gleich die neue Version, nach der Änderung, die ich ein paar Zeilen zuvor angesprochen habe:

Beispiel 13.2.2 Neuer Konstruktor: Parameter sind teilweise initialisiert

```
1
2   def __init__(self, ktoNr,  vorname,  nachname,  gebdat,  wohnort,
3                  ktoStand = 0,  zinssatz = 0.02):
4     self.ktoNr = ktoNr
5     self.inhaber = Mensch(vorname,  nachname, gebdat,  wohnort)
6     self.ktoStand = ktoStand
7     self.zinssatz = zinssatz
8
```

Was hier passiert, wurde schon im Abschnitt über variable Parameter beim Funktionsaufruf beschrieben.

Im Aufruf-Programm muss sich allerdings viel ändern. Als erstes bekommt das Programm zwei Funktionen, mit denen man dem Benutzer die möglichen Eingaben, je nachdem in welchem Programm-Teil der Benutzer sich gerade befindet, auf den Bildschirm schreibt. Ein bißchen Hilfe muss sein.

Beispiel 13.2.3 Ein paar kleine Hilfen für die Benutzer

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4   from clKonto import Konto
5
6   alleKonten = {}
7
8   def meldModus():
9     print "Konto-Verwaltung\ne für Einzahlung\na für Auszahlung\ns für Schluss"
10
11  def meldAnl():
12    print "Konto-Verwaltung\nn für 'Neues Konto anlegen'\ns für Schluss"
13
```

Außerdem wird ein leeres Dictionary `alleKonten` angelegt.

Danach startet man eine Schleife, um einige Konten anzulegen:

Beispiel 13.2.4 Eine Schleife für das Anlegen eines Kontos

```
1
2   eingabe = 'q'
3   while (eingabe <> 's'):
4     meldAnl()
5     eingabe = raw_input('Bitte Eingabe:')
6     if eingabe == 'n':
7       neuKtoNr = input('Bitte neue Konto-Nummer eingeben: ')
8       neuVorname = raw_input('Bitte Vornamen eingeben: ')
9       neuNachname = raw_input('Bitte Nachnamen eingeben: ')
10      neuGebdat = raw_input('Bitte Geburtsdatum eingeben: ')
11      neuWohnort = raw_input('Bitte Wohnort eingeben: ')
12      aktuellesKonto = Konto(neuKtoNr,  neuVorname,  neuNachname,
13                                          neuGebdat,  neuWohnort)
14      alleKonten[neuKtoNr] = aktuellesKonto
15    elif eingabe == 's':
16      pass
17    else:
18      print "Falsche Eingabe. nur n und s erlaubt"
19
```

In den Zeilen mit `input` bzw. `raw_input` werden die Parameter für jeweils ein neues Konto entgegengenommen. Mit diesen Werten wird der Konstruktor für ein Konto aufgerufen und damit ein Objekt erzeugt, das eindeutig zu identifizieren ist durch die Kontonummer. Diese Kontonummer als Schlüssel und das Konto-Objekt als Wert werden in dem Dictionary abgelegt.

Jetzt folgen zwei ineinanderverschachtelte while-Schleifen. In der äußeren Schleife wählt man bis zur Abbruchbedingung immer ein neues Konto aus. Die Abbruchbedingung ist hier willkürlich auf die Kontonummer 9999 gesetzt. In der inneren Schleife befindet sich die Bearbeitung eines einzelnen Kontos mit einzahlen, auszahlen und verzinsen, so wie sie bisher schon existierte:

Beispiel 13.2.5 Interaktive Auswahl eines Kontos aus einer Liste

```
aktuelleKontoNr = 0
while (aktuelleKontoNr <> 9999):
  print "angelegte Konten: ",  alleKonten.keys()
  print "Konto auswählen: (ENDE mit 9999)"
  aktuelleKontoNr = input('Kontonummer eingeben: ')
  if aktuelleKontoNr <> 9999:
    aktuellesKonto = alleKonten[aktuelleKontoNr]

    eingabe = 'q'
    while (eingabe <> 's'):
      meldModus()
      eingabe = raw_input('Bitte Eingabe:')
      if eingabe == 'e':
        betrag = input('Einzahlungsbetrag: ')
        aktuellesKonto.einzahlen(betrag)
      elif eingabe == 'a':
        betrag = input('Auszahlungsbetrag: ')
        aktuellesKonto.auszahlen(betrag)
      elif eingabe == 's':
        print 'und tschüß'
      else:
        print "Falsche Eingabe. nur a, e, s erlaubt"
      aktuellesKonto.kontoStandAusgeben()
  else:
    pass
```

13.3. In die Datei damit!!!

Fast schämt man sich, dafür ein neues Kapitel anzufangen. Das Schreiben von Objekten in eine Datei, im nächsten Programm auch das Lesen von Objekten aus einer Datei, wird von einem Modul erledigt, der auf den schönen Namen `pickle` hört. Zuerst muss natürlich das Modul pickle importiert werden. Der Anfang des Aufruf-Programms ändert sich dadurch zu:

Beispiel 13.3.1 Modul importieren: Ein Objekt in eine Datei schreiben

```
#!/usr/bin/python
# -*- coding: utf8 -*-
from clKonto import Konto
import pickle
```

Ganz an das Ende des Programms wird dann die Dateibehandlung eingefügt. Sie besteht aus 4 Schritten: Dateinamen vergeben, Datei öffnen, Objekt in die Datei dumpen, Datei schließen.

Beispiel 13.3.2 Ein Objekt in eine Datei schreiben

```
dateiName = 'alleKonten.pydat'
ktoDatei = open(dateiName, 'a')
pickle.dump(alleKonten,  ktoDatei)
ktoDatei.close()
```

Damit das ganze Kapitel aber nicht total peinlich ist, folgt noch ein Aufruf-Programm, mit dem man die geschriebene Datei wieder auslesen kann. Auf die Verarbeitung der gelesenen Daten wird hier verzichtet, da man das aus dem vorigen Aufruf-Programm kopieren kann.

Beispiel 13.3.3 Lesen eines Objektes aus einer Datei

```
#!/usr/bin/python
# -*- coding: utf8 -*-
from clKonto import Konto
import pickle

alleKonten = {}

dateiName = 'alleKonten.pydat'
ktoDatei = open(dateiName, 'r')
while ktoDatei:
  try:
    alleKonten = pickle.load(ktoDatei)
  except EOFError:
    break
ktoDatei.close()

print "angelegte Konten: ",  alleKonten.keys()
```

WICHTIG

Unter Python 3.x ist der Dateizugriff mittels `pickle` nicht mehr mit den Attributen r und w, sondern mit rb und wb (read binary / write binary).

Ein leeres Dictionary wird angelegt, dann der Dateiname an eine Variable zugewiesen, die Datei wird geöffnet, und dann wird die Datei, die nur ein einziges Objekt enthält, in das Dictionary eingelesen. Der Lese-Prozess wird durch ein try - except - Konstrukt abgesichert. Danach wird die Datei geschlossen und nur zur Kontrolle werden die gelesenen Kontonummern ausgegeben.

13.4. Übungen zu Dateien

1. Das Eisdielen-Programm soll weiter verändert werden, so dass eine Rechnung zu einer Bestellung in eine Datei geschrieben wird. Die Rechnungen sollen eine eindeutige Rechnungsnummer bekommen (hochzählen!), und der Dateiname soll jeweils `RNr234.txt` heißen (wobei 234 hier für die Rechnungsnummer steht.

14. Datenbanken

14.1. MySQL

Python hat Module für den Datenbankzugriff für viele verschiedene relationale Datenbanken. MySQL ist eine Datenbank, die unter die freie Software fällt, und MySQL ist weit verbreitet, weil es eine sehr mächtige Datenbank ist. Für MySQL gibt es auch die Möglichkeit, eine Lizenz zu erwerben und sich damit Support zu sichern. Das führte dazu, dass MySQL nicht nur im „Hobby"-Bereich verbreitet ist, sondern auch große Unternehmen mit dieser Datenbank arbeiten.

Für die Arbeit mit Datenbanken gilt vieles, was auch für die Arbeit mit Dateien (siehe dort) gilt. Ein Datenbankzugriff besteht immer aus drei Schritten:

1. Verbindung zur Datenbank herstellen
2. Aktion auf der Datenbank ausführen
3. Verbindung zur Datenbank lösen

Das Herstellen der Verbindung besteht wiederum aus zwei Teilen:

1. Eine Verbindung wird eingerichtet
2. Ein Cursor wird auf diese Verbindung gesetzt

Das Schreiben in eine Datenbank sieht im schlichtesten Fall folglich so aus:

Beispiel 14.1.1 Schreiben in eine Datenbank

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   import MySQLdb
6   verbindung = MySQLdb.connect("localhost", db="nameort")
7   cursor = verbindung.cursor()
8
9   cursor.execute("INSERT INTO namen (vorname, name, plz)
10                          VALUES (%s, %s, %s)", ('Ernst', 'Maier', '72108'))
11
12  cursor.close()
13  verbindung.close()
14
```

Das Modul für die Datenbank, hier für MySQL, muss import werden. Dieses Modul enthält eine Methode `connect`, die als erstes aufgerufen wird. Auf die erstellte Verbindung wird der Cursor aufgesetzt, sozusagen ein Zeiger auf die Datenbank, an den eventuelle Befehle übergeben werden. Im nächsten Schritt wird für diesen Cursor die Methode `execute` aufgerufen, die den als Parameter übergebenen SQL-Befehl ausführt. Danach wird zuerst der Cursor, dann die Verbindung geschlossen.

Das alles sind durchaus kritische Operationen. Aber der Umgang mit kritischen Operationen wurde ja im vorigen Kapitel angerissen: Dinge, von denen man nicht weiß, ob sie wirklich so klappen, wie man sich das als Entwickler gedacht hat, versucht man erst und fängt eine mögliche Ausnahme ab. Eine bessere Version sieht so aus:

Beispiel 14.1.2 Schreiben in eine Datenbank mit Fehlerbehandlung

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   import MySQLdb
6   try:
7     verbindung = MySQLdb.connect("localhost", db="nameort")
8   except:
9     print 'Verbindung konnte nicht hergestellt werden'
10  else:
11    try:
12      cursor = verbindung.cursor()
13    except:
14      print 'Cursor konnte nicht gesetzt werden'
15    else:
16      cursor.execute("INSERT INTO namen (vorname, name, plz)
17                           VALUES (%s, %s, %s)", ('Ernst', 'Maier', '72108'))
18
19      cursor.close()
20      verbindung.close()
21
```

Man sieht: das ist eine verschachtelte `try - except` Anweisung, da ja bei jedem Schritt eine Ausnahme auftreten kann.

Das ist aber nicht wirklich schön, diese geschachtelte `try - except` Anweisung. Hier ist es besser, mehrere Anweisungen zu versuchen und für jede mögliche Ausnahme eine spezielle Fehlerklasse abzufragen. Das Programm ändert sich also noch einmal:

Beispiel 14.1.3 Schreiben in eine Datenbank mit differenzierter Fehlerbehandlung

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   import MySQLdb
6   try:
7     verbindung = MySQLdb.connect("localhost", db="plzort")
8     cursor = verbindung.cursor()
9   except MySQLdb.DatabaseError:
10    print 'Verbindung konnte nicht hergestellt werden'
11  except:
12    print 'Cursor konnte nicht gesetzt werden'
13  else:
14    try:
15      cursor.execute("INSERT INTO plzort (plz, ort)
16                      VALUES (%s, %s)", ('72070', 'Tübingen'))
17    except:
18      print 'Insert fehlgeschlagen'
19    else:
20      print 'ok'
21      verbindung.commit()
22      cursor.close()
23      verbindung.close()
24
25
```

Lesen aus einer Datenbank:

Beispiel 14.1.4 Lesen aus einer Datenbank

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   import MySQLdb
6   connection = MySQLdb.connect("localhost", db="nameort")
7   cursor = connection.cursor()
8   t2 = cursor.execute("SELECT * FROM namen")
9   for i in range(t2):
10    print cursor.fetchone()
11
12  cursor.close()
13  connection.close()
14
```

Teil IX.

Noch ein paar Beispiele zur OOP

15. Ein weiteres Projekt: ein Getränkeautomat

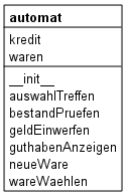
15.1. Objektorientierter Entwurf und Realisierung der Klasse

Ein Getränkeautomat nimmt Geld entgegen und spuckt ein Getränk aus, sofern das Geld reicht und die Ware vorhanden ist. Falls zuviel Geld eingeworfen wurde, wird das restliche Geld auch ausgegeben.

Diese Beschreibung gilt nicht nur für einen Getränkeautomaten, sondern allgemein für einen Automaten: der nimmt Geld entgegen, überprüft den Warenbestand, gibt Waren aus und berechnet Restgeld und gibt das zurück. Deswegen wird die Klasse gleich als allgemeiner Automat codiert. Wiederverwertbarkeit ist ja eines der Ziele der objektorientierten Programmierung.

Die einfache Klasse sieht folgendermaßen aus:

Abbildung 15.1. Klassendiagramm `Automat`



Die einzelnen Methodennamen sprechen hoffentlich für sich. Trotzdem:

- **bestandPruefen:** ist noch genügend des gewünschten Getränks im Automaten?
- **geldEinwerfen:** Klar! Hier soll aber geprüft werden, ob es eine zulässige Münze ist
- **neueWareAufnehmen:** ein neues Getränk wird eingespeist
- **wareWaehlen:** Welches Schweinderl hättn's denn gern?

Beispiel 15.1.1 Klassenentwurf des Getränkeautomaten

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   class Automat():
6     def __init__(self, waren=[], kredit = 0):
7       self.waren = waren
8       self.kredit = kredit
9
10    def geldEinwerfen(self):
11      print 'Geld eingeworfen'
12
13    def guthabenAnzeigen(self):
14      print 'Guthaben anzeigen'
15
16    def wareWaehlen(self):
17      print 'Ware auswählen'
18
19    def neueWareAufnehmen(self):
20      print 'neue Ware hinzugenommen'
21
22    def bestandPruefen(self):
23      print 'Bestand der gewählten Ware geprüft'
24
```

Dieser Klassenentwurf wird in einer Datei `cl_automat.py` gespeichert.

Das interessanteste und vielleicht wichtigste Attribut ist das Attribut `waren`. Das ist eine Liste, in die in einem der späteren Entwicklungsschritte eine ganze Menge von Waren „eingelagert“ werden. Da zu Beginn von einem leeren Automaten ausgegangen wird, wird an den Konstruktor eine leere Liste übergeben. Außerdem sind die einzelnen Methoden noch nicht ausformuliert, sondern geben einfach einen Text aus.

Zu der Klasse `Automat` gehört jetzt eigentlich ein aufrufendes Programm. Dieses sollte die einzelnen Methoden testen, die bisher nur als „Dummies“ existieren. Die einfache Lösung wäre ein sequentielles Programm, das eine Instanz des Automaten erzeugt und dann nacheinander die einzelnen Methoden aufruft.

Aber das würde den Regeln der Objektorientiertheit widersprechen. Denn das, was im vorigen Abschnitt beschrieben wurde, ist eigentlich wieder ein Objekt, nämlich ein Objekt einer noch zu schaffenden Klasse `Menu`. Diese Klasse benötigt nur eine einzige Methode, die Methode `auswahlTreffen`

- **auswahlTreffen:** das Menu für den Getränkeautomaten: neue Waren anlegen, Geld einkassieren, etc.

Also dann: los gehts!

Beispiel 15.1.2 Klassenentwurf des Menus

```
1
2  #!/usr/bin/python
3  # -*- coding: utf8 -*-
4
5  from cl_automat import Automat
6
7  class Menu():
8
9    def __init__(self):
10     self.automat = Automat()
11     self.prompt = 'g = Geld einwerfen,
12                           w = Ware wählen, n = neue Ware einführen,
13                           e = Ende\n'
14
15   def auswahlTreffen(self):
16     auswahl = ['e', 'w',  'g',  'n']
17     eingabe = 'x'
18     while eingabe <>'e':
19       eingabe = raw_input(self.prompt)
20       if eingabe == 'g':
21         self.automat.geldEinwerfen()
22       elif eingabe == 'w':
23         self.automat.wareWaehlen()
24       elif eingabe == 'n':
25         self.automat.neueWare()
26       elif eingabe == 'e':
27         pass
28       else:
29         print 'fehlerhafte Eingabe; nur e, g, n, w zulässig'
30
```

Damit das Menu weiß, dass es sich um ein Menu für einen Automaten handelt, muss zuerst die Klassenbeschreibung aus der Datei `cl_automat` importiert werden; danach wird im Konstruktor des Menus ein Objekt der Klasse `Automat` erzeugt. Die Methode `auswahlTreffen` ruft dann die Methoden des Automaten auf.

Wie immer gehört dazu auch ein aufrufendes Programm, das einfach ein Objekt der Klasse anlegt und dann die Haupt-Methode der Klasse, nämlich das Auswahlmenu aufruft.

Beispiel 15.1.3 Das aufrufende Programm für den Getränkeautomaten

```
#!/usr/bin/python
# -*- coding: utf8 -*-

from cl_menu import Menu

getraenkeautomat = Menu()
getraenkeautomat.auswahlTreffen()
```

Das ist doch richtig schön, ein solch kurzes Programm. Die ganze Intelligenz steckt in der Klasse. Und jedes Objekt der Klasse macht alles genau so gut wie jedes andere!!

Kommen wir also zur zweiten Version, in der ein bißchen etwas passiert.

15.2. Die Waren kommen in den Automaten

... und damit die Waren in den Automaten kommen, müssen wir natürlich zuerst eine Klasse für die Waren deklarieren.

Abbildung 15.2. Klassendiagramm `Ware`

Diese Klasse hat vorläufig eine einzige Methode, nämlich den Konstruktor.

Interaktiv soll jetzt eine Ware angelegt werden, wobei die drei Attribute des Objektes gefüllt werden. Gleichzeitig wird die Methode `warenAnzeigen` geschrieben, die jedes Mal nach Anlegen eines neuen Objektes aufgerufen wird. Im folgenden werden nur diese beiden Methoden angezeigt, der Rest der Klassendefinition bleibt so wie in der vorigen Version.

Beispiel 15.2.1 Die Waren kommen hinzu

```
1
2  class Ware():
3    def __init__(self, bezeichnung,  verkPreis,  bestand):
4      self.bezeichnung = bezeichnung
5      self.verkPreis = verkPreis
6      self.bestand = bestand
7
8  class Automat():
9      ...
10     ...
11   def warenAnzeigen(self):
12     print 'folgende Waren können gekauft werden: '
13     print  '%20s %10s %10s' % ('Ware',  'Preis',  'Bestand')
14     for ware  in self.waren:
15       print '%20s %10.2f %10i' % (ware.bezeichnung, ware.verkPreis, ware.bestand)
16
17   def neueWareAufnehmen(self):
18     bez = raw_input('Bezeichnung der neuen Ware: ')
19     vkPr = input('Verkaufspreis der neuen Ware: ')
20     best = input('Lagerbestand der neuen Ware: ')
21     nW = ware(bez, vkPr,  best)
22     self.waren.append(nW)
23     print 'neue Ware hinzugenommen'
24     self.warenAnzeigen()
25
```

Das aufrufende Programm ändert sich überhaupt nicht! (Ist das ein gutes Zeichen??? Schon, oder?)

15.3. Geld regiert die Welt

Jetzt soll das Geld behandelt werden. Das bedeutet, dass die Methode des Geldeinwerfens geschrieben werden muss, und es ist sinnvoll, die Methode, die den aktuell eingeworfenen Betrag anzeigt, gleich mitzubehandeln. Das UML-Diagramm der Klasse ändert sich nicht, nur die beiden angesprochenen Methoden werden ausformuliert.

Beispiel 15.3.1 Geld einwerfen und Guthaben anzeigen

```
1
2   def geldEinwerfen(self):
3     muenzen = [1, 2, 5, 10, 20, 50,  100,  200]
4     einwurf = input('Münze einwerfen: ')
5     if einwurf in muenzen:
6       self.kredit += einwurf
7     else:
8       print 'ungültige Münze, gültig sind: ', muenzen
9     self.guthabenAnzeigen()
10
11  def guthabenAnzeigen(self):
12    print 'aktuelles Guthaben: %5.2f \euro\/ ' % (self.kredit/100.0)
13
```

Das ist wirklich sehr einfach. Es wird eine Liste der zulässigen Münzen erstellt (dabei muss nur beachtet werden, dass 100 Cent gleich 1 € ist). Falls man eine korrekte Münze eingibt, wird das zum aktuellen Guthaben — hier Kredit genannt — dazuaddiert.

Das aufrufende Programm ändert sich überhaupt nicht! (Ist das ein gutes Zeichen??? Aber sicher!!) Ich weiß, ich wiederhole mich . . .

15.4. Jetzt kann gekauft werden!

Es fehlt nur noch die Realisierung der Methoden, die für den tatsächlichen Kauf zuständig sind. Dazu gehört, dass geprüft wird, ob der Bestand ausreichend ist und ob das Guthaben groß genug ist. Falls ja, muss Restgeld herausgegeben werden und das Guthaben zurückgesetzt werden. Außerdem muss in diesem Fall der Bestand heruntergezählt werden.

Beispiel 15.4.1 Der Kauf

```
1
2   def wareWaehlen(self):
3     print 'Ware auswählen'
4     self.warenAnzeigen()
5     ausgewaehlt = input('Nr. der gewünschten Ware eingeben: ')
6     if not self.bestandPruefen(ausgewaehlt):
7       print 'zu wenig Ware auf Lager'
8     elif self.kredit <= self.waren[ausgewaehlt].verkPreis:
9       print 'Du hast zuwenig Geld eingeworfen'
10    else:
11      print 'Ware wird ausgegeben'
12      self.waren[ausgewaehlt].bestand -= 1
13      print 'Restgeld %4.2f \euro\/  wird ausgegeben' %
14                  (self.kredit - self.waren[ausgewaehlt].verkPreis)
15      self.kredit = 0.0
16
17  def bestandPruefen(self, nr):
18    if self.waren[nr].bestand <= 1:
19      return False
20    else:
21      return True
22
```

Hier gehört nur noch ein blöder Spruch hin: Das aufrufende Programm ändert sich überhaupt nicht! (Ist das ein gutes Zeichen??? Super!!!!)

15.5. To Do!!

Vorschläge, was zu diesem Programm noch hinzugefügt werden kann:

- Fehler müssen abgefangen werden (z.B. wenn man eine Produktnummer außerhalb des gültigen Bereichs wählt)
- Die Münzen können erweitert werden, so dass auch Geldscheine angenommen werden.
- Eine Stückelung des Restgeldes kann realisiert werden, wobei zu beachten ist, dass nicht die optimale Stückelung herausgegeben wird, sondern unter Umständen auch viel Kleingeld, weil andere Münzen nicht mehr vorhanden sind. Aha: ein Geldbestand des Automaten muss also hinzugefügt werden.
- Es kann ermöglicht werden, dass man mehr Geld als für eine Ware einwirft und dann auch mehrere Waren kauft.

16. Noch ein Beispiel: Zimmerbuchung in einem Hotel

16.1. Vorstellung des Projekts

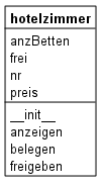
Natürlich soll hier nicht eine vollständige Software für das Management eines Hotels geschrieben werden. Das Modell wird sehr vereinfacht, und in verschiedenen Versionen dann verfeinert.

Wir gehen von einem Hotelzimmer aus. Dieses wird als Klasse modelliert, wobei diese Klasse bewusst einfach gehalten. wird

16.2. Der erste Entwurf: Eine Klasse für Hotelzimmer

Ein ordentliches Hotelzimmer hat eine Zimmernummer, man sollte wissen, wieviele Betten in dem Zimmer sind, ob das Zimmer aktuell frei ist und wieviel eine Übernachtung in diesem Zimmer kostet. Man sollte die Attribute des Zimmers, so wie sie im vorigen Satz beschrieben worden sind, anzeigen können und in dieser ersten Version den Schalter für die Belegung (frei / belegt) umlegen können. Das UML-Bild dazu sieht folgendermaßen aus (beachte dabei, dass die Attribute und Methoden alphabetisch sortiert sind).

Abbildung 16.1. Klassendigramm `Hotelzimmer`



Die Realisierung in Python folgt:

Beispiel 16.2.1 Eine erste Klasse

```
#!/usr/bin/python
# -*- coding: utf8 -*-

class Hotelzimmer():
  def __init__(self, nr,  anzBetten,  preis = 30,  frei = True):
    self.nr = nr
    self.anzBetten = anzBetten
    self.frei = frei
    self.preis = preis

  def belegen(self):
    if self.frei:
      self.frei = False
    else:
      print 'Zimmer ist bereits belegt'

  def freigeben(self):
    if self.frei:
      print 'Zimmer ist bereits frei'
    else:
      self.frei = True

  def anzeigen(self):
    print 'Nr.:',  self.nr,  'Anzahl Betten: ',  self.anzBetten,
    if self.frei:
      print 'frei'
    else:
      print 'belegt'

if __name__ == '__main__':
  zi1 = Hotelzimmer(1, 2)
  zi1.anzeigen()
  zi1.belegen()
  zi1.anzeigen()
  zi1.belegen()
  zi1.anzeigen()
  zi1.freigeben()
  zi1.anzeigen()
```

Der Konstruktor bekommt als Parameter die oben genannten Attribute, wobei den Attributen `preis` und `frei` ein Defaultwert mitgegeben wird. Die Methoden `belegen` und `freigeben` fragen den aktuellen Stand ab und führen die gewünschte Aktion durch, wenn sie möglich ist. Beachte hierbei, dass das Attribut `frei` mit den Wahrheitswerten `True` und `False` arbeitet.

ANMERKUNG

Da dieser Klassenentwurf noch sehr einfach ist, wird hier kein externes aufrufendes Programm geschrieben, sondern über die Abfrage `if __name__ == '__main__'`: die Klasse aus sich heraus gestartet. Es wird mit dieser Zeile abgefragt, ob dieses Modul als Hauptprogramm oder als eine importierte Datei gestartet wird. Da das jetzt das Hauptprogramm ist, werden die folgenden Zeilen ausgeführt

16.3. Der zweite Entwurf: ein aufrufendes Programm

You could have done better
But I don't mind

(Bob Dylan [1])

Die Klasse `Hotelzimmer` wird hier nicht verändert, sondern es wird ein aufrufendes Programm hinzugefügt. Dieses bietet in einer Endlos-Schleife ein Menu an und verarbeitet die Auswahl.

Beispiel 16.3.1 Menu fürs Hotel

```
#!/usr/bin/python
# -*- coding: utf8 -*-
from clZimmer import Hotelzimmer

def menu():
  auswahl = raw_input('\n#### AUSWAHL ####\nZimmer anzeigen: Z
                       \nZimmer belegen: B\nZimmer freigeben: F
                       \nEnde: E\n#### ..... ####
                       \b\b\b\b\b\b\b\b\b\b\b\b')
  menuListe = {'Z': zimmer1.anzeigen,  'B': zimmer1.belegen,
               'F': zimmer1.freigeben,  'E': beenden}
  menuListe[auswahl]()

def beenden():
  global weiter
  print 'danke für Deinen Besuch'
  weiter  =  0


weiter = 1
zimmer1 = Hotelzimmer(1, 2)
while weiter:
  menu()
```

16.4. Der dritte Entwurf: das Hotel

Dieser dritte Entwurf geht wesentlich weiter: wenn man viele freie Zimmer hat, kann man ein Hotel aufmachen. Oder in der Sprache Python: ich erstelle eine neue Klasse `Hotel`, die als einziges Attribut eine (zu Beginn leere) Liste von Zimmern hat. (Fast einziges Attribut: es gibt noch ein Attribut `weiter`,

[1] Don't think twice, it's alright *auf:* The Freewheeling Bob Dylan

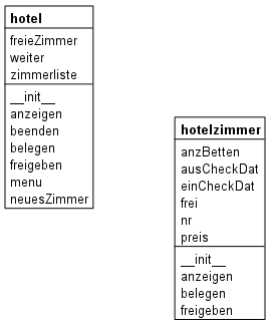
das die Werte `True` oder `False` annehmen kann. Das Hotel hat ferner die Methode `menu`, die vom Konstruktor aufgerufen wird und eine Endlos-Schleife enthält, die die anderen Methoden aufruft.

Diese anderen Methoden sind:

- **anzeigen:** ruft (falls es überhaupt schon Zimmer gibt!!) in einer Schleife die Methode anzeigen der Klasse `Hotelzimmer` auf.
- **beenden:** beendet die Endlos-Schleife dadurch, dass der Schalter `weiter` umgelegt wird.
- **neuesZimmerAnlegen:** erfragt vom Benutzer die benötigten Werte und ruft dann den Konstruktor der Klasse `Hotelzimmer` auf.
- **belegen:** ist noch nicht realisiert, sondern nur als Muster angelegt.
- **freigeben:** ist noch nicht realisiert, sondern nur als Muster angelegt.

Das zugehörige UML-Diagramm sieht so aus:

Abbildung 16.2. Klassendiagramm `Hotel`



Beispiel 16.4.1 Entwurf der Klasse `Hotel`

```
class Hotel():
  def __init__(self):
    self.zimmerliste = []
    self.weiter = True
    while self.weiter:
      self.menu()

  def menu(self):
       auswahl = raw_input('\n#### AUSWAHL ####\nalle Zimmer anzeigen: Z
                        \nZimmer belegen: B\nZimmer freigeben: F
                        \nneues Zimmer anlegen: N
                        \nEnde: E\n#### ..... ####
                        \b\b\b\b\b\b\b\b\b\b\b\b')
    menuListe = {'Z': self.anzeigen, 'B': self.belegen,
                 'F': self.freigeben, 'N':self.neuesZimmer, 'E': beenden}
    menuListe[auswahl]()

  def beenden(self):
    print 'danke für Deinen Besuch'
    self.weiter  =  False

  def anzeigen(self):
    if len(self.zimmerliste) == 0:
      print 'Es gibt noch keine Zimmer'
    else:
      for zimmer in self.zimmerliste:
        zimmer.anzeigen()

  def belegen(self):
    print 'Funktion noch nicht verfügbar'

  def freigeben(self):
    print 'Funktion noch nicht verfügbar'

  def neuesZimmerAnlegen(self):
    anzBetten = input('Anzahl Betten des Zimmers: ')
    preis = input('Preis des Zimmers: ')
    neuesZi = Hotelzimmer(len(self.zimmerliste)+1, anzBetten,  preis)
    self.zimmerliste.append(neuesZi)
```

Hinzu kommt ein aufrufendes Programm. Nicht erschrecken: völlig undurchschaubar !!!!!

Beispiel 16.4.2 Aufrufendes Programm für das Hotel

```
#!/usr/bin/python
# -*- coding: utf8 -*-
from clZiHo import Hotel

meinHotel = Hotel()
```

16.5. Der vierte Entwurf: Methoden werden ergänzt

Die Methode `anzeigen` der Klasse `Hotel` bekommt den zusätzlichen Parameter `frei`; sofern dieser den Wert True hat, werden nur die freien Zimmer angezeigt.

Die Methoden `belegen` und `freigeben` rufen nach Auswahl des Zimmers durch den Benutzer die entsprechende Methode der Klasse `Hotelzimmer` auf.

Vorsicht! Da es kein Zimmer mit der Nummer "0"gibt, Python aber ab 0 zählt, muss hier der eingegebene (weil angezeigte) Index um eins vermindert werden.

Beispiel 16.5.1 Nächster Entwurf für die Klasse `Hotel`

```
class Hotel():
  def __init__(self):
    self.zimmerliste = []
    self.weiter = True
    while self.weiter:
      self.menu()

  def menu(self):
    auswahl = raw_input('\n#### AUSWAHL ####\nalle Zimmer anzeigen: Z
                        \nZimmer belegen: B\nZimmer freigeben: F
                        \nneues Zimmer anlegen: N
                        \nEnde: E\n#### ..... ####
                        \b\b\b\b\b\b\b\b\b\b\b\b')
    menuListe = {'Z': self.anzeigen, 'B': self.belegen,
                 'F': self.freigeben, 'N':self.neuesZimmer, 'E': beenden}
    menuListe[auswahl]()


  def beenden(self):
    print 'danke für Deinen Besuch'
    self.weiter  =  False

  def anzeigen(self,  frei = False):
    if len(self.zimmerliste) == 0:
      print 'Es gibt noch keine Zimmer in diesem Hotel'
    else:
      if frei:
        for zimmer in self.zimmerliste:
          if zimmer.frei:
            zimmer.anzeigen()
      else:
        for zimmer in self.zimmerliste:
          zimmer.anzeigen()

  def belegen(self):
    print 'folgende Zimmer sind verfügbar: '
    # erster Parameter = True:  freie Zimmer anzeigen
    self.anzeigen(True)
    auswahl =  input('gewünschte Zimmer-Nr. angeben: ')
    self.zimmerliste[auswahl - 1].belegen()

  def freigeben(self):
    self.anzeigen(False)
    auswahl =  input('freizugebendes Zimmer: ')
    self.zimmerliste[auswahl - 1].freigeben()

  def neuesZimmerAnlegen(self):
    anzBetten = input('Anzahl Betten des Zimmers: ')
    preis = input('Preis des Zimmers: ')
    neuesZi = Hotelzimmer(len(self.zimmerliste)+1, anzBetten,  preis)
    self.zimmerliste.append(neuesZi)
```

17. Und noch ein Beispiel: ein Adressbuch

17.1. Vorstellung des Projekts

Auch dieses Beispiel soll Schritt für Schritt entwickelt werden, wobei in diesem Fall zuerst das Adressbuch (der Name stimmt nicht, denn unter dem Namen einer Person werden nicht seine Adressdaten gespeichert, sondern sein Alter. Einfach weniger zu schreiben, Faulheit regiert die Welt.) **nicht-objektorientiert** realisiert wird. Auch das wird zeigen, dass es in Python einfach und vor allem durchschaubar ist, ein solches kleines Projekt durchzuführen. Vor allem sind die Veränderungen von einem Schritt zum nächsten oft offensichtlich und gut nachzuvollziehen.

Es sollen in diesem Adressbuch nur drei Daten einer Person gespeichert werden, der Vorname, der Nachname und das Alter. Im Adressbuch sollen diese drei Daten geändert werden können, es sollen natürlich neue Datensätze hinzugefügt werden können und ebenso soll man Datensätze löschen können.

17.2. Der erste Entwurf: eine Liste von Listen

Der erste Versuch speichert die Daten einer Person in einer Liste. Das sieht also so aus:

Beispiel 17.2.1 Adresse in einer Liste

```
#!/usr/bin/python
# -*- coding: utf8 -*-

martin = ['Martin', 'Schimmels',54]
ekki = ['Eckard', 'Krauss',41]
```

Die Personen-Daten von mehreren Personen speichern wir in einer Liste von Listen, also:

```
namen = [martin,ekki]
```

Auf diese Informationen kann man jetzt über die üblichen Listen-Operationen zugreifen (daran denken: man fängt bei 0 an zu zählen!), im folgenden Programm-Listing in der ersten for-Schleife auf die Elemente der äußeren Liste, in der zweiten for-Schleife auf die Elemente der Elemente der äußeren Liste, also auf die Elemente der jeweils inneren Listen.

Beispiel 17.2.2 Zugriff auf die Adress-Liste

```
#!/usr/bin/python
# -*- coding: utf8 -*-

martin = ['Martin', 'Schimmels',54]
ekki = ['Eckard', 'Krauss',41]
namen = [martin,ekki]

for einName in namen:
  print einName

for einName in namen:
  print einName[0],einName[1]
```

Die Ausgabe sieht so aus:

```
>>> ['Martin', 'Schimmels', 54]
>>> ['Eckard', 'Krauss', 41]
>>> Martin Schimmels
>>> Eckard Krauss
```

17.3. Der zweite Entwurf: ein fauler Trick

Hier soll nur eine weiter oben bei [Anwendung von Tupeln] bereits angesprochene Vereinfachung eingebaut werden. Vielen wird das wie ein fauler Trick vorkommen, aber es ist etwas sehr Python-spezifisches. Da die Daten in einem Datensatz immer an der selben Stelle auftauchen, Vorname immer an 0. Stelle, Alter immer an 2. Stelle, wird hier eine Tupel-Zuweisung gemacht, so dass man die Elemente eines Datensatzes mit ihren Bedeutungen ansprechen kann:

Beispiel 17.3.1 Adress-Liste (mit faulem Trick)

```
#!/usr/bin/python
# -*- coding: utf8 -*-

martin = ['Martin', 'Schimmels',54]
ekki = ['Eckard', 'Krauss',41]
namen = [martin,ekki]
#### Trick: Dem Tupel vorname, nachname, alter wird das Tupel 0,1,2 zugewiesen.
#### Damit kann man
#### das 0.te Element mit dem "Index" "vorname" ansprechen
vorname,  nachname,  alter = 0, 1, 2

for einName in namen:
  print einName

for einName in namen:
  print einName[vorname],einName[nachname]
```

17.4. Der dritte Entwurf: eine Liste von Dictionaries

Was mit dem vorigen Trick bereits angedeutet wurde, wird hier mit typischen Python-Mitteln realisiert: die Daten jeder einzelnen Person werden in einem Dictionary gespeichert, die Dictionaries kommen in eine Liste.

Beispiel 17.4.1 Adress-Liste als Dictionary

```
#!/usr/bin/python
# -*- coding: utf8 -*-

martin = {'vorname':'Martin', 'nachname':'Schimmels','alter':54}
ekki = {'vorname':'Eckard', 'nachname':'Krauss','alter':41}
namen = [martin,ekki]

for einName in namen:
  print einName

for einName in namen:
  print einName['vorname'],einName['nachname']
```

Das ist zwar ein bißchen mehr Schreibarbeit, aber es ist dafür auch noch nach Jahren beim ersten Lesen zu verstehen!

17.5. Der vierte Entwurf: Aktionen (aber nur angedeutet)!!

In diesem Entwurf wird das Adressbuch mit Leben gefüllt, das heißt, hier werden Funktionen eingefügt, die tatsächlich etwas mit den Daten machen. Zuerst muss aber ein Menu eingebaut werden, das diese Funktionen in einer Endlos-Schleife aufruft. Die Funktionen werden erst einmal als Dummy angelegt, das Menu selber fehlt noch. Aber wenn das Menu dann fertig ist, weiß ich schon, was gemacht werden soll: die Initialisierung wird gestartet, dann verschwindet man im Menu, bis dieses ausdrücklich wieder verlassen wird. Der Rahmen dafür sieht so aus:

Beispiel 17.5.1 Funktionen für die Adress-Liste(noch Dummies)

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   def initialisiere():
6     global namen
7     martin = {'vorname':'Martin', 'nachname':'Schimmels','alter':54}
8     ekki = {'vorname':'Eckard', 'nachname':'Krauss','alter':41}
9     namen = [martin,ekki]
10
11  def anzeigen(mitNummer = False):
12    print 'hier ist Funktion anzeigen'
13
14  def einfuegen():
15    print 'hier ist Funktion einfügen'
16
17  def aendern():
18    print 'hier ist Funktion ändern'
19
20  def loeschen():
21    print 'hier ist Funktion loeschen'
22
23  def beenden():
24    print 'Jetzt wird das Programm beendet'
25
26  def menu():
27    print 'tut mir leid, das Menu ist noch nicht fertig'
28
29  initialisiere()
30  menu()
31
```

An die Arbeit! Das Menu wird gebaut. Zuerst wird ein Schalter **weiter** eingefügt, dessen Wert standardmäßig 1 (und damit True) ist und dessen Wert nur bei Betätigung der Taste **E** auf 0 (und damit auf False) geändert wird. Danach wird in der Endlos-Schleife (solange weiter = True ist) ein Buchstabe eingelesen. Die vielen „\b“ bewirken, dass der Buchstabe in der letzten Zeile der Eingabeaufforderung an der Stelle der Punkte eingelesen wird. Jetzt passiert wieder etwas, was in Python so elegante Programm erlaubt. Wert in einem Dictionary kann irgendetwas sein: eine Zahl, ein Text, eine Liste, selbst noch ein Dictionary, aber auch eine Funktion. Das wird hier ausgenutzt: das Menu selber ist ein Dictionary, dessen Schlüssel die einzugebenden Buchstaben und dessen Werte die Funktionen sind. Damit ist der Aufruf der Funktion je nach eingegebenem Buchstabe nur noch das Herausholen eines Wertes aus dem Dictionary.

Beispiel 17.5.2 Adress-Liste mit Menu

```
#!/usr/bin/python
# -*- coding: utf8 -*-

def initialisiere():
  global namen
  martin = {'vorname':'Martin', 'nachname':'Schimmels','alter':54}
  ekki = {'vorname':'Eckard', 'nachname':'Krauss','alter':41}
  namen = [martin,ekki]

def anzeigen(mitNummer = False):
  print 'hier ist Funktion anzeigen'

def einfuegen():
  print 'hier ist Funktion einfügen'

def aendern():
  print 'hier ist Funktion ändern'

def loeschen():
  print 'hier ist Funktion loeschen'

def beenden():
  print 'Jetzt wird das Programm beendet'

def menu():
  weiter = 1
  while weiter:
    auswahl = raw_input('#### AUSWAHL ####\n
                  alle Namen anzeigen: Z\n
                  Namen einfügen: N\n
                  Namen löschen: L\n
                  Namen ändern: A\n
                  Ende: E\n
                  #### ..... ####\b\b\b\b\b\b\b\b\b\b\b\b')
    menuListe = {'N': einfuegen, 'L': loeschen,'A':aendern,
                          'E':beenden,  'Z':anzeigen}
    menuListe[auswahl]()
    if auswahl == 'E':
      weiter = 0

initialisiere()
menu()
```

17.6. Der fünfte Entwurf: Aktionen (jetzt tut sich wirklich etwas)!!

Jetzt wollen wir etwas sehen. Also fangen wir mit der Funktion `anzeigen` an. In dieser Funktion muss die Liste `namen` als global bekanntgegeben werden. Danach wird eine Überschriftszeile generiert. Die Ausgabe der einzelnen Datensätze geschieht in einer for-Schleife, analog wie im dritten Entwurf.

Beispiel 17.6.1 Adress-Liste anzeigen

```
1
2   def anzeigen():
3     global namen
4     print '##### Namensliste #####'
5     for einName in namen:
6       print '##\t',
7       print einName['vorname'],einName['nachname'],einName['alter'],'\n'
8     print '##### ----------- #####'
9
```

Das Einfügen eines neuen Namens in die Liste ist auch kein Hexenwerk. Die einzelnen Informationen werden über `raw_input` eingelesen, das ganze wird zu einem Dictionary verbunden und dieses Dictionary wird an die Liste `namen` angehängt

Beispiel 17.6.2 Eintrag in Adress-Liste machen

```
1
2   def einfuegen():
3     global namen
4     print 'Neuer Name wird eingefügt!'
5     vorname = raw_input('Vorname: ')
6     nachname = raw_input('Nachname: ')
7     alter = raw_input('Alter: ')
8     namen.append({'vorname':vorname,  'nachname':nachname,  'alter':alter})
9
```

Das Löschen eines Namens ist eigentlich auch nicht viel schwieriger. Dazu kommt die Funktion `del` für Listen ins Spiel, die mit der Nummer des Listenelements aufgerufen wird. Wenn die Liste allerdings länger (als 5) ist, wird die Abzählerei doch ein wenig lästig. Deswegen muss zuerst die Funktion `anzeigen` so abgeändert werden, dass zusätzlich zu den Informationen über die Personen auch noch die laufende Nummer in der Liste angezeigt wird. Dazu wird die Funktion `anzeigen` mit einem Parameter, der den Wahrheitswert `True` hat, aufgerufen. Der Standardwert für diesen Parameter ist `False`.

Beispiel 17.6.3 Adress-Liste mit Nummern anzeigen

```
1
2   def anzeigen(mitNummer = False):
3     global namen
4     i = 0
5     print '##### Namensliste #####'
6     for einName in namen:
7       print '##\t',
8       if mitNummer:
9         print i,
10        i += 1
11      print einName['vorname'],einName['nachname'],einName['alter'],'\n'
12    print '##### ----------- #####'
13
```

Und die Funktion „löschen“ gleich hintendran:

Beispiel 17.6.4 Adresse löschen

```
def loeschen():
  global namen
  print 'Diese Namen sind in der Liste! '
  anzeigen(1)
  loeschNr = input('Bitte Nummer des zu löschenden Namens eingeben: ')
  del namen[loeschNr]
```

Richtig umfangreich wird jetzt allerdings die Funktion `aendern`. Allerdings wirklich nur umfangreich, schwierig ist das nicht. Es werden wieder alle Einträge mit laufender Nummer angezeigt, man wählt eine Nummer aus, die Informationen zu einem Namen werden ausgegeben und eine Eingabe angefordert. Falls nur die **Enter**-Taste gedrückt wurde, das heißt also, wenn die Eingabe der leere String ist, wird der alte Wert beibehalten.

Beispiel 17.6.5 Adresse ändern

```
def aendern():
  global namen
  print 'Diese Namen sind in der Liste! '
  anzeigen(1)
  aendNr = input('Bitte Nummer des zu ändernden Namens eingeben: ')
  #### Vorname
  print 'alter Vorname: ', namen[aendNr]['vorname'],
  neuerVorname = raw_input(' neuer Vorname(Enter für beibehalten): ')
  if neuerVorname <> u'':
    namen[aendNr]['vorname'] = neuerVorname
  #### Nachname
  print 'alter Nachname: ', namen[aendNr]['nachname'],
  neuerNachname = raw_input(' neuer Nachname(Enter für beibehalten): ')
  if neuerNachname <> u'':
    namen[aendNr]['nachname'] = neuerNachname
  #### Alter
  print 'altes Alter: ', namen[aendNr]['alter'],
  neuesAlter = input(' neues Alter(Enter für beibehalten): ')
  if neuesAlter <> u'':
    namen[aendNr]['alter'] = neuesAlter
```

Das kann natürlich auch etwas weniger umfangreich geschrieben werden, indem man im Hauptprogramm die Struktur eines Datensatzes festlegt, diese Struktur in der Funktion `aendern` global macht und dann über die Felder dieser Struktur schleift:

Beispiel 17.6.6 Adress-Liste anzeigen (über eine Schleife)

```
strukturAdr = ['vorname', 'nachname', 'alter']

def aendern():
  global strukturAdr
  print 'Diese Namen sind in der Liste! '
  anzeigen(1)
  aendNr = input('Bitte Nummer des zu ändernden Namens eingeben: ')
  for attribut in strukturAdr:
    print attribut.capitalize(), ' (alt): ',  namen[aendNr][attribut]
    aenderung = raw_input(' neu (Enter für beibehalten): ')
    if aenderung <> u'':
      namen[aendNr][attribut] = aenderung
```

17.7. Der sechste Entwurf: persistente Speicherung

Die Daten sollen jetzt auch erhalten bleiben, wenn das Programm beendet wird und danach neu aufgerufen wird, sie sollen also in einer Datei gespeichert werden. Zu diesem Zweck wird das Modul `pickle` benutzt, der eine Hülle (ein Wrapper) für beliebige Datenstrukturen darstellt. Beim Aufruf des Programms wird überprüft, ob schon eine Adress-Datei existiert. Wenn ja, wird diese gelesen, ansonsten wird mit einer vorgegebenen Liste mit zwei Einträgen gearbeitet. Diese Entscheidung wird in der ursprünglichen Funktion `initialisiere` getroffen, wobei über eine Ausnahme eine Nicht-Existenz der Datei abgefangen wird. Falls die Datei existiert, wird in einer neuen Funktion `nDatLesen`die Datei gelesen.

Beispiel 17.7.1 gespeicherte Adress-Liste lesen

```
def nDatLesen(namenDatei):
  global namen
  while namenDatei:
    try:
      namen = pickle.load(namenDatei)
    except EOFError:
      break
  namenDatei.close()

def initialisiere():
  global namen
  nDat = 'meineNamen.dat'
  try:
    namenDatei = open(nDat, 'r')
    nDatLesen(namenDatei)
  except IOError:
    martin = {'vorname':'Martin', 'nachname':'Schimmels','alter':54}
    ekki = {'vorname':'Eckard', 'nachname':'Krauss','alter':41}
    namen = [martin,ekki]
```

Beim Beenden des Programms wird jetzt die Liste in eine Datei geschrieben. Sicherheitshalber wird überprüft, ob sich die Datei zum Schreiben öffnen lässt und im Falle des Mißlingens wird das über eine Ausnahme abgefangen. Ansonsten ist das Schreiben einer beliebigen Struktur einfach: `pickle` nimmt die Struktur und schreibt sie per dump, so wie sie ist, in eine Datei.

Beispiel 17.7.2 Adress-Liste schreiben

```
def beenden():
  print 'Jetzt wird das Programm beendet'
  global namen
  nDat = 'meineNamen.dat'
  try:
    namenDatei = open(nDat, 'w')
  except IOError:
    print 'na so was'
  pickle.dump(namen,  namenDatei)
  namenDatei.close()
```

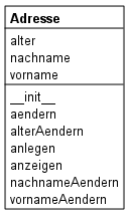
17.8. Jetzt wird es objektorientiert

17.8.1. Der Klassenentwurf

Wer bis hierher die verschiedenen Stadien des Programms verfolgt hat und weiter oben die Beispiele objektorientierter Programmierung studiert hat, wird an den bisherigen Entwürfen viele objektorientierte Ansätze feststellen. Dann soll das Programm jetzt also wirklich objektorientiert werden. Dazu wird zuerst einmal eine Klasse `Adresse` mit ihren Attributen und Methoden entworfen und getestet. In einem weiteren Schritt wird eine Klasse Adressbuch hinzugefügt.

Die Klasse `Adresse` hat die Attribute Vorname, Nachname und Alter, außerdem Methoden, um jedes der Attribute zu ändern und ein Menu, das in die verschiedenen Änderungen verzweigt. Außerdem können Datensätze angelegt und angezeigt werden.

Abbildung 17.1. Klassendiagramm `Adresse`



Der Klassenentwurf ist danach selbsterklärend:

Beispiel 17.8.1 Klasse Adresse

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   class Adresse():
6     def __init__(self,  vorname='',  nachname='',  alter=0):
7       if vorname == '' and nachname == '' and alter == 0:
8         self.anlegen()
9       else:
10        self.vorname = vorname
11        self.nachname = nachname
12        self.alter = alter
13
14    def anzeigen(self):
15      print self.vorname, ' ', self.nachname,  ' ',  self.alter
16
17    def vornameAendern(self):
18      aString = 'alter Vorname: '+self.vorname+ ' neuer Vorname: '
19      self.vorname = raw_input(aString)
20
21    def nachnameAendern(self):
22      aString = 'alter Nachname: '+self.nachname+' neuer Nachname: '
23      self.nachname = raw_input(aString)
24
25    def alterAendern(self):
26      aString = 'altes Alter: '+str(self.alter)+' neues Alter: '
27      self.alter = raw_input(aString)
28
29    def aendern(self):
30      auswahl = raw_input( 'Was soll geändert werden? (V)orname,
31                                          (N)achname, (A)lter ' )
32      menuAendern = {'V': self.vornameAendern,  'N': self.nachnameAendern,
33                                  'A':self.alterAendern}
34      menuAendern[auswahl]()
35
36    def anlegen(self):
37      print 'neuer Name wird aufgenommen!'
38      self.vorname = raw_input('Vorname: ')
39      self.nachname = raw_input('Nachname: ')
40      self.alter = raw_input('Alter: ')
41
```

Dazu benötigt man wieder ein Programm, das diese Klasse aufruft:

Beispiel 17.8.2 Aufruf der Klasse `Adresse`

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   from clAdr import Adresse
6
7   ich = Adresse('Martin',  'Schimmels',  54)
8   ich.anzeigen()
9   ich.aendern()
10  ich.anzeigen()
11
12  du = Adresse()
13  du.anzeigen()
14
```

17.8.2. Zur Klasse `Adresse` kommt die Klasse `Adressbuch` hinzu

Im nächsten Schritt wird eine weitere Klasse erstellt, die Elemente der Klasse Adresse enthält: das Adressbuch. Die Realisierung dieses Adressbuches sieht der Anwendung im fünften Entwurf sehr ähnlich.

Beispiel 17.8.3 Adressbuch-Klasse

```
1
2   class AdressBuch():
3     def __init__(self):
4       self.weiter = True
5       self.adrListe = []
6       while self.weiter:
7         self.menu()
8
9     def menu(self):
10      auswahl = raw_input('\n#### AUSWAHL ####\n
11                  alle Namen anzeigen: Z\n
12                  Namen einfügen: N\n
13                  Namen löschen: L\n
14                  Namen ändern: A\n
15                  Ende: E\n
16                  #### ..... ####\b\b\b\b\b\b\b\b\b\b\b\b')
17      menuListe = {'N':self.einfuegen, 'L':self.loeschen,
18                   'A':self.aendern, 'E':self.beenden, 'Z':self.anzeigen}
19      menuListe[auswahl]()
20
21    def einfuegen(self):
22      neuerName = Adresse()
23      self.adrListe.append(neuerName)
24
25    def loeschen(self):
26      print 'Folgende Namen sind in der Liste: '
27      self.anzeigen()
28      loeschNr = input('Nummer des zu löschenden Datensatzes angeben: ')
29      del self.adrListe[loeschNr]
30      print "So sieht's jetzt aus: "
31      self.anzeigen()
32
33    def aendern(self):
34      print 'Folgende Namen sind in der Liste: '
35      self.anzeigen()
36      aendNr = input('Nummer des zu ändernden Datensatzes angeben: ')
37      self.adrListe[aendNr].aendern()
38
39    def beenden(self):
40      self.weiter = False
41
42    def anzeigen(self):
43      lfdNr = 0
44      print '\n##### Namensliste #####'
45      for einName in self.adrListe:
46        print '##\t',lfdNr,'\t',
47        einName.anzeigen()
48        lfdNr += 1
49      print '##### ---------- #####'
50
```

Das Adressbuch selber wird durch das folgende aufrufende Programm realisiert:

Beispiel 17.8.4 Aufruf des Adressbuchs

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   from clAdr import AdressBuch
6
7   meineAdressen = AdressBuch()
8
```

Das soll wirklich alles sein? Wer es nicht glaubt, soll es halt einfach ausprobieren!!

17.8.3. Dauerhafte Speicherung

Das wird wieder fast wortwörtlich aus dem fünften Entwurf übernommen:

Beispiel 17.8.5 Adressbuch-Klasse mit Speicherung

```
#!/usr/bin/python
# -*- coding: utf8 -*-
import pickle
class Adresse():
  def __init__(self,  vorname='',  nachname='',  alter=0):
    if vorname == '' and nachname == '' and alter == 0:
      self.anlegen()
    else:
      self.vorname = vorname
      self.nachname = nachname
      self.alter = alter

  def anzeigen(self):
    print self.vorname, ' ', self.nachname,  ' ',  self.alter

  def vornameAendern(self):
    aString = 'alter Vorname: '+self.vorname+ ' neuer Vorname: '
    self.vorname = raw_input(aString)

  def nachnameAendern(self):
    aString = 'alter Nachname: '+self.nachname+' neuer Nachname: '
    self.nachname = raw_input(aString)

  def alterAendern(self):
    aString = 'altes Alter: '+str(self.alter)+' neues Alter: '
    self.alter = raw_input(aString)

  def aendern(self):
    auswahl = raw_input('Was soll geändert werden?
                (V)orname, (N)achname, (A)lter ' )
    menuAendern = {'V': self.vornameAendern,
                   'N': self.nachnameAendern,
                   'A':self.alterAendern}
    menuAendern[auswahl]()

  def anlegen(self):
    print 'neuer Name wird aufgenommen!'
    self.vorname = raw_input('Vorname: ')
    self.nachname = raw_input('Nachname: ')
    self.alter = raw_input('Alter: ')

```

Beispiel 17.8.6 Adressbuch-Klasse mit Speicherung (Forts.)

```
1
2   class AdressBuch():
3
4     def __init__(self):
5       self.weiter = True
6
7       self.nDat = 'meineNamen.dat'
8       try:
9         namenDatei = open(self.nDat, 'r')
10        self.nDatLesen(namenDatei)
11      except IOError:
12        self.adrListe = []
13      while self.weiter:
14        self.menu()
15
16    def nDatLesen(self, namenDatei):
17      while namenDatei:
18        try:
19          self.adrListe = pickle.load(namenDatei)
20        except EOFError:
21          break
22      namenDatei.close()
23
24    def menu(self):
25      auswahl = raw_input('\n#### AUSWAHL ####\n
26                   alle Namen anzeigen: Z\n
27                   Namen einfügen: N\n
28                   Namen löschen: L\n
29                   Namen ändern: A\n
30                   Ende: E\n
31                   #### ..... ####\b\b\b\b\b\b\b\b\b\b\b\b\b')
32      menuListe = {'N':self.einfuegen,
33                   'L':self.loeschen,
34                   'A':self.aendern,
35                   'E':self.beenden,
36                   'Z':self.anzeigen}
37      menuListe[auswahl]()
38
39    def einfuegen(self):
40      neuerName = Adresse()
41      self.adrListe.append(neuerName)
42
43    def loeschen(self):
44      print 'Folgende Namen sind in der Liste: '
45      self.anzeigen()
46      loeschNr = input('Nummer des zu löschenden Datensatzes angeben: ')
47      del self.adrListe[loeschNr]
48      print "So sieht's jetzt aus: "
49      self.anzeigen()
50
```

Beispiel 17.8.7 Adressbuch-Klasse mit Speicherung (2. Forts.)

```
1
2       def aendern(self):
3         print 'Folgende Namen sind in der Liste: '
4         self.anzeigen()
5         aendNr = input('Nummer des zu ändernden Datensatzes angeben: ')
6         self.adrListe[aendNr].aendern()
7
8       def beenden(self):
9         self.weiter = False
10        try:
11          namenDatei = open(self.nDat, 'w')
12        except IOError:
13        print 'na so was'
14        pickle.dump(self.adrListe,  namenDatei)
15        namenDatei.close()
16
17      def anzeigen(self):
18        lfdNr = 0
19        print '\n##### Namensliste #####'
20        for einName in self.adrListe:
21          print '##\t',lfdNr,'\t',
22          einName.anzeigen()
23          lfdNr += 1
24        print '##### ---------- #####'
25
```

18. Immer noch das Adressbuch — nur schöner

18.1. Das Adressbuch kommt in die Datenbank

Die permanente Speicherung der Daten in einer Datei mit Hilfe von `pickle` ist nicht der Weisheit letzter Schluß. Das vorhergehende Beispiel wird daher so erweitert, dass die Adress-Daten jetzt in einer Tabelle einer relationen Datenbank gespeichert werden. Der Open-Source-Standard für solch kleine Anwendungen ist das „DBMS“ `mysql`.

An der Klasse `Adresse` muss zuallererst eine Kleinigkeit geändert werden. Da die Daten in der Datenbank einen Primärschlüssel haben sollten, wird in der Tabellenstruktur für die Adressen eine laufende Nummer eingefügt. Diese laufende Nummer muss auch in der Klasse `Adresse` bearbeitet werden. Zusätzlich wird, weil die beiden Klassen `Adresse` und `Adressbuch` in der selben Datei stehen, bereits hier die MySQL-Bibliothek eingebunden. Die Klasse sieht dann so aus:

Beispiel 18.1.1 Klasse `Adresse` mit laufender Nummer

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4   import MySQLdb
5   class Adresse():
6     def __init__(self, lfdnr = 0, vorname='', nachname='',
7                        strasse = '', plz = '',  ort = ''):
8       if vorname == '' and nachname == '':
9         self.anlegen()
10      else:
11        self.lfdnr =  lfdnr
12        self.vorname = vorname
13        self.nachname = nachname
14        self.strasse =  strasse
15        self.plz = plz
16        self.ort = ort
17
18    def anzeigen(self):
19      print '\n######## Nr. des Datensatzes: ', self.lfdnr,
20            '\n## ',self.vorname, ' ', self.nachname,
21            '\n## ',  self.strasse, '\n## ',
22            self.plz, ' ',  self.ort, '\n########'
23
24    def vornameAendern(self):
25      print 'häää?? Vornamen ändern??'
26      aString = 'alter Vorname: '+self.vorname+ ' neuer Vorname: '
27      self.vorname = raw_input(aString)
28
29    def nachnameAendern(self):
30      aString = 'alter Nachname: '+self.nachname+' neuer Nachname: '
31      self.nachname = raw_input(aString)
32
33    def strAendern(self):
34      aString = 'alte Strasse: '+str(self.strasse)+' neue Strasse: '
35      self.strasse = raw_input(aString)
36
37    def plzAendern(self):
38      aString = 'alte PLZ: '+str(self.plz)+' neue PLZ: '
39      self.plz = raw_input(aString)
40
41    def ortAendern(self):
42      aString = 'alter Ort: '+str(self.ort)+' neuer Ort: '
43      self.ort = raw_input(aString)
44
45    def aendern(self):
46      auswahl = raw_input('Was soll geändert werden?
47                           (V)orname, (N)achname, (A)lter,
48                           (S)trasse, (P)LZ, (O)rt ' )
49      menuAendern = {'V': self.vornameAendern, 'N': self.nachnameAendern,
50                     'S': self.strAendern, 'P': self.plzAendern,
51                     'O': self.ortAendern}
52      menuAendern[auswahl]()
53
```

Beispiel 18.1.2 Klasse `Adresse` mit laufender Nummer (Forts.)

```
    def anlegen(self):
      print 'neuer Name wird aufgenommen!'
      self.vorname = raw_input('Vorname: ')
      self.nachname = raw_input('Nachname: ')
      self.strasse = raw_input('Strasse: ')
      self.plz = raw_input('PLZ: ')
      self.ort = raw_input('Ort: ')

```

Die Klasse `AdressBuch` ändert sich mehr. Im Konstruktor der Klasse wird weiterhin zuerst eine leere Liste `adrListe` erzeugt, in der für den Fall, dass alle Datensätze gelesen werden sollen, diese zwischengespeichert werden. Danach wird die Verbindung zum DBMS hergestellt und versucht, die Datenbank `test_adr` zu öffnen. Falls das nicht gelingt, wird diese Datenbank erstellt. In einem zweiten Schritt wird dann versucht, die Tabelle `namen` zu lesen, im Falle des Mißlingens wird eine solche Tabelle erstellt. Danach wird die Liste gefüllt (eventuell bleibt sie leer, falls in der Tabelle der Datenbank noch keine Daten vorhanden sind). Im letzten Schritt des Konstruktors wird die Endlos-Schleife des Menus aufgerufen. Außerdem wird hier am Ende jeder Transaktion die Liste geleert und neu aus der Datenbank eingelesen.

Beispiel 18.1.3 Konstruktor der Klasse `AdressBuch`

```
class AdressBuch():

  def __init__(self):
    self.weiter = True
    self.adrListe = []
    try:
      self.verbindung =  MySQLdb.connect(host='localhost',  db='test_adr')
      self.meinCursor = self.verbindung.cursor()
    except:
      print 'Datenbank "test_ms" existiert noch nicht und wird angelegt'
      self.verbindung =  MySQLdb.connect(host='localhost')
      self.meinCursor = self.verbindung.cursor()
      self.meinCursor.execute('CREATE DATABASE test_adr')
    self.meinCursor.execute('use test_adr')
    try:
      self.meinCursor.execute('SELECT COUNT(*) FROM namen')
    except:
      self.meinCursor.execute('CREATE TABLE namen (
                               lfdnr INT NOT NULL AUTO_INCREMENT,
                               vorname VARCHAR(30),
                               nachname VARCHAR(30),
                               str VARCHAR(30),
                               plz CHAR(5),
                               ort VARCHAR(30),
                               PRIMARY KEY(lfdnr))')
    self.nDatLesen()
    while self.weiter:
      self.menu()
      self.adrListe = []
      self.nDatLesen()

```

Die einzelnen Methoden der Klasse `AdressBuch` sehen denen aus dem vorigen Beispiel sehr ähnlich.

Es wird (fast) immer ein Query-String aufgebaut, der dann mit dem Befehl `execute` ausgeführt wird. Bei der Methode `aendern` muss allerdings eine andere Logik verwendet werden: vor dem Ändern wird der zu ändernde Datensatz in ein Objekt der Klasse `Adresse` eingelesen, dieses Objekt wird für die Änderung an die Klasse `Adresse` weitergegeben, und das veränderte Objekt, das aus der Klasse zurückkommt, wird in die Datenbank geschrieben. Da man nicht weiß, welches Attribut des Objekts geändert wurde, wird einfach jedes Attribut geändert.

Beispiel 18.1.4 Methoden der Klasse `AdressBuch`

```
1
2     def nDatLesen(self):
3       self.meinCursor.execute('SELECT * FROM namen ORDER BY nachname, vorname')
4       for (lfdnr, vn,  nn,  strasse,  plz,  ort) in self.meinCursor.fetchall():
5         geleseneAdr = Adresse(lfdnr,  vn,  nn,  strasse,  plz,  ort)
6         self.adrListe.append(geleseneAdr)
7
8     def menu(self):
9       auswahl = raw_input('\n#### AUSWAHL ####\n
10                           alle Namen anzeigen: Z\n
11                           Namen einfügen: N\n
12                           Namen löschen: L\n
13                           Namen ändern: A\n
14                           Ende: E\n
15                           #### ..... ####\b\b\b\b\b\b\b\b\b\b\b\b')
16      menuListe = {'N':self.einfuegen, 'L':self.loeschen,
17                   'A':self.aendern, 'E':self.beenden, 'Z':self.anzeigen}
18      menuListe[auswahl]()
19
20    def einfuegen(self):
21      neuerName = Adresse()
22      self.meinCursor.execute('INSERT INTO namen
23                               (vorname, nachname, str, plz, ort)
24                               VALUES(%s, %s, %s, %s, %s)',
25                               (neuerName.vorname, neuerName.nachname,
26                                neuerName.strasse, neuerName.plz,
27                                neuerName.ort))
28
29    def loeschen(self):
30      print 'Folgende Namen sind in der Liste: '
31      self.anzeigen()
32      loeschNr = input('Nummer des zu löschenden Datensatzes angeben: ')
33      loeschBefehl = 'DELETE FROM namen WHERE lfdnr = '+str(loeschNr)
34      self.meinCursor.execute(loeschBefehl)
35
```

Beispiel 18.1.5 Methoden der Klasse `AdressBuch` (Forts.)

```
1
2       def aendern(self):
3         print 'Folgende Namen sind in der Liste: '
4         self.anzeigen()
5         aendNr = input('Nummer des zu ändernden Datensatzes angeben: ')
6         vorleseBefehl = 'SELECT * FROM namen WHERE lfdnr = '+str(aendNr)
7         self.meinCursor.execute(vorleseBefehl)
8         for (lfdnr, vn,  nn,  strasse,  plz,  ort) in self.meinCursor.fetchall():
9           geleseneAdr = Adresse(lfdnr,  vn,  nn,  strasse,  plz,  ort)
10
11        geleseneAdr.aendern()
12        updBefehl = "UPDATE namen SET vorname = '"+geleseneAdr.vorname+"',
13                                      nachname = '"+geleseneAdr.nachname+"',
14                                      str = '"+geleseneAdr.strasse+"',
15                                      plz = '"+geleseneAdr.plz+"',
16                                      ort ='"+geleseneAdr.ort+"'
17                                      WHERE lfdnr = "+str(aendNr)
18        self.meinCursor.execute(updBefehl)
19
20      def beenden(self):
21        self.weiter = False
22
23      def anzeigen(self):
24        lfdNr = 0
25        print '\n##### Namensliste #####'
26
27        for einName in self.adrListe:
28          einName.anzeigen()
29        print '##### ---------- #####'
30
```

19. Eine Ampel

19.1. Der Entwurf

The traffic lights they turn blue tomorrow

(Jimi Hendrix[1])

In meinem nächsten Beispiel soll eine funktionierende Ampel programmiert werden. Dabei soll die in Deutschland übliche Reihenfolge der Ampelfarben angezeigt werden, d.h. rot -> rot-gelb -> grün -> gelb -> rot. Starten wir also mit einem Modell in UML.

Betrachten wir zuerst die Attribute der Ampel. Eine Ampel hat also verschiedene Zustände: rot, rot-gelb, grün, gelb, also ein Attribut `zustand`. Diese Zustände werden in zwei Dictionaries abgelegt. Im ersten Dictionary wird den Zahlen 0 bis 3 jeweils einer der Zustände zugeordnet, und zwar in der Reihenfolge, die oben angegeben wurde. Ferner wird ein zweites Dictionary angelegt, in dem den Zuständen (in der beschreibenden Form rot, rot-gelb usw.) ein Tripel von Farben zugeordnet wird. Die Farben werden der Übersichtlichkeit halber aus einem weiteren Dictionary genommen, in dem dem Farbnamen der übliche numerische Wert in der RGB-Schreibweise zugeordnet wird. Das letzte Attribut der Ampel ist der aktuelle Zustand.

Die Klasse Ampel hat nur zwei Methoden. Zum einen die Methode `umschalten`, die Modulo 3 durch die numerischen Zustände wechselt, zum zweiten die Methode `anzeigen`, die genau das macht. Das UML-Diagramm sieht also so aus:

Abbildung 19.1. Klassendiagramm `Ampel`

19.2. Die Realisierung

Der Quellcode für diese Klasse sieht so aus:

[1] The wind cries Mary *auf:* Are you experienced?

Beispiel 19.2.1 Klassenentwurf der Ampel

```
1
2   #!/usr/bin/python
3
4  class Ampel():
5    def __init__(self):
6      self.zustaendeNum = {0:'rot',  1:'rot-gelb', 2:'gruen', 3:'gelb'}
7      self.farben = {'rot':'#FF0000', 'gelb':'#FFFF00',
8                     'gruen':'#00FF00', 'schwarz':'#000000'}
9      self.zustaende = {
10       'rot':[self.farben['rot'], self.farben['schwarz'], self.farben['schwarz']],
11       'rot-gelb':[self.farben['rot'], self.farben['gelb'], self.farben['schwarz']],
12       'gruen':[self.farben['schwarz'], self.farben['schwarz'], self.farben['gruen']],
13       'gelb':[self.farben['schwarz'], self.farben['gelb'], self.farben['schwarz']]}
14     self.zustand = 0
15
16   def umschalten(self):
17     # das eigentliche Umschalten. BEACHTE: hier wird mod(3) gerechnet
18     self.zustand = (self.zustand + 1)%len(self.zustaendeNum)
19
20   def anzeigen(self):
21     print '\n=========\nDie Ampel ist %s' % self.zustaendeNum[self.zustand]
22     print 'Farben: ',  self.zustaende[self.zustaendeNum[self.zustand]]
23
```

Das aufrufende Programm ist wieder sehr einfach. Hier wird eine Endlosschleife benutzt, um die Ampel umzuschalten.

Beispiel 19.2.2 Aufruf einer Ampel

```
#!/usr/bin/python

from clAmpel import Ampel

eingabe = '12345'
meineAmpel =  Ampel()
while eingabe <> 'x':
  eingabe = raw_input('Umschalten!!!  Ende mit x')
  meineAmpel.umschalten()
  meineAmpel.anzeigen()
```

19.3. Persistente Speicherung

Die Ampel funktioniert. Sie schaltet um und zeigt den aktuellen Zustand an. Aber leider merkt sich die Ampel nicht, welchen Zustand sie gerade hat. Bei jedem Neustart des Programms ist der Zustand der selbe, nämlich der, der im Konstruktor angegeben wird. Um das zu verbessern, wird der aktuelle Zustand in eine Datei geschrieben und bei jedem Neustart des Programms aus dieser Datei gelesen.

Beispiel 19.3.1 Klasse Ampel mit Speicherung des Zustands

```
1
2  #!/usr/bin/python
3
4  class Ampel():
5    def __init__(self,  bezeichnung = 'Nord'):
6      self.bezeichnung = bezeichnung
7      self.zustaendeNum = {0:'rot',  1:'rot-gelb', 2:'gruen', 3:'gelb'}
8      self.farben = {'rot':'#FF0000',  'gelb':'#FFFF00',
9                             'gruen':'#00FF00', 'schwarz':'#000000'}
10     self.zustaende = {
11       'rot':[self.farben['rot'], self.farben['schwarz'], self.farben['schwarz']],
12       'rot-gelb':[self.farben['rot'], self.farben['gelb'], self.farben['schwarz']],
13       'gruen':[self.farben['schwarz'], self.farben['schwarz'], self.farben['gruen']],
14       'gelb':[self.farben['schwarz'], self.farben['gelb'], self.farben['schwarz']]}
15     self.lampen = ['oben',  'mitte',  'unten']
16
17   def umschalten(self):
18     # aus Datei den aktuellen Zustand lesen
19     try:
20       self.dateiOeffnen('r')
21       self.zustand = int(self.meineDat.read())
22       self.dateiSchliessen()
23     except:
24       self.zustand = 0
25
26     # das eigentliche Umschalten. BEACHTE: hier wird mod(3) gerechnet
27     self.zustand = (self.zustand + 1)%len(self.zustaendeNum)
28
29     # in Datei den aktuellen Zustand schreiben
30     self.dateiOeffnen('w')
31     self.dateiSchreiben()
32     self.dateiSchliessen()
33
34   def anzeigen(self):
35     print '\n=========\nDie Ampel ist %s' % self.zustaendeNum[self.zustand]
36     print 'Farben: ',  self.zustaende[self.zustaendeNum[self.zustand]]
37
38 ### Datei-Operationen
39   def dateiOeffnen(self,  modus):
40     dateiName = self.bezeichnung+'.dat'
41     self.meineDat = open(dateiName,modus)
42
43   def dateiSchreiben(self):
44     self.meineDat.write(str(self.zustand))
45
46   def dateiSchliessen(self):
47     self.meineDat.close()
48
```

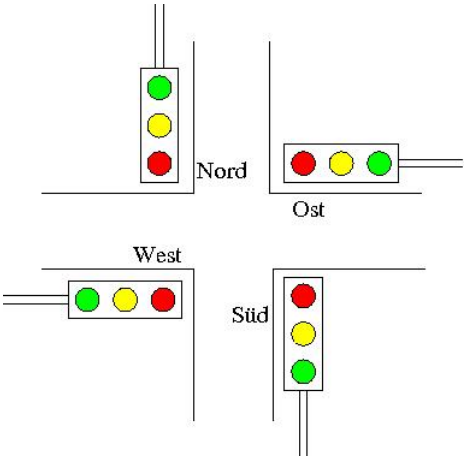
Das aufrufende Programm muss nicht verändert werden!

19.4. Eine Kreuzung hat 4 Ampeln! Der Entwurf.

Wenn eine Ampel funktioniert, kann man mal vier Ampeln an einer Kreuzung aufstellen. Der Einfachheit halber werden die 4 Straßen, die an der Kreuzung zusammentreffen, und damit die 4 Ampeln, die

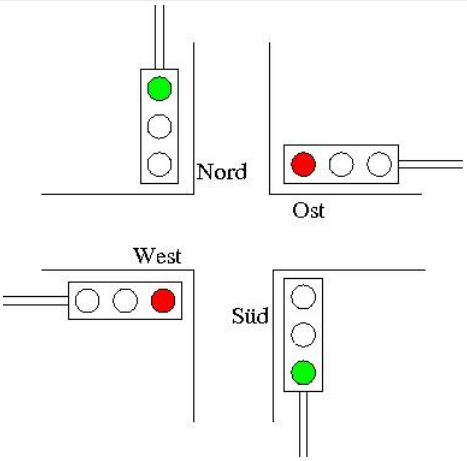
an den jeweiligen Straßen stehen, mit den 4 Himmelsrichtungen benannt.

Abbildung 19.2. Eine Kreuzung



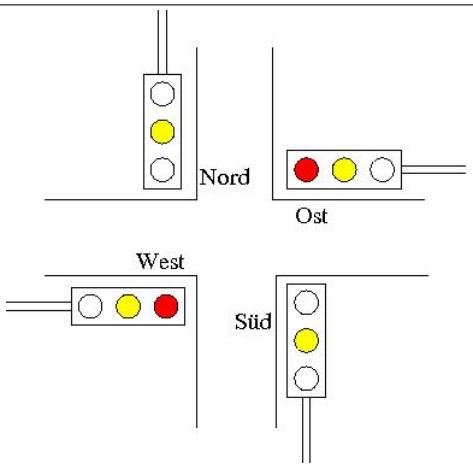
Aber natürlich sind nicht immer alle Lichter aller 4 Ampeln an. Die Regeln für die Ampeln sind aus den folgenden 4 Zeichnungen zu entnehmen. Zuerst darf man von Nord nach Süd (und umgekehrt) fahren.

Abbildung 19.3. Kreuzung: Freie Fahrt von Nord nach Süd



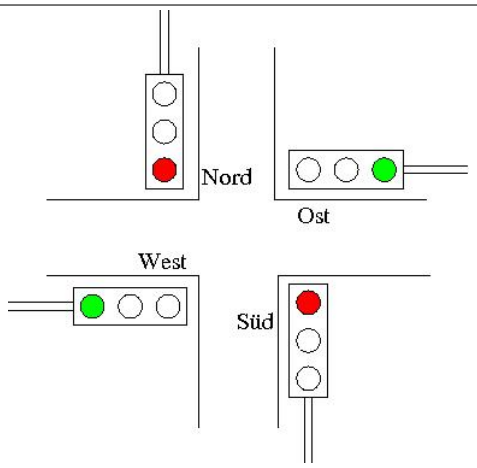
Dann wird umgeschaltet in die Gelb-Phase.

Abbildung 19.4. Kreuzung: Alle warten



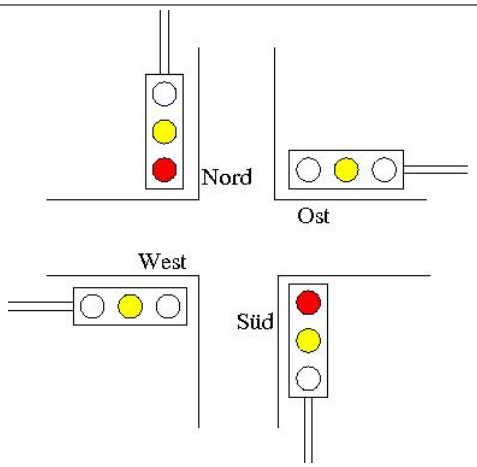
Dann darf man von West nach Ost (und umgekehrt) fahren.

Abbildung 19.5. Kreuzung: Freie Fahrt von West nach Ost



Dann wird umgeschaltet in die andere Gelb-Phase.

Abbildung 19.6. Kreuzung: Wieder warten alle



In einer Tabelle sieht das so aus:

Tabelle 19.1. Die Ampel-Phasen

Beschreibung	Nord	Ost	Süd	West
freie Fahrt N - S	grün	rot	grün	rot
Umschalten	gelb	rot - gelb	gelb	rot - gelb
freie Fahrt W - O	rot	grün	rot	grün
Umschalten	rot - gelb	gelb	rot - gelb	gelb

19.5. Die Realisierung

Da die Klasse Ampel so gut funktioniert, ist die Klasse Kreuzung ganz kurz. Und viel erklärt werden muss hier auch nicht. Hier also der Code.

Beispiel 19.5.1 Klassenentwurf der Kreuzung

```
1
2  #!/usr/bin/python
3
4  from clAmpel import Ampel
5
6  class Kreuzung():
7    def __init__(self):
8      self.richtungen = ['Nord',  'West',  'Sued',  'Ost']
9
10     self.nordAmpel = Ampel('Nord')
11     self.westAmpel = Ampel('West')
12     self.suedAmpel = Ampel('Sued')
13     self.ostAmpel = Ampel('Ost')
14     self.alleAmpeln = [self.nordAmpel, self.westAmpel, self.suedAmpel, self.ostAmpel]
15
16   def umschalten(self):
17     for eineAmpel in self.alleAmpeln:
18       eineAmpel.umschalten()
19
```

Weiter hinten bei [Ampel im WWW], wenn Python als Sprache für dynamische Web-Seiten behandelt wird, gibt es die Realisierung der Kreuzung als HTML-Seite mit einem Knopf zum Umschalten.

Teil X.

Grafik! Internet!

20. CGI-Programme

20.1. HTML und Kollegen

Hier soll keine Einführung in HTML gegeben werden. Eine gute Empfehlung im WWW für die, die HTML lernen wollen, ist immer noch Self-HTML. [1]

Auf den folgenden Seiten wird nur ein kurzer Abriss von HTML gegeben. Für die Programmierung von dynamischen Seiten mit Python ist das aber ausreichend.

20.2. Allgemeines zu HTML

20.2.1. Was ist HTML?

HTML ist die Abkürzung für „Hypertext Markup Language".

HTML gehört zu den Auszeichnungssprachen. Bei Dokumenten, die in einer Auszeichnungssprache geschrieben werden, wird nur die Struktur des Textes beschrieben, nicht das tatsächliche Aussehen.

Die Darstellung eines HTML-Dokuments ist die Aufgabe eines Browsers. Da es verschiedene Browser gibt, kann es vorkommen, daß Dokumente auch verschieden aussehen. Das kann sich auf die Schrift auswirken: manche Schriften existieren im einen System, im anderen nicht. Unangenehm wird es, wenn der Hersteller eines Browsers meint, von den festgelegten Standards abweichen zu müssen. (Meistens stecken da wirtschaftliche Interessen dahinter.)

Es gibt inzwischen nicht nur viele verschiedene Browser, sondern auch verschiedene HTML-Editoren. Das sind Editoren, mit denen man „auf Knopfdruck" HTML-Code schreiben kann. Warum soll man dann diese Sprache noch lernen? Dafür gibt es immer noch ein paar Gründe:

- Wenn man keine Vorstellung von der Struktur einer Seite hat, dann bewirkt die Benutzung eines HTML-Editors meistens, daß eine damit erstellte Seite nicht sehr elegant aussieht.
- Die meisten HTML-Editoren hinken den Browsern hinterher! Nicht alles, was angezeigt werden kann, beherrscht auch der Editor.
- Die Einbindung von dynamischen Web-Seiten erfordert Programmierung in einer Skript-Sprache; spätestens hier muß man den HTML-Code „zu Fuß" eingeben.
- Auch die Erstellung einer HTML-Seite, die mit Werten aus einer Datenbank gefüllt wird, benötigt HTML-Befehle.

20.2.2. Grundlagen

HTML-Code besteht aus dem Inhalt und den Auszeichnungen. Diese Auszeichnungen werden als „tag" bezeichnet. Ein HTML-Befehl umschließt einen Bereich und wird von einem „Start-tag" und einem „Ende-tag" eingerahmt. Ein solcher Bereich wird auch „Tag-Container" genannt. HTML-tags können geschachtelt werden. Start-tags sind in größer- und kleiner-Zeichen eingeschlossen. Ende-tags unterscheiden sich von Start-tags dadurch, daß das erste Zeichen nach dem kleiner-Zeichen ein Schrägstrich ist. Beispiele siehe nächstes Kapitel.

Einige tags können Attribute haben, die das Verhalten des tags ändern. Die wichtigsten Attribute werden weiter unten im Kontext genannt.

20.2.3. Obligatorische HTML-Befehle

Jedes HTML-Dokument benötigt wenigstens folgende HTML-tags:

1. <HTML> und </HTML>

[1] siehe hierzu: <http://de.selfhtml.org/>

2. <HEAD> und </HEAD>
3. <TITLE> und </TITLE>
4. <BODY> und </BODY>

Damit sieht ein einfaches HTML-Dokument (ohne jeglichen Inhalt) so aus:

Beispiel 20.2.1

```
<HTML>
  <HEAD>
    <TITLE>
    </TITLE>
  </HEAD>
  <BODY>
  </BODY>
</HTML>
```

20.2.4. Struktur eines HTML-Dokuments

Es gibt 6 verschiedene Hierarchiestufen für Überschriften.

20.2.5. Absätze und Leerzeichen

Bei der Verarbeitung von HTML-Code werden Leerzeichen und Leerzeilen ignoriert; Aus mehreren Leerzeichen wird ein einzelnes gemacht. Deswegen sind Absätze und Leerzeichen besonders zu bearbeiten. Ein Absatz wird durch <P> begonnen und durch </P> beendet. Absätze dürfen nicht geschachtelt werden.

Sollen Leerzeichen nicht zusammengeschoben werden, kann man das dadurch verhindern, daß man einen Block dadurch schützt, daß er in den HTML-Container <PRE> ... </PRE> (für „to preserve: erhalten“) gesteckt wird.

Soll nur eine neue Zeile begonnen werden, ohne daß ein Absatz-Abstand eingefügt werden soll, so wird das durch den HTML-Befehl
 erledigt. Vorsicht: zu diesem „tag“ gibt es kein Ende-tag.

20.2.6. Hervorhebungen

Besondere Textstellen können auf verschiedene Art hervorgehoben werden:

- <EM>*hebt Text einfach hervor. Dies geschieht durch kursive Schrift.*
- <STRONG>**hebt Text verstärkt hervor. Dies geschieht durch fette Schrift.**
- <CODE> [code] ist für Programmcode gedacht. Dieser Text wird in Schreibmaschinenschrift wiedergegeben. [/code]
- <SAMP> Dieser Text wird (auf einer HTML-Seite) in Schreibmaschinenschrift wiedergegeben.
- <KBD>**ist für Tastatureingaben gedacht.**
- <CITE> für Namen und Titel eines Werkes, das zitiert wird.
- <BLOCKQUOTE> wird benutzt, wenn ein längerer Text zitiert werden soll. Der von diesen tags eingeschlossene Block wird eingerückt und als Block gesetzt. Das wirkt erst, wenn der Text über mehrere Zeilen geht.

Wenn man einen ganzen Absatz besonders darstellen will, bietet sich der <DIV>-Container an. Damit wird ein beliebiges Stück Text zusammengefasst, dem man mittels diverser Attribute ein eigenes Aussehen verpassen kann.

20.2.7. Listen

1. **Geordnete Listen.** Geordnete Listen sind durch Numerierung der Listeneinträge gekennzeichnet; das kann verschiedene Formen haben: 1., 2., 3., usw. oder a), b), c) usw. Geordnete Listen werden in den <OL>-Container gepackt. Listenelemente werden in den <LI>-Container gepackt.
2. **Ungeordnete Listen.** Ungeordnete Listen werden durch Rauten, Rechtecke, etc. gekennzeichnet. Ungeordnete Listen werden in den <UL>-Container gepackt. Listenelemente werden in den <LI>-Container gepackt.
3. **Definitionslisten.** Definitionslisten werden haben einen zu definierenden Begriff und eine definierende Beschreibung. Definitionslisten werden in den <DL>-Container gepackt. Der zu definierende Begriff wird in den <DT>-Container gepackt. Die definierende Beschreibung wird in den <DD>-Container gepackt.

20.2.8. Links

Ein Link ist eine Verknüpfung oder Verbindung zu einem anderen Textstück. Zu einem Link gehören immer zwei Auszeichnungen: erstens die Adresse, zu der gesprungen werden soll, und zweitens der Sprungbefehl.

1. **Interne Links.** Ein interner Link ist eine Verbindung zu einem Textstück im selben Dokument. Er wird häufig benutzt, um auf einer Seite an den Anfang oder an das Ende zu springen.
   - Die Adresse, zu der gesprungen werden soll, wird im <A NAME=*Adresse*>-Container verpackt.
   - Der Sprungbefehl wird im <A HREF=#*Adresse*>-Container verpackt.
2. **Externe Links.** Ein externer Link ist eine Verbindung zu einem HTML-Dokument auf dem eigenen oder einem entferneten Rechner.
   - Die Adresse, zu der gesprungen werden soll, kann hier jede URL sein.
   - Der Sprungbefehl wird im <A HREF=*Adresse*>-Container verpackt. (Der Unterschied zum internen Link ist das Fehlen des "Lattenzaunes".)

20.3. CGI (Das Common Gateway Interface)

Für den Datenaustausch zwischen einem Webserver und einem Client, also für das, was passiert, wenn man eine Seite im Internet aufruft, gibt es verschiedene Ansätze. Python benutzt das normierte Verfahren über CGI. Siehe dazu: <http://de.wikipedia.org/wiki/Common_Gateway_Interface> Hier Bemerkungen über Web-Techniken zu machen ist nichts für die Ewigkeit, denn diese ändern sich wahrscheinlich noch schneller als die meisten anderen Bereiche der Informatik.

Trotzdem soll hier ein kurz dargestellt werden, wie Internet-Seiten mit Hilfe des CGI erstellt und dargestellt werden. CGI-Skripte, also Programme, die meistens in einer der p-Sprachen (perl, python) geschrieben werden, sind serverseitige Skripte. Das heißt, dass diese Skripte auf einem Web-Server laufen und das CG-Interface benutzen, um die durch das Programm produzierten Daten mittels eines Browsers auf dem Bildschirm des Benutzers darstellen. CGI selber ist ein Anwendungsprotokoll, das Eingabedaten und Anwendungsergebnisse vom Client zum Server und zurück transportiert.

Die einzelnen Schritte eines solchen Ablaufs sind die folgenden:

- Der Benutzer empfängt eine HTML-Seite, die ein Formular enthält.
- Die Formulardaten, die der Benutzer eingibt, werden an den Server geschickt.
- Ein HTTP-Server (oft der Apache) läuft ständig auf dem Server und horcht an dem zugeordneten Port auf eingehende Signale.
- Der HTTP-Server leitet die eingehenden Daten an das betroffene Programm, in unserem Fall den Python-Interpreter, weiter.
- Aus den eingehenden Daten erstellt ein CGI-Skript eine HTML-Antwortseite. Diese dynamische Seite besteht aus einem Header und HTML-Code.
- Diese Antwortseite wird an den Client gesendet und dort von einem Browser angezeigt.

20.4. Die Pflicht: hallo, ihr alle da draußen

Wie schon weiter oben gesagt, muss man als Programmierer damit anfangen, dass man seine Umwelt grüßt: ”hallo world”. Die dazugehörige HTML-Seite ist schnell geschrieben:

Beispiel 20.4.1 HTML-Seite ”Hallo world”

```
1
2
3   <html>
4     <head>
5       <title>erste HTML-Seite</title>
6     </head>
7     <body>
8       <p>hello world</p>
9     </body>
10  </html>
11
12
```

So weit, so gut, aber das soll ja keine Anleitung für HTML sein, und das ist eine einfache HTML-Seite.

20.5. Hello world als dynamische Web-Seite

Also wird jetzt aus dieser statischen Seite eine dynamische. Eine solche dynamische Web-Seite besteht aus 2 Teilen: dem Header und der eigentlichen Seite. Der Header besteht aus der Zeile ”Content-type: text/html\n”, die so ausgegeben wird:

Beispiel 20.5.1 Header für dynamische HTML-Seiten

```
print "Content-type: text/html\n"
```

Wichtig dabei ist, dass nach dem Header immer eine Zeile freigelassen werden muss. Die ganze dynamische Web-Seite wird jetzt folgendermaßen gebaut. Der gesamte HTML-Block von oben wird als eine Zeichenkette, die sich über mehrere Zeilen erstreckt, in dreifache Anführungsstriche geschrieben und an eine Variable zugewiesen.

Beispiel 20.5.2 Die HTML-Seite als String

```
1
2
3   seite = """
4   <html>
5     <head>
6       <title>erste HTML-Seite</title>
7     </head>
8     <body>
9       <p>hello world</p>
10    </body>
11  </html>  """
12
13
```

Diese Variable wird dann nach der Header-Zeile ausgegeben.

Beispiel 20.5.3 Dynamisches Hello world

```
seite = """
<html>
  <head>
    <title>erste HTML-Seite</title>
  </head>
  <body>
    <p>hello world</p>
  </body>
</html>  """
print "Content-type: text/html\n"

print seite
```

Das war es schon! Leider ist es jetzt nicht für jeden sofort zu überprüfen, ob das auch wirklich wie gewünscht funktioniert, denn nicht jeder hat einen Webserver auf seinem Rechner. Diejenigen, die unter Linux arbeiten, haben wahrscheinlich schon den Apache installiert. Dann muss das oben geschriebene Programm noch gespeichert werden (etwa unter dem Namen `halloCGI.py`), und in das richtige Verzeichnis verschoben werden, meistens `/srv/www/cgi-bin`. Jetzt muss man noch bedenken, dass ein CGI-Skript ein Programm ist, das aufrufbar sein muss. Unter Unix / Linux wird also das Attribut auf 0755 gesetzt werden, und dann kann die Seite mit `http://localhost/halloCGI.py` im Browser aufgerufen werden.

Für diejenigen, die keinen Webserver eingerichtet haben, kommt im nächsten Kapitel ein Mini-Webserver, der in Python geschrieben ist.

20.6. Ein eigener Mini-Webserver

Dieser Webserver ist übernommen aus M. Lutz, [Programming Python]. Wie Mark Lutz schreibt, erhebt er keinen Anspruch auf Komfort und Sicherheit, aber er tut es, um auf seinem eigenen Rechner etwas schnell zu testen. In der Zeile mit `webdir = '???'` müssen natürlich die Fragezeichen ersetzt werden durch das Verzeichnis, in dem die CGI-Skripte gespeichert werden. Damit ist man aber auf der anderen Seite unabhängig von den Standard-Vorgaben über die Speicherung von CGI-Skripten und kann diese in einem beliebigen Verzeichnis (auch im eigenen home-Verzeichnis) speichern. In der darauffolgenden Zeile ist der Standard-Port für HTTP eingetragen, aber auch der kann geändert werden, falls das benötigt oder gewünscht ist.

Beispiel 20.6.1 Ein Web-Server in Python

```
1
2   #!/usr/bin/python
3   import os, sys
4   from BaseHTTPServer import HTTPServer
5   from CGIHTTPServer import CGIHTTPRequestHandler
6
7   webdir = '?????'  ## webdir = './wwwDateien'
8   port = 80         ## oder ein anderer Port
9
10  if sys.platform[:3]  == 'win':
11    CGIHTTPRequestHandler.have_popen2 = False
12    CGIHTTPRequestHandler.have_popen3 = False
13    sys.path.append('cgi-bin')
14
15  os.chdir(webdir)
16  srvaddr = ("", port)
17  srvobj = HTTPServer(srvaddr,  CGIHTTPRequestHandler)
18  srvobj.serve_forever()
19
```

So ein schön kurzes Programm, aber es reicht wirklich als Webserver für den Hausgebrauch.

20.7. Dynamik durch Schleifen

In diesem Teil sollen jetzt erste dynamische Seiten erstellt werden. Dazu lassen wir zuerst einmal eine Variable eine Schleife durchlaufen, vorwärts und rückwärts, und dann bringen wir das kleine Einmaleins auf eine HTML-Seite.

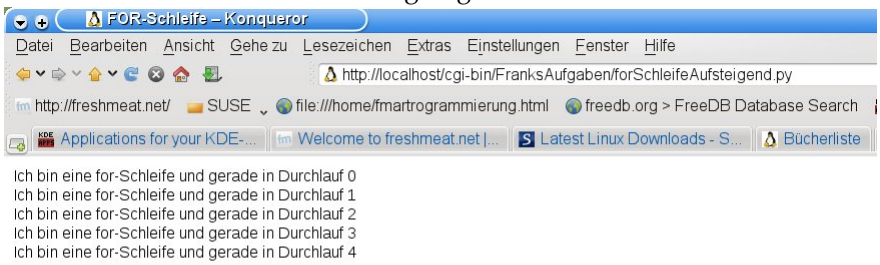
20.7.1. Vorwärtszählen ..

Wie im vorigen Beispiel wird zuerst der HTML-Code aufgebaut. Dieser wird in dreifache Anführungsstriche geschrieben. An der Stelle, wo durch Python dynamischer Inhalt eingefügt werden soll, wird der Platzhalter %s geschrieben.

Beispiel 20.7.1 For-Schleife in einer dynamischen HTML-Seite

```
#!/usr/bin/python

seite = '''
<html>
  <head>
    <title>FOR-Schleife</title>
  </head>
  <body>
    %s
  </body>
</html>'''

zeile = "Ich bin eine for-Schleife und gerade in Durchlauf %d<br/>"
text = ''

for i in range(5):
  text += zeile % i

print "Content-type: text/html\n\n"

print seite % text
```

In der letzten Zeile wird die Seite ausgegeben, wobei durch den Prozentoperator der Inhalt der Variablen `text` in die Variable `seite` eingefügt wird.

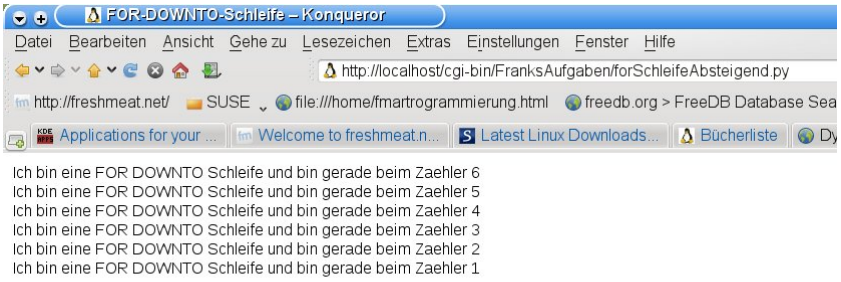


20.7.2. .. und rückwärtszählen

Wie man sofort sieht, ändert sich nur eine Zeile, nämlich die Schleifenbedingung für das Zählen. Während vorher nur ein Parameter an die Schleife mitgegeben wurde, nämlich das Ende der Schleife (das ist dort oben bei 5), werden jetzt drei Parameter mitgegeben: der Start (6), das Ende (0) und die Schrittweite (-1). Das ist alles.

Beispiel 20.7.2 For-DOWNTO-Schleife in einer dynamischen HTML-Seite

```
#!/usr/bin/python

seite = '''
<html>
  <head>
    <title>FOR-DOWNTO-Schleife</title>
  </head>
  <body>
    %s
  </body>
</html>
'''

zeile = "Ich bin eine FOR DOWNTO Schleife und bin gerade beim Zaehler %d<br/>"
text = ''


for i in range(6, 0, -1):
  text += zeile % i

print "Content-type: text/html\n\n"

print seite % text
```



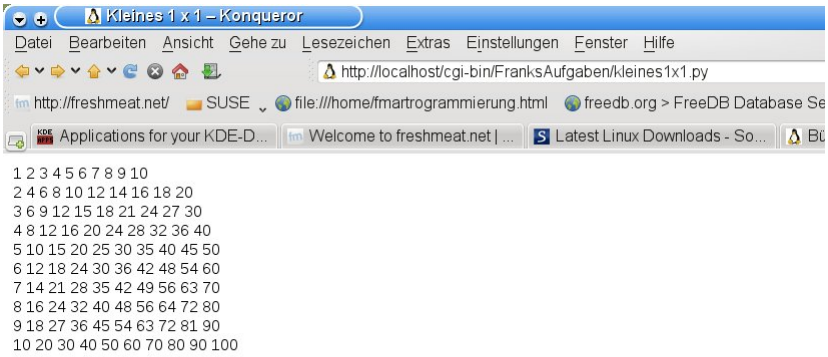
20.7.3. Das kleine 1 x 1

Jetzt folgt als nächstes Beispiel eine HTML-Seite, die das kleine Einmaleins anzeigt. Hier ist es sinnvoll, eine Funktion zu schreiben, die das erledigt. Im ersten Anlauf wird jeweils eine Zeile generiert, die Zeile mit einem Zeilenvorschub beendet, und dann die nächste Zeile erstellt. Die einzelnen Zeilen werden aneinandergehängt, und der erzeugte String wieder mit Hilfe der Formatierung von Zeichenketten in den HTML-Code eingefügt.

Beispiel 20.7.3 Das kleine 1 mal 1 auf einer dynamischen HTML-Seite

```
#!/usr/bin/python

headerZeile = "Content-type: text/html\n\n"

seite = '''
<html>
  <head>
    <title>Kleines 1 x 1</title>
  </head>
  <body>
  %s
  </body>
</html>
'''

def klEinMalEins():
  klEinMalEinsString = ''                    # eine leere Zeichenkette wird
                                             # bereit gestellt
  for z in range(1, 11):
    for s in range(1, 11):
      klEinMalEinsString += str(z*s) + ' '   # die nächste Zahl wird nach einem
                                             # Leerzeichen eingefügt
    klEinMalEinsString += '<br/>'            # die Zeile wird mit einem
                                             # Zeilenvorschub beendet
  return klEinMalEinsString                  # die Zeichenkette wird zurück-
                                             # gegeben
                                             # Ende der Funktion
print headerZeile                            # Header ausgeben

print seite % klEinMalEins()                 # Seite ausgeben
```

Das funktioniert. Aber es sieht wirklich nicht schön aus, das muss man wohl zugeben.



Die Verbesserung ist, dass man das kleine Einmaleins jetzt in einer HTML-Tabelle ausgibt. Die Variable `seite` sieht fast genau so aus wie bisher, es sind nur die Tags für die Tabelle dazugekommen. Die Funktion `klEinMalEins` hat sich etwas mehr geändert. Die Zeile, die jetzt generiert wird, beginnt und

endet mit den Tags für eine Tabellenzeile, jeder einzelne Eintrag bekommt die Table-Data-Tags, versehen mit der Option rechtsbündig.

Beispiel 20.7.4 Das kleine 1 mal 1 in einer Tabelle (dynamische HTML-Seite)

```
#!/usr/bin/python

headerZeile = "Content-type: text/html\n\n"

seite = '''
<html>
  <head>
    <title>Kleines 1 x 1</title>
  </head>
  <body>
    <table border='1'>
    %s
    </table>
  </body>
</html>
'''

def klEinMalEins():
  klEinMalEinsString = ''
  for z in range(1, 11):
    klEinMalEinsString += '<tr>'
    for s in range(1, 11):
      klEinMalEinsString += '<td align="right">'+str(z*s)+ '</td>'
    klEinMalEinsString += '</tr>'
  return klEinMalEinsString

print headerZeile

print seite % klEinMalEins()
```

Und das sieht schon besser aus.

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

20.8. Formular-Eingabe und Auswertung

Jeder nur ein Kreuz!

—Monty Python[2]

In diesem Abschnitt soll jetzt das Zusammenspiel von HTML-Formularen und auswertenden Python-Programmen bearbeitet werden. Aus diesem Grund wird zuerst einmal ein HTML-Formular entworfen, in das Vorname, Nachname und Betrieb des Benutzers eingegeben werden können.

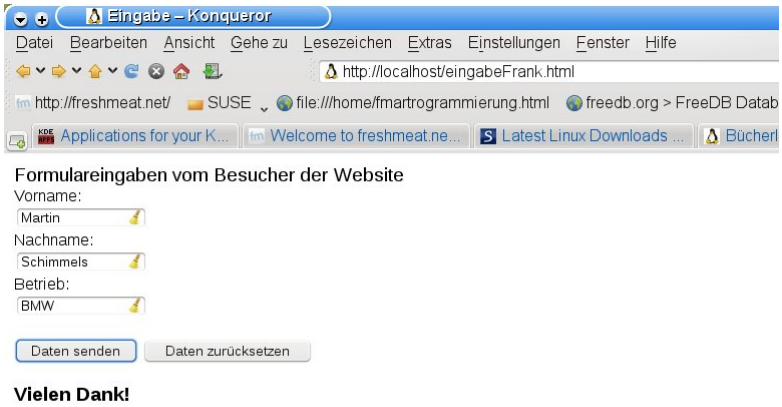
Beispiel 20.8.1 Eingabeformular in HTML

```
1
2
3   <html>
4     <head>
5       <title>Eingabe</title>
6     </head>
7     <body>
8
9       <form action="./cgi-bin/FranksAufgaben/auswerten.py" method="POST">
10      <font size="4">Formulareingaben vom Besucher der Website</font>
11      <br>
12
13      Vorname:<br>
14      <input type="text" name="vorname"><br>
15      Nachname:<br>
16      <input type="text" name="nachname"><br>
17      Betrieb:<br>
18      <input type="text" name="betrieb"><br>
19      <br>
20
21      <input type="Submit" value="Daten senden">
22      <input type="Reset" value="Daten zur&uuml;cksetzen">
23      <br>
24      <br><b><font size="4">Vielen Dank!</font></b><br>
25
26      </form>
27    </body>
28  </html>
29
30
```

Das sieht dann so aus:

[2]Das Leben des Brian

Abbildung 20.1. Eingabe-Formular in HTML(Namen)



Die HTML-Tags sollen hier im einzelnen nicht durchgesprochen werden. Erwähnenswert ist hier nur die Zeile mit `form action`. Hier muss der Dateiname des Python-Skriptes angegeben werden, das die Eingabe auswertet. Daten aus Formularen können mit 2 Methoden weitergegeben werden: mittels `post` oder mittels `get`. Der wesentliche Unterschied ist der, dass die mit `post` geschickten Daten in einem gesonderten Paket an den Server geschickt werden, während bei der Methode `get` die Daten an die URL nach einem Fragezeichen angehängt werden. Das kann man nachher ganz schnell testen, wenn das Formular und das zugehörige Programm erst einmal funktionieren.

Kommen wir also zum auswertenden Programm.

Beispiel 20.8.2 Auswertung des Eingabeformulars(Namen)

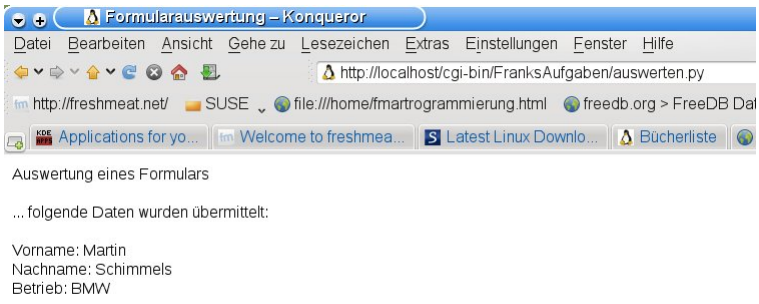
```
#!/usr/bin/python
# -*- coding: utf8 -*-

import cgi
formularFelder = cgi.FieldStorage()

seite = '''
<html>
  <head>
    <title>Formularauswertung</title>
  </head>
  <body>
    Auswertung eines Formulars
    <br>
    <br>
    %s
  </body>
</html>'''

zeilen = ' ... folgende Daten wurden übermittelt:<br><br>'
zeilen += 'Vorname:  '+formularFelder['vorname'].value+'<br>'
zeilen += 'Nachname: ' +formularFelder['nachname'].value+'<br>'
zeilen += 'Betrieb:  '+formularFelder['betrieb'].value+'<br>'

print "Content-type: text/html\n\n"
print seite % zeilen
```

Die wichtigen Zeilen sind die beiden ersten. Um Formulareingaben zu verarbeiten, muss die Bibliothek `cgi` eingebunden werden. Diese Bibliothek enthält die Klasse `FieldStorage`, die alle übergebenen Daten enthält, egal wie sie übergeben wurden.

Danach wird wieder eine Seite in HTML aufgebaut, und diese wird in der Variablen `seite` gespeichert. Der von Python generierte dynamische Code beginnt mit der Zuweisung von Text an die Variable `zeilen`. Danach wird mittels des Kurzschluss-Additions-Operators `+=` Text angehängt, der aus den übertragenen Daten aus dem Formular bestehen.

Die Klasse cgi.FieldStorage enthält die Daten in Form eines Dictionary. Das heißt im vorliegenden Fall, dass die Variable `formularFelder` so aufgebaut ist:

```
formularFelder = {'vorname':'Martin', 'nachname':'Schimmels', 'betrieb':'BMW'}
```



Es ist jetzt ganz einfach, die Methode `get` bei der Übertragung von Daten zu testen. Es muss nur in der Zeile mit `form action` im HTML-Formular das `post` in ein `get` geändert werden. Das verarbeitende Python-Programm muss nicht geändert werden, weil die Methoden des Moduls `cgi` die ganze Arbeit erledigen. Das Modul erkennt, in welcher Form die Daten übermittelt werden und wertet alles richtig

aus. Den Unterschied sieht man aber in der Adresszeile des Browsers. Bei Benutzung der Post-Methode sieht diese Zeile so aus:

```
http://localhost/cgi-bin/FranksAufgaben/auswerten.py
```

bei Benutzung der Get-Methode so:

```
http://localhost/cgi-bin/FranksAufgaben/auswerten.py?vorname=Martin
&nachname=Schimmels&betrieb=BMW+Dingolfing
```

20.9. Eine Rechnung

Im nächsten Beispiel soll eine Rechnung für den Eintritt in eine Sportveranstaltung geschrieben werden. Es gibt einen Normaltarif, dazu allerdings Ermäßigungen für Kinder, Studenten und Senioren. In ein HTML-Formular soll die Anzahl Erwachsener, die Anzahl Kinder, Senioren und Studenten eingegeben werden. Ein Python-Programm berechnet den Rechnungsbetrag, der dann auf der nächsten HTML-Seite ausgegeben wird.

Das Eingabeformular könnte so aussehen:

Abbildung 20.2. Eingabe-Formular für eine Rechnung in HTML

Der dazugehörende HTML-Code ist im nächsten Bild zu sehen:

Beispiel 20.9.1 Eingabeformular für Rechnung

```
1
2
3   <html>
4
5     <head>
6       <title>Rechnung Eintritt Sport</title>
7       <meta name="generator" content="Bluefish 2.2.3" >
8       <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
9     </head>
10    <body>
11      <h1>Eintrittskarten für Sportveranstaltung</h1>
12      <p>
13        <form action="./cgi-bin/FranksAufgaben/rechnSchreiben.py" method="POST">
14          <br>
15          <table>
16            <tr>
17              <td>Anzahl Erwachsene:</td>
18              <td><input type="text" name="erw"></td>
19            </tr>
20            <tr>
21              <td>Anzahl Kinder:</td>
22              <td><input type="text" name="kind"></td>
23            </tr>
24            <tr>
25              <td>Anzahl Senioren:</td>
26              <td><input type="text" name="sen"></td>
27            </tr>
28            <tr>
29              <td>Anzahl Studenten:</td>
30              <td><input type="text" name="stud"></td>
31            </tr>
32          </table>
33
34          <input type="Submit" value="Daten senden">
35          <input type="Reset" value="Daten zur&uuml;cksetzen">
36          <br>
37          <br><b><font size="4">Vielen Dank!</font></b><br>
38
39        </form>
40      </p>
41
42    </body>
43  </html>
44
45
```

Für die Berechnung des Rechnungsbetrags erstellen wir eine Klasse `Rechnung`. Der Konstruktor der Klasse nimmt die Anzahl der Erwachsenen, Kinder, Senioren und Studenten in Empfang. Die Klasse hat die Methode `rechnungSchreiben`, zusätzlich zum Testen auch noch die Methode `rechnungAnzeigen`.

Um den Test durchzuführen, wird wieder einmal mittels des Konstrukts `if __name__ == '__main__':` der Test in die Klassen-Datei eingefügt.

Beispiel 20.9.2 Klasse Rechnung

```
1
2   #!/usr/bin/python
3
4   class Rechnung():
5     def __init__(self,  anzErw=0,  anzKids=0,  anzSen=0,  anzStud=0):
6       self.anzErw = anzErw
7       self.anzKids = anzKids
8       self.anzSen = anzSen
9       self.anzStud = anzStud
10      self.normalPreis = 10.0
11      self.ermKids = 0.5
12      self.ermStud = 0.3
13      self.ermSen = 0.4
14
15    def rechnungSchreiben(self):
16      self.rechnBetrag = self.anzErw * self.normalPreis
17                                     + self.anzKids * self.normalPreis*self.ermKids
18                                     + self.anzSen*self.normalPreis*self.ermSen
19                                     + self.anzStud*self.normalPreis*self.ermStud
20
21    def rechnungAnzeigen(self):
22      print 'Rechnungsbetrag: '+ str(self.rechnBetrag)
23
24  if __name__ == '__main__':
25    meineRechnung = Rechnung(3, 2, 4, 1)
26    meineRechnung.rechnungSchreiben()
27    meineRechnung.rechnungAnzeigen()
28
```

Nachdem der Test erfolgreich war, wird das Skript, das die Auswertung durchführt, geschrieben. Das sieht so aus:

Beispiel 20.9.3 Auswertung des Eingabeformulars Rechnung

```
1
2
3   #!/usr/bin/python
4
5   import cgi
6   formularFelder = cgi.FieldStorage()
7
8   seite = '''
9     <html>
10      <head>
11        <title>Rechnung Sportveranstaltung</title>
12      </head>
13      <body>
14        <h1>Rechnung</h1>
15        <br>
16        <br>
17        %s
18      </body>
19    </html>'''
20
21
22  from clRechnungEinfach import Rechnung
23  try:
24    erw = int(formularFelder['erw'].value)
25  except:
26    erw = 0
27  try:
28    kind = int(formularFelder['kind'].value)
29  except:
30    kind = 0
31  try:
32    sen = int(formularFelder['sen'].value)
33  except:
34    sen = 0
35  try:
36    stud = int(formularFelder['stud'].value)
37  except:
38    stud = 0
39
40  neueRechnung = Rechnung(erw,  kind,  sen,  stud)
41  neueRechnung.rechnungSchreiben()
42
43  print "Content-type: text/html\n\n"
44
45  print seite % str(neueRechnung.rechnBetrag)
46
47
48
```

Zu beachten in diesem Stück Code sind die 4 `try - except`-Anweisungen. Damit wird gesichert, dass das Skript auch dann korrekt arbeitet, wenn im Formular ein Feld nicht gefüllt wird. In diesem Fall wird angenommen, dass für diese Gruppe Menschen der Wert „0“ gemeint ist.

Abbildung 20.3. Ausgabe einer Rechnung in HTML (aber wirklich nicht schön)

Da die Ausgabe des Programms wirklich nicht sehr schön ist, denn mit der alleinigen Angabe des zu zahlenden Betrags kann man nicht zufrieden sein, wird die Klasse `Rechnung` verbessert. Die Veränderung findet in der Methode `rechnungSchreiben` statt. Von der Logik ändert sich nichts, es handelt sich hier nur um Änderungen in der Darstellung durch HTML. Die Ausgabe wird hier in eine Tabelle gemacht.

Beispiel 20.9.4 Klasse Rechnung (verbessert)

```
1
2
3   #!/usr/bin/python
4
5   class Rechnung():
6     def __init__(self,  anzErw=0,  anzKids=0,  anzSen=0,  anzStud=0):
7       self.anzErw = anzErw
8       self.anzKids = anzKids
9       self.anzSen = anzSen
10      self.anzStud = anzStud
11      self.normalPreis = 10.0
12      self.ermKids = 0.5
13      self.ermStud = 0.3
14      self.ermSen = 0.4
15
16    def rechnungSchreiben(self):
17      self.rechnBetrag = self.anzErw * self.normalPreis
18                                 + self.anzKids * self.normalPreis*self.ermKids
19                                 + self.anzSen*self.normalPreis*self.ermSen
20                                 + self.anzStud*self.normalPreis*self.ermStud
21      self.rechnStringHTML = '<table border="1">'
22      # Zeile f. Erwachsene
23      self.rechnStringHTML += '<tr>'
24      self.rechnStringHTML += '<td>Erwachsene:</td>
25                    <td align="right">'+str(self.anzErw)+'</td>'
26      t1 = '%4.2f' % self.normalPreis
27      self.rechnStringHTML += '<td align="right">'+str(t1)+' Euro</td>'
28      t2 = '%4.2f' % (self.normalPreis * self.anzErw)
29      self.rechnStringHTML += '<td align="right">'+str(t2)+' Euro</td>'
30      self.rechnStringHTML += '</tr>'
31      # Zeile f. Kinder
32      self.rechnStringHTML += '<tr>'
33      self.rechnStringHTML += '<td>Kinder:</td>
34                    <td align="right">'+str(self.anzKids)+'</td>'
35      t1 = '%4.2f' % (self.normalPreis * self.ermKids)
36      self.rechnStringHTML += '<td align="right">'+str(t1)+' Euro</td>'
37      t2 = '%4.2f' % (self.normalPreis * self.ermKids * self.anzKids)
38      self.rechnStringHTML += '<td align="right">'+str(t2)+' Euro</td>'
39      self.rechnStringHTML += '</tr>'
40
41
```

Beispiel 20.9.5 Klasse Rechnung (verbessert) (Forts.)

```
1
2
3       # Zeile f. Senioren
4       self.rechnStringHTML += '<tr>'
5       self.rechnStringHTML += '<td>Senioren:</td>
6                    <td align="right">'+str(self.anzSen)+'</td>'
7       t1 = '%4.2f' % (self.normalPreis * self.ermSen)
8       self.rechnStringHTML += '<td align="right">'+str(t1)+' Euro</td>'
9       t2 = '%4.2f' % (self.normalPreis * self.ermSen * self.anzSen)
10      self.rechnStringHTML += '<td align="right">'+str(t2)+' Euro</td>'
11      self.rechnStringHTML += '</tr>'
12      # Zeile f. Studis
13      self.rechnStringHTML += '<tr>'
14      self.rechnStringHTML += '<td>Studenten:</td>
15                   <td align="right">'+str(self.anzStud)+'</td>'
16      t1 = '%4.2f' % (self.normalPreis * self.ermStud)
17      self.rechnStringHTML += '<td align="right">'+str(t1)+' Euro</td>'
18      t2 = '%4.2f' % (self.normalPreis * self.ermStud * self.anzStud)
19      self.rechnStringHTML += '<td align="right">'+str(t2)+' Euro</td>'
20      self.rechnStringHTML += '</tr>'
21      # Summen-Zeile
22      self.rechnStringHTML += '<tr> <td colspan="3">Summe:</td> <td>'
23      t1 = '%4.2f' % (self.rechnBetrag)
24      self.rechnStringHTML += str(t1)+' Euro'+ '</td> </tr>'
25      # Ende
26      self.rechnStringHTML += '</table>'
27
28    def rechnungAnzeigen(self):
29      print 'Rechnungsbetrag: '+ str(self.rechnBetrag)
30
31  if __name__ == '__main__':
32    meineRechnung = Rechnung(3, 2, 4, 1)
33    meineRechnung.rechnungSchreiben()
34    meineRechnung.rechnungAnzeigen()
35    print meineRechnung.rechnStringHTML
36
37
```

In dem CGI-Skript muss dann nur die letzte Zeile ersetzt werden durch

Beispiel 20.9.6 Verbesserung im auswertenden Programm

```
print seite % str(neueRechnung.rechnStringHTML)
```

Abbildung 20.4. Ausgabe einer Rechnung in HTML (so kann man das akzeptieren)

20.10. Jetzt aber wirkliche Dynamik

Eine typische Anwendung für eine dynamische Web-Seite ist ein Besucherzähler. Jedesmal, wenn eine bestimmte Seite aufgerufen wird, wird der Zähler um eins erhöht und dann angezeigt mit einer freundlichen Bemerkung wie etwa: Du bist der 35812. Besucher dieser Seite.

Zuerst brauchen wir also eine Klasse `Zaehler`. Diese Klasse hat zwei Attribute, nämlich einen Initial-Wert und den eigentlichen Wert, den der Zähler zu einer bestimmten Zeit innehat. Der aktuelle Wert wird in einer einfachen Textdatei gespeichert, weswegen von Hand eine Datei `zaehlerDat` angelegt werden muss, in der einfach eine 0 steht. Der Konstruktor liest die Datei und weist den Inhalt dem Attribut `zWert` zu.

Die Klasse `Zaehler` benötigt dann noch 3 Methoden, nämlich `inkrementieren`, `dekrementieren` und `wertZuruecksetzen` (die braucht man nicht für den Besucherzähler, aber vielleicht für andere Dinge). Jede dieser Methoden besteht aus 4 Teilen, dem Öffnen der Textdatei im (Über-)Schreib-Modus, der Aktion selber, die `zWert` verändert, dem Schreiben in die Datei und dem Schließen der Datei.

Beispiel 20.10.1 Die Klasse Zähler

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4   class Zaehler(object):
5     def __init__(self):
6       self.dateiOeffnen('r')
7       self.zWert = int(self.meineDat.read())
8       self.initWert = 0
9       self.dateiSchliessen()
10
11    def dateiOeffnen(self, modus):
12      self.meineDat = open("zaehlerDat",modus)
13
14    def dateiSchreiben(self, z):
15      self.meineDat.write(str(z))
16
17    def dateiSchliessen(self):
18      self.meineDat.close()
19
20    def wertZuruecksetzen(self):
21      self.dateiOeffnen('w')
22      self.zWert = self.initWert
23      self.dateiSchreiben(self.zWert)
24      self.dateiSchliessen()
25
26    def inkrement(self):
27      self.dateiOeffnen('w')
28      self.zWert = self.zWert + 1
29      self.dateiSchreiben(self.zWert)
30      self.dateiSchliessen()
31
32    def dekrement(self):
33      self.dateiOeffnen('w')
34      self.zWert = self.zWert - 1
35      self.dateiSchreiben(self.zWert)
36      self.dateiSchliessen()
37
```

Man kann sich natürlich jetzt ein kleines Programm schreiben, das diese Klasse benützt – und das sollte man auch tun, um zu testen, ob alles so funktioniert, wie man sich das vorstellt.

Wenn man das gemacht hat und sicher ist, dass alles wie gewünscht tut, schreibt man sich die HTML-Seite, die den Besucherzähler aufnehmen soll. An der Stelle, wo der Besucherzähler nachher tatsächlich arbeiten soll, stehen im Moment noch 3 Fragezeichen.

Beispiel 20.10.2 Der Besucherzähler (HTML-Teil)

```
<html>
  <head>
    <title>Martins Test</title>
    <meta http-equiv="Content-type" content="text/html; charset=latin1" />
  </head>
  <body>
    <h1>Ein Besucherzähler</h1>
    <p>Hallo Du da!!</p>
    <p>Du bist der ???. Besucher</p>
  </body>
</html>
```

Diese ganzer HTML-Code wird, wie schon weiter oben, in dreifache Anführungszeichen gesetzt. Statt der 3 Fragezeichen wird jetzt ein Python-Platzhalter, das

%s

, geschrieben. Die Header-Zeile wird geschrieben und es wird die Klasse `Zaehler` importiert. Eine Instanz von `Zaehler` wird erzeugt und der Wert erhöht. Dann wird die vorher in Anführungszeichen geschriebene Seite geschrieben, wobei der aktuelle Wert des Zählers über die String-Ersetzung des

%s

vorgenommen wird. (Falls Du nicht mehr weißt, wie das mit der Formatierung funktioniert, hier ist der Link zu dieser Seite.)

Beispiel 20.10.3 Der Besucherzähler (vollständig)

```
#!/usr/bin/python

seite = '''
<html>
  <head>
    <title>Martins Test</title>
    <meta http-equiv="Content-type" content="text/html; charset=latin1" />
  </head>
  <body>
    <h1>Ein Besucherzähler</h1>
    <p>Hallo Du da!!</p>
    <p>Du bist der %s . Besucher</p>
  </body>
</html>
'''

print "Content-type: text/html\n\n"

from cl_zaehler import Zaehler
besucherZaehler = Zaehler()
besucherZaehler.inkrement()

print seite % besucherZaehler.zWert
```

21. Eine Kreuzung auf meiner Web-Seite

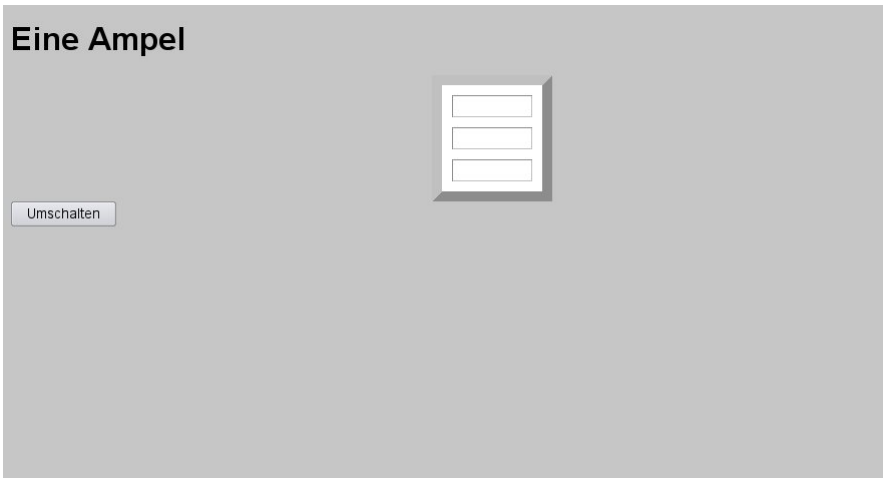
21.1. Die Ampel kommt ins Netz

Weiter oben bei [Ampel auf der Konsole] haben wir eine Ampel und eine Kreuzung realisiert. Nun soll daraus eine dynamische WEeb-Seite werden. In einem ersten Schritt erstellen wir eine statische Web-Seite, die in einer Tabelle die Ampel darstellt. Der HTML-Code dazu sieht so aus:

Beispiel 21.1.1 HTML-Code für eine Ampel

```
1
2
3  <html>
4    <head>
5      <title>Eine Ampel</title>
6      <meta name="generator" content="Bluefish 2.2.3" />
7      <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
8    </head>
9    <body bgcolor="#C6C6C6">
10     <h1>Eine Ampel</h1>
11     <table width="120" cellspacing="10" border="10" cellpadding="10"
12                 align="center" bgcolor="white">
13       <tbody>
14         <tr>
15           <td></td>
16         </tr>
17         <tr>
18           <td></td>
19         </tr>
20         <tr>
21           <td></td>
22         </tr>
23       </tbody>
24     </table>
25   </body>
26 </html>
27
```

Im Browser sieht diese Seite so aus:



Der HTML-Code wird jetzt in eine Variable `seite` eingebunden. In den 3 Lichtern der Ampel wird als Attribut die Hintergrundfarbe eingefügt, wobei der Wert der Hintergrundfarbe über den %s-Operator durch den folgenden Python-Code eingefügt wird. (Das kleine `o` im Tabellen-Element dient dazu, die Ampel etwas größer zu machen.)

Der Python-Code bindet zuerst die Klasse `Ampel` ein, sodann wird ein Objekt dieser Klasse erzeugt und die Methode `umschalten` dieses Objekts aufgerufen. Nachdem die Lichter der Ampel umgeschaltet wurden, werden die Farben der Lichter in einen Tupel geschrieben, der mit dem %-Operator in die Variable `seite` eingefügt wird.

Beispiel 21.1.2

```
1
2
3  #!/usr/bin/python
4
5  seite = '''
6  <html>
7  <head>
8    <title>Eine Ampel</title>
9  </head>
10 <body bgcolor="#C6C6C6">
11    <h1>Eine Ampel</h1>
12    <table width="120" cellspacing="10" border="10" cellpadding="10"
13                  align="center" bgcolor="white">
14   <tbody>
15     <tr>
16       <td bgcolor="%s">o</td>
17     </tr>
18     <tr>
19       <td bgcolor="%s">o</td>
20     </tr>
21     <tr>
22       <td bgcolor="%s">o</td>
23     </tr>
24   </tbody>
25 </table>
26 <FORM action="ampelHTML.py" method="POST">
27   <INPUT type="submit" name="umschalten" value="Umschalten" size="120">
28 </FORM>
29 </body>
30 </html>
31 '''
32
33 from clAmpel import Ampel
34
35 meineAmpel = Ampel()
36 meineAmpel.umschalten()
37
38 headerZeile = "Content-type: text/html\n\n"
39
40 print headerZeile
41 print seite %
42      tuple(meineAmpel.zustaende[meineAmpel.zustaendeNum[meineAmpel.zustand]])
43
44
```

21.2. Die Kreuzung im Netz ist auch nicht schwerer

Die Kreuzung ist eine 3x3 - Tabelle, bei der in die mittleren Zellen der äußeren Zeilen und Spalten jeweils eine 3x1 - Tabelle die Ampel abbildet.

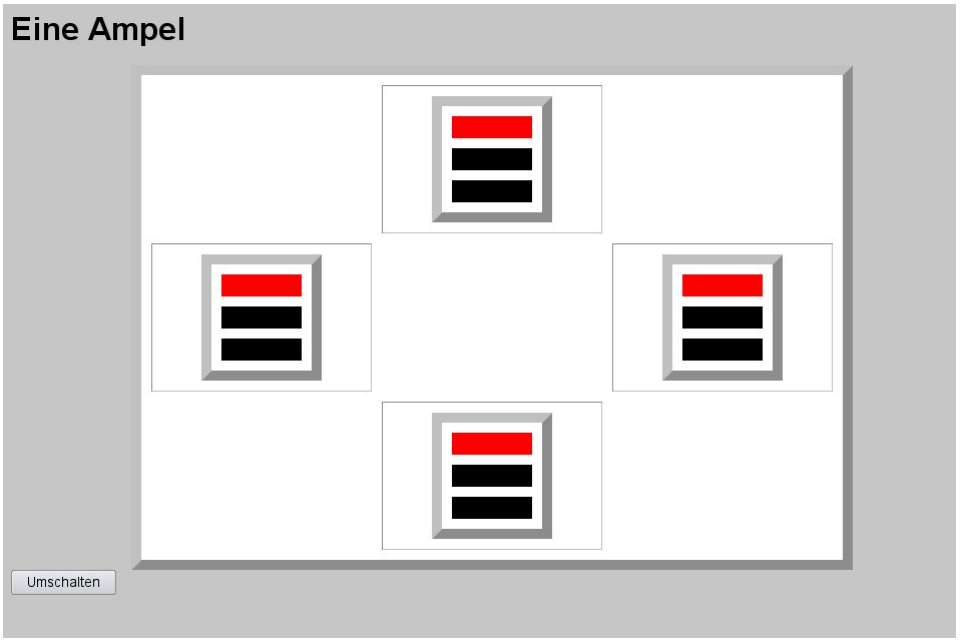
Beispiel 21.2.1 HTML und Python für die Kreuzung

```
1
2
3  <html>
4    <head>
5      <title>Eine Ampel</title>
6      <meta name="generator" content="Bluefish 2.2.3" >
7      <meta http-equiv="Content-Type" content="text/html">
8    </head>
9    <body bgcolor="#C6C6C6">
10     <h1>Eine Kreuzung</h1>
11     <table width="720" cellspacing="10" border="10" cellpadding="10"
12             align="center" bgcolor="white">
13       <tr>
14         <td></td>
15         <td>
16           <table width="120" cellspacing="10" border="10" cellpadding="10"
17             align="center" bgcolor="white">
18             <tbody>
19               <tr>
20                 <td bgcolor="#FF0000"></td>
21               </tr>
22               <tr>
23                 <td bgcolor="#000000"></td>
24               </tr>
25               <tr>
26                 <td bgcolor="#000000"></td>
27               </tr>
28             </tbody>
29           </table>
30         </td>
31         <td></td>
32       </tr>
33       <tr>
34         <td>
35           <table width="120" cellspacing="10" border="10" cellpadding="10"
36             align="center" bgcolor="white">
37             <tbody>
38               <tr>
39                 <td bgcolor="#FF0000"></td>
40               </tr>
41               <tr>
42                 <td bgcolor="#000000"></td>
43               </tr>
44               <tr>
45                 <td bgcolor="#000000"></td>
46               </tr>
47             </tbody>
48           </table>
49         </td>
50
```

Beispiel 21.2.2 HTML und Python für die Kreuzung (Forts.)

```
        <td></td>
        <td>
          <table width="120" cellspacing="10" border="10" cellpadding="10"
            align="center" bgcolor="white">
            <tbody>
              <tr>
                <td bgcolor="#FF0000"></td>
              </tr>
              <tr>
                <td bgcolor="#000000"></td>
              </tr>
              <tr>
                <td bgcolor="#000000"></td>
              </tr>
            </tbody>
          </table>
        </td>
      </tr>
      <tr>
        <td></td>
        <td>
          <table width="120" cellspacing="10" border="10" cellpadding="10"
            align="center" bgcolor="white">
            <tbody>
              <tr>
                <td bgcolor="#FF0000"></td>
              </tr>
              <tr>
                <td bgcolor="#000000"></td>
              </tr>
              <tr>
                <td bgcolor="#000000"></td>
              </tr>
            </tbody>
          </table>
        </td>
        <td></td>
      </tr>
    </table>

  <FORM action="ampelHTML.py" method="POST">
    <INPUT type="submit" name="umschalten" value="Umschalten" size="120">
  </FORM>
  </body>
</html>

```

Im Browser sieht das so aus:



Der Code für die dynamische Web-Seite ist nach dem selben Schema gebaut wie der Code für die dynamische Web-Ampel. Der HTML-Code wird in die Variable `seite` eingebunden. In den 3 Lichtern der 4 Ampeln wird als Attribut die Hintergrundfarbe eingefügt, wobei der Wert der Hintergrundfarbe über den %s-Operator durch den folgenden Python-Code eingefügt wird. (Das kleine `o` im Tabellen-Element dient dazu, die Ampel etwas größer zu machen.)

Der Python-Code bindet zuerst die Klasse `clKreuzung` ein, sodann wird ein Objekt dieser Klasse erzeugt und die Methode `umschalten` dieses Objekts aufgerufen. Nachdem die Lichter der Ampel umgeschaltet wurden, wird ein Tupel aufgebaut, der von allen 4 Ampeln die Farben der einzelnen Lichter enthält, insgesamt also 12 Werte. Das ist der einzige etwas lästige Schritt in diesem Programm. Denn diese 12 Werte werden zuerst in eine Liste, die `zustandsListe`, geschrieben, die dann beim Einfügen in die `seite` in ein Tupel umgewandelt wird.

Beispiel 21.2.3 Die Kreuzung

```
#!/usr/bin/python

seite = '''
<html>
  <head>
    <title>Eine Kreuzung</title>
  </head>
  <body bgcolor="#C6C6C6">
    <h1>Eine Kreuzung</h1>
    <table width="720" cellspacing="10" border="10" cellpadding="10"
            align="center" bgcolor="white">
      <tbody>
      <tr>
        <td></td>
        <td>
          <table width="120" cellspacing="10" border="10" cellpadding="10"
            align="center" bgcolor="white">
            <tbody>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
            </tbody>
          </table>
        </td>
        <td></td>
      </tr>
      <tr>
        <td>
          <table width="120" cellspacing="10" border="10" cellpadding="10"
            align="center" bgcolor="white">
            <tbody>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
            </tbody>
          </table>
        </td>

```

Beispiel 21.2.4 Die Kreuzung (Forts.)

```
        <td></td>
        <td>
          <table width="120" cellspacing="10" border="10" cellpadding="10"
                      align="center" bgcolor="white">
            <tbody>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
            </tbody>
          </table>
        </td>
      </tr>
      <tr>
        <td></td>
        <td>
          <table width="120" cellspacing="10" border="10" cellpadding="10"
                     align="center" bgcolor="white">
            <tbody>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
              <tr>
                <td bgcolor="%s">o</td>
              </tr>
            </tbody>
          </table>
        </td>
        <td></td>
      </tr>
      </tbody>
    </table>


  <FORM action="kreuzungHTML.py" method="POST">
    <INPUT type="submit" name="umschalten" value="Umschalten" size="120">
  </FORM>
  </body>
</html>
'''
```

Beispiel 21.2.5 Die Kreuzung (2. Forts.)

```
1
2
3
4  from clKreuzung import Kreuzung
5
6  meineKreuzung = Kreuzung()
7  meineKreuzung.umschalten()
8
9  zustandsListe = []
10 for einZustand in meineKreuzung.nordAmpel.zustaende
11       [meineKreuzung.nordAmpel.zustaendeNum[meineKreuzung.nordAmpel.zustand]]:
12   zustandsListe.append(einZustand)
13 for einZustand in meineKreuzung.westAmpel.zustaende
14       [meineKreuzung.westAmpel.zustaendeNum[meineKreuzung.westAmpel.zustand]]:
15   zustandsListe.append(einZustand)
16 for einZustand in meineKreuzung.ostAmpel.zustaende
17       [meineKreuzung.ostAmpel.zustaendeNum[meineKreuzung.ostAmpel.zustand]]:
18   zustandsListe.append(einZustand)
19 for einZustand in meineKreuzung.suedAmpel.zustaende
20       [meineKreuzung.suedAmpel.zustaendeNum[meineKreuzung.suedAmpel.zustand]]:
21   zustandsListe.append(einZustand)
22 #print zustandsListe
23 headerZeile = "Content-type: text/html\n\n"
24
25 print headerZeile
26
27 print seite % tuple(zustandsListe)
28
```

22. Programme mit grafischer Oberfläche

The empty handed painter from your streets
Is drawing crazy patterns on your sheets

(Bob Dylan[1])

22.1. Tkinter

Das Toolkit von J. Ousterhout, abgekürzt ganz einfach „tk“, hat eine Schnittstelle für Python, die sich „tkinter“ nennt. Die Benutzung dieses Toolkits macht deutlich, dass grundsätzlich bei einem Programm unterschieden werden muss zwischen dem Teil, der das gegebene Problem löst, und dem Teil, der die Lösung in einer dem Benutzer angenehmen Form präsentiert.

Vor allem Anfänger neigen dazu, das Design einer Schnittstelle zwischen Problemlösung und Benutzer vorrangig zu bearbeiten, weil sich jeder, auch ein Programmieranfänger, zutraut, mit grafischen Tools eine grafische Oberfläche zu erstellen. Weil dies hier aber ein Programmierkurs ist, wurde das Design, die Benutzerfreundlichkeit, die Bedienbarkeit eines Programms bisher nicht in Angriff genommen.

Den wichtigsten Vorteil von tk (und damit von Tkinter) will ich hier aber auch nicht verschweigen: auch „tk“ ist freie Software und kann aus dem Internet heruntergeladen werden. In den meisten Python-Distributionen ist Tkinter allerdings sogar eingebaut.

22.2. Problem: Temperaturen umrechnen (zuerst ohne grafische Oberfläche) . . .

Hier soll keine Einführung in Tkinter gegeben werden, sondern nur an einem Beispiel gezeigt werden, wie die verschiedenen Komponenten zusammenspielen. Es soll ein kleiner Umrechner für Temperaturen (Celsius nach Fahrenheit und umgekehrt) geschrieben werden.

Dazu wird zuerst eine angemessene Klasse erzeugt:

Abbildung 22.1. Klassendiagramm `Temperatur`

In Python sieht der dazugehörige Quellcode so aus:

[1] It's all over now, Baby Blue *auf:* Bringing it all back home

Beispiel 22.2.1 Klassendefinition: Temperaturen

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4
5   class Temperatur():
6     def __init__(self):
7         self.celsius = 0.001
8         self.fahrenheit = 0.002
9
10    def leseCelsius(self):
11        self.celsius = input("Temperatur in Grad Celsius eingeben: ")
12
13    def leseFahrenheit(self):
14        self.fahrenheit = input("Temperatur in Grad Fahrenheit eingeben: ")
15
16    def celsius2fahrenheit(self):
17        self.fahrenheit = 9.0 / 5.0 * self.celsius + 32.0
18
19    def schreibeFahrenheit(self):
20        print "Temperatur in Grad Fahrenheit: ",self.fahrenheit
21
22    def fahrenheit2celsius(self):
23        self.celsius = 5.0 / 9.0 * (self.fahrenheit - 32)
24
25    def schreibeCelsius(self):
26        print "Temperatur in Grad Celsius: ",self.celsius
27
28    def wertZuruecksetzen(self):
29        self.celsius = 0.01
30        self.fahrenheit = 0.02
31
```

Wie immer gehört dazu ein aufrufendes Programm, das wieder einmal in einer While-Schleife Eingaben ermöglicht und dann diese bearbeitet.

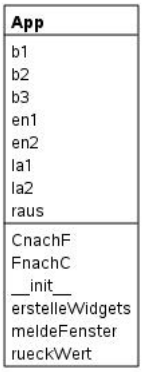
Beispiel 22.2.2 Aufruf der Klasse Temperatur

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4   from cl_Temperatur import Temperatur
5
6   modus = 'x'
7   t1 = temperatur()
8   while modus <> 'E':
9     modus = raw_input('\nAuswahl: \nCelsius nach Fahrenheit mit C\n
10          Fahrenheit nach Celsius mit F\nEnde mit E: ')
11    if modus == 'C':
12      t1.leseCelsius()
13      t1.celsius2fahrenheit()
14      t1.schreibeFahrenheit()
15    elif modus == 'F':
16      t1.leseFahrenheit()
17      t1.fahrenheit2celsius()
18      t1.schreibeCelsius()
19    elif modus == 'E':
20      pass
21    else:
22      print 'Falsche Eingabe. Nur C, F, E erlaubt'
23
24  print 'Ciao'
25
```

22.3. ... und jetzt mit grafischer Oberfläche

Auch dazu wird zuerst eine Klasse definiert, nämlich die Klasse der grafischen Oberfläche. Diese benötigt einige Teile des Tkinter-Moduls und natürlich die Klasse `Temperatur`. Ein solches grafisches Fenster leitet sich von der Tkinter-Klasse `frame` ab. Die Buttons werden mit `b1` bis `b3` sowie `raus` bezeichnet, die Eingabefelder (auf englisch entry) mit `en1` bis `en2` und die Textfelder (auf englisch label) mit `la1` bis `la2`

Abbildung 22.2. Klassendefinition: GUI für die Temperatur



Der Konstruktor der grafischen Oberfläche erstellt das Objekt und ruft die Methode `erstelleWidgets` auf, die dann für die Elemente der Oberfläche zuständig ist. Diese Elemente werden in `tk` widgets genannt. Jedes widget ist ein Objekt einer Tkinter-Klasse, so ist zum Beispiel das widget `b1` ein Objekt der Klasse `Button`. Die Methode `grid` gibt an, wieviel Platz das Objekt im Gitter haben soll. Mit den Methoden `insert` und `delete` werden Texte in die Entry-Objekte geschrieben. Den Buttons `b1` und `b2` werden die `Arbeits`–Methoden, nämlich die Methoden, die die Umrechnung leisten, zugeordnet. Diese Methoden der Oberfläche rufen natürlich nur die Methoden der Klasse Temperatur auf.

Beispiel 22.3.1 Die Klasse für das Fenster

```
1
2   #!/usr/bin/python
3   # -*- coding: utf8 -*-
4   from Tkinter import *
5   from Tkconstants import *
6   from cl_Temperatur import temperatur
7   import tkMessageBox
8
9   t1 = temperatur()
10
11  class App(Frame):
12    def __init__(self,  master = None):
13        Frame.__init__(self,  master)
14        self.master.title('Temperatur-Rechner')
15        self.grid(padx = 10,  pady = 10)
16        self.erstelleWidgets()
17
18    def erstelleWidgets(self):
19        self.raus = Button(self, text="Ende", bg="#C1281F", command=self.meldeFenster)
20        self.raus.grid(row = 1, column = 4, columnspan = 1)
21
22        self.b1 = Button(self, text = "umrechnen Celsius nach Fahrenheit",
23                         command = self.CnachF)
24        self.b1.grid(row = 1, column = 1, columnspan = 1)
25
26        self.b2 = Button(self, text = "umrechnen Fahrenheit nach Celsius",
27                         command = self.FnachC)
28        self.b2.grid(row = 1, column = 2, columnspan = 1)
29
30        self.b3 = Button(self, text = "zurücksetzen",
31                         command = self.rueckWert)
32        self.b3.grid(row = 1, column = 3, columnspan = 1)
33
34        self.la1 = Label(self, text='Grad Celsius')
35        self.la1.grid(row = 2,  column = 1,  columnspan = 1)
36
37        self.en1 = Entry(self, textvariable = t1.celsius)
38        self.en1.insert(0, t1.celsius)
39        self.en1.grid(row = 2, column = 2, columnspan = 1)
40
41        self.la2 = Label(self, text='Grad Fahrenheit')
42        self.la2.grid(row = 2,  column = 3,  columnspan = 1)
43
44        self.en2 = Entry(self, textvariable = t1.fahrenheit)
45        self.en2.insert(0, t1.fahrenheit)
46        self.en2.grid(row = 2, column = 4, columnspan = 1)
47
48    def rueckWert(self):
49        t1.wertZuruecksetzen()
50        self.en1.delete(0, END)
51        self.en1.insert(0,t1.celsius)
52        self.en2.delete(0, END)
53        self.en2.insert(0,t1.fahrenheit)
54
55    def meldeFenster(self):
56        if tkMessageBox.askokcancel('Quit', 'Wirklich aufhören?'):
57            k = self.winfo_toplevel()
58            k.destroy()
59
```

Beispiel 22.3.2 Die Klasse für das Fenster (Forts.)

```
1
2       def CnachF(self):
3           t1.celsius = float(self.en1.get())
4           t1.celsius2fahrenheit()
5           self.en2.delete(0, END)
6           self.en2.insert(0,t1.fahrenheit)
7
8       def FnachC(self):
9           t1.fahrenheit = float(self.en2.get())
10          t1.fahrenheit2celsius()
11          self.en1.delete(0, END)
12          self.en1.insert(0,t1.celsius)
13
```

Das eigentliche Programm, das aufgerufen wird, ist folglich wieder sehr kurz. Es konstruiert nur ein Objekt der gerade definierten Klasse der grafischen Oberlfläche und verschwindet in einer Endlos-Schleife. Diese wird über den Ende-Button verlassen. That's it!

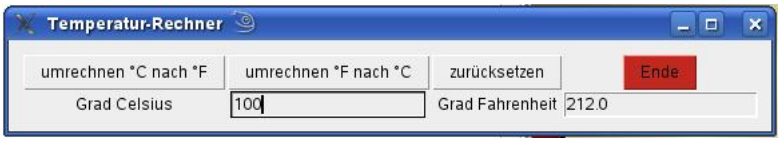
Beispiel 22.3.3 Aufruf des Fensters

```
#!/usr/bin/python
# -*- coding: utf8 -*-
from cl_Gui import *
import tkMessageBox

def callback():
  if tkMessageBox.askokcancel('Quit', 'Wirklich aufhören?'):
      root.destroy()

if __name__ == "__main__":
  root = Tk()
  meineTemp = App(root)
  root.protocol('WM_DELETE_WINDOW', callback)
  meineTemp.mainloop()
```

Eine nette Kleinigkeit wurde noch eingebaut: ein Zusatzrahmen öffnet sich, wenn man das Programm beenden will. Dort wird man gefragt, ob man wirklich aufhören will (es geht auch ohne, und dann ist das Hauptprogramm nochmal 4 Zeilen kürzer).

Das aufgerufene Programm präsentiert sich so:

Abbildung 22.3. Temperaturprogramm



Und das oben erwähnte Zusatzfenster, mit dem man das Programm endgültig beendet, erscheint so:

Abbildung 22.4. Beendigung des Temperaturprogramms

Teil XI.

Python für Verwaltungsaufgaben

23. Python als Sprache für den eigenen Rechner

23.1. Das Betriebssystem

Für die Arbeit mit dem eigenen Rechner ist Python auch sehr gut geeignet. Man erinnere sich: Batteries included! Es gibt für alle Betriebssysteme Bibliotheken, die Aufrufe an das Betriebssystem erleichtern. Das erste Modul ist das `sys`. Rein ins Vergnügen!

Also gehen wir auf die Python-Shell oder in eine IDE unserer Wahl und geben die Befehle **import sys** gefolgt von **dir(sys)** ein. Jede Menge Informationen können vom Modul `sys` geliefert werden. Probieren wir es also aus:

Beispiel 23.1.1 Versuche mit `sys`

```
>>> print sys.platform
linux2
>>> print sys.version
2.7.2 (default, Aug 19 2011, 20:41:43) [GCC]
>>> print sys.copyright
Copyright (c) 2001-2011 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

Hier wurde das Betriebssystem, die Versionsnummer von Python und die Copyrights für Python ausgegeben.

Das ist hilfreich, wenn man ein Programm schreibt, von dem man vermutet, dass es auf verschiedenen Rechnern unter verschiedenen Betriebssystemen laufen soll.

Beispiel 23.1.2 Abfrage des Betriebssystems

```
>>> import sys
>>> if sys.platform[:5] == 'linux':
        print 'Da hast Du aber Glück gehabt! Hier funktioniert fast alles!'
    elif sys.platform[:3] == 'win':
        print 'Na, trotzdem viel Glück!'
    else:
        print 'Exot! Wird schon tun'
Da hast Du aber Glück gehabt! Hier funktioniert fast alles!
```

Nicht schlecht. Aber seit einem der vorigen Kapitel wissen wir, dass durchaus noch mehr Informa-

tionen über einen Modul abgefragt werden können, nämlich mit **help(sys)**. Was ist denn die Standard-Kodierung für Dateien in unserem System?

Beispiel 23.1.3 Standardkodierung abfragen

```
>>> sys.getfilesystemencoding()
'UTF-8'
```

Das zweite Modul, das mit dem Betriebssystem interagiert, ist das Modul `os`. Dieses Modul enthält für fast alle gängigen Betriebssysteme Systemaufrufe, so dass Skripte, die dieses Modul benutzen, oft unverändert auf verschiedenen Systemen funktionieren. Schauen wir also auch hier einmal rein, indem wir zuerst mit **import os** das Modul importieren und dann mit **dir(os)** uns den Inhalt des Moduls anzeigen lassen.

Teil XII.

Glossar

Glossar

C

C C ist eine Programmiersprache, die sehr eng mit dem Betriebssystem Unix verbunden ist. Beim Entwurf von Unix war ein Ziel, den Code des Betriebssystems rechnerunabhängig und besser wartbar zu schreiben. Das war in Maschinensprache oder Assembler, wie es bis dahin üblich war, nicht gut möglich. Die Entwickler von Unix entwarfen zu diesem Zweck eine Programmiersprache, die auf alle möglichen Probleme anwendbar war, strukturierte Programmierung ermöglichte, aber trotzdem systemnah arbeitet. Ein wesentliches Sprachelement von C ist der Pointer (auf deutsch: Zeiger), ein Verweis auf einen Speicherplatz. C-Programme zeichnen sich dadurch aus, dass sie schnell sind, weil sie (unter anderem) Speicherplätze nicht über deren Namen, sondern über die Adresse ansprechen.

CGI Common Gateway Interface. Eine standardisierte Schnittstelle zwischen einem Programm (meistens in einer P-Sprache, also PHP, perl, oder Python) und einem Web-Server.

COBOL COBOL steht für Common Business Oriented Language. Diese Sprache ist geeignet für die Programmierung von wirtschaftlichen Anwendungen. Was sich die Entwickler dieser Sprache dabei gedacht haben? Ob sie Wirtschaftswissenschaftler und Kaufleute für unfähig, eine formale Sprache zu lernen, gehalten haben? Stellen wir einfach einmal gegenüber, wie ein kleineres Problem in Python und in COBOL gelöst wird. Zuerst in Python:

```
a = 3
b = 5
c = a + b
```

und jeder wird verstehen, dass c danach den Wert 8 hat. Jetzt das selbe in COBOL:

```
MOVE 3 TO a.
MOVE 5 TO b.
ADD b TO a GIVING c.
```

Okay, das kann man auch lesen. Vor allem wegen der schönen Punkte am Ende jedes Statements. Aber wer will so etwas schreiben???

Compiler Ein Compiler ist ein Programm, das einen Quelltext in eine vom jeweiligen Computer ausführbare Datei umwandelt. Dabei wird der gesamte Quelltext genommen, auf Syntax-Fehler, semantische Fehler und auch auf Schreibfehler untersucht, in der Regel mit den benötigten Bibliotheken verknüpft und in Maschinensprache übersetzt. [1]

Corba Common Object Request Broker Architecture

CSS Cascading Style Sheets sind Dateien, die Formatierungsanweisungen für verschiedene Auszeichnungssprachen (HTML, XML) enthalten

D

DCOM Distributed Component Object Model

DBMS Database Management System. Ein DBMS ist eine Kombination von Datenbanken mit ihren Tabellen, ihren Zugriffsrechten, ihren Zugriffsbefehlen und manchmal noch mit eine grafischen Oberfläche.

Default ist das, was standardmäßig vorgegeben ist, sprachlich taucht es meistens als Defaultwert auf.

[1] siehe auch [Compiler]

DTD Document Type Definition. Hier wird ein Modell eines XML-Dokuments beschrieben.

Dummy Ein Dummy ist ein Stellvertreter für irgendetwas. Er zeichnet sich dadurch aus, dass er sich gar nicht auszeichnet: ein Dummy hat nichts und kann nichts von dem, was das Original auszeichnet.

F

Fortran Fortran ist die Zusammenschreibung der ersten Silben von „Formular translator“. Daraus geht schon hervor, dass diese Programmiersprache sich in Mathematik, Physik und den Ingenieurwissenschaften zu Hause fühlt. Fortran-Compiler sind meistens recht teuer, aber die Sprache ist nicht vergnügungssteuerpflichtig. Nein, es macht nicht unbedingt Spaß, Fortran zu programmieren, aber die Sprache kann mit Zahlen dafür sehr gut umgehen.

H

HTML Hypertext Markup Language. Die Sprache des Internet. HTML ist eine Seitenbeschreibungssprache, die sich als Standard für Internet-Seiten durchgesetzt hat.

I

IDE IDE, Abkürzung für Integrated Development Environment.

Interpreter Ein Interpreter wandelt (im Gegensatz zum Compiler) einen Quelltext nicht als Ganzes in Maschinensprache um, sondern zeilenweise. Das bedeutet, dass der Quelltext erst zur Laufzeit des Programms auf Korrektheit untersucht wird. Ist der Maßstab für ein Programm die Ausführungsgeschwindigkeit, so sind interpretierte Programme langsamer als compilierte, weil vor der Ausführung eines Befehls dieser erst in Maschinensprache übersetzt werden muss. [2]

M

Monty Python Englische Komikertruppe. Hier sind einige Internet-Seiten zu Monty Python:

- <http://de.wikipedia.org/wiki/Monty_Python>
- <http://de.wikipedia.org/wiki/Das_Leben_des_Brian>
- Welcome to the completely unauthorized, completely silly, and completely useless Monty Python web site! <http://www.intriguing.com/mp/>

O

ODBC Open Database Connectivity. Standardisierte Schnittstelle zwischen Programmen in verschiedenen Programmiersprachen und Datenbanken verschiedener DBMS.

P

perl perl ist eine der drei P-Sprachen, die bei der Erstellung dynamischer Internet-Seiten benutzt wird. perl wird interpretiert und hat seine Stärken in der Behandlung von Texten.

perl ist die eierlegende Wollmilchsau. In perl kann man alles machen, vor allem kann man alles kaputt machen. Böse Menschen behaupten, das Motto von perl sei „write once, never read again“, und das ist durchaus ernst zu nehmen: oft versteht der perl-Programmierer schon nach einer Woche nicht mehr, was er da geschrieben hat.

[2] siehe auch [Interpreter]

Pseudocode Pseudocode ist kein Programmcode in irgendeiner Programmiersprache, sondern die Formulierung eines Algorithmus in Umgangssprache, wobei aber die logische Struktur programmiersprachenähnlich abgebildet wird. So werden in Pseudocode etwa die typischen reservierten Wörter für Alternativen und Iterationen (if - then - elseif - else bzw. while ... bzw. for ...) benutzt, die Aktionen, die auf diese Strukturierungsmittel folgen, in Umgangssprache beschrieben.

Q

Quelltext, auch Quellcode Der Quelltext eines Programms ist das, was der Programmierer schreibt. Dazu benutzt er eine Programmiersprache. Den Quelltext versteht der Computer noch nicht; er muss erst in Maschinensprache umgewandelt werden.

S

SGML Standard Generalized Markup Language.

SQL Structured Query Language. **Die** Datenbank-Sprache. Der kleinste gemeinsame Nenner.

Source siehe Quelltext

U

UML UML ist die Abkürzung für Unified Modelling Language, die Softwaresysteme, Geschäftsprozesse, aber auch Unternehmensmodelle veranschaulichen soll. Gleichzeitig ist UML auch Basis für die Konstruktion von Software auf der Basis dieser Modelle und für deren Dokumentation. UML hat viele „Unter-“Sprachen, die verschiedene Diagrammarten hervorbringen. Diese ergeben sich aus den verschiedenen Perspektiven, unter denen man Geschäftsprozesse, die beteiligten Objekte und die Akteure betrachtet. Die wichtigsten Diagrammarten von UML sind:

- Klassendiagramme
- Anwendungsfalldiagramme
- Zustandsdiagramme
- Aktivitätsdiagramme

Die Sprache hat begonnen, sich durchzusetzen, als Software immer komplexer wurde und die Wiederverwertbarkeit von Software für Unternehmen ein wichtiges Kriterium wurde.

Für die Programmierung — also für den vorliegenden Text — sind nur Klassendiagramme von Bedeutung.

URL URL ist die Abkürzung für Uniform Resource Locator. Es handelt sich hier um eine Adress-Angabe im WWW, zusammen mit einer Angabe, mit welchem Protokoll die Daten übermittelt werden, z.B. http oder ftp.

W

W3C World Wide Web Consortium

X

XML XML ist die Abkürzung für Extensible Markup Language. In einer Markup Language werden — anders als in einem Textverarbeitungsprogramm — keine Eigenschaften des Aussehens wie Schriftgröße oder Schriftstil festgelegt, sondern Textstellen wird eine Klasse zugeordnet, etwa die Klasse „Kapitelüberschrift“. Erst in einem weiteren Schritt wird dann allen mit Kapitelüberschrift ausgezeichneten Elementen ein Aussehen verpasst. Es handelt sich dabei um eine logische Auszeichnung, die aber auch eine semantische Bedeutung zum Inhalt hat. So erwartet XML etwa, dass nach einer Kapitelüberschrift ein Absatz kommen muss, sicher aber keine Buchüberschrift kommen darf. XML erwartet also sauber strukturierte Dokumente.

A. Lösungen zu Aufgaben

A.1. aus Kapitel 7: Programmstrukturen

Wochentagsberechnung.

```
#!/usr/bin/python
#-*- coding: utf-8 -*-
# berechnet zu einem Datum den Wochentag

wochentage = ["Sonntag", "Montag", "Dienstag", "Mittwoch",
                         "Donnerstag", "Freitag", "Samstag"]

tag = input("Tag eingeben: ")
monat = input("Monat eingeben: ")
jahr = input("Jahr eingeben: ")
monat_r = monat

if (monat > 2):
  monat_r = monat_r - 2
else:
  monat_r = monat_r + 10
  jahr = jahr - 1


(c, a) = divmod(jahr, 100)
# d.h. a = jahr % 100, c = jahr / 100

b = (13 * monat_r - 1) / 5 + a / 4 + c / 4
wotag = (b + a + tag - 2 * c) % 7

print "Der ",tag,".", monat,".",jahr, " ist ein ", wochentage[wotag]
```

A.2. aus Kapitel 8: Schleifen

Sternchen-Pyramide.

```
#!/usr/bin/python
for i in range(1,40,2):
  z = '*'*i
print z.center(40,' ')
```

Quadratzahlen.

```
#!/usr/bin/python
# -*- coding: utf8 -*-

endZahl = input('bis zu welcher Zahl: ')
```

```
for i in range(endZahl+1):
  print "%i*%i = %i" % (i, i, i*i)
```

Sieb des Eratosthenes.

```
#!/usr/bin/python

schranke = input("Bis zu welcher Zahl sollen Primzahlen berechnet werden? ")

alleZahlen = range(1,schranke+1)

alleZahlen[0] = 'x'

start = alleZahlen[1]
while start*start < schranke:
  n = 2
  while start*n <= schranke:
    alleZahlen[start*n - 1] = 'x'
    n += 1
  k = 0
  while type(alleZahlen[start+k]) != int:
    k +=1
  start = alleZahlen[start+k]
print "\n\nSo! Das sind jetzt alle Primzahlen bis ",schranke,"!!"

primz = []
for x in alleZahlen:
  if type(x) == int:
    primz.append(x)

print primz
```

Größter gemeinsamer Teiler und kleinstes gemeinsames Vielfaches mit dem Euklidschen Algorithmus.

```
#!/usr/bin/python
# -*- coding: utf8 -*-
# berechnet ggT und kgV zweier Zahlen mit dem Euklidschen Algorithmus

zz1 = input("erste Zahl eingeben ")
zz2 = input("zweite Zahl eingeben ")
z1 = zz1
z2 = zz2

while (z1 <> z2):
  while (z1 > z2):
    z1 = z1 - z2
while (z2 > z1):
  z2 = z2 - z1

print "ggT von ",zz1," und ",zz2," ist ",z1

print "kgV von ",zz1," und ",zz2," ist ",zz1 * zz2 / z1
```

Stellenwertsysteme.

```
#!/usr/bin/python

zahl = input("Zahl im Dezimalsystem eingeben: ")
modus = input("Basis des Stellenwertsystems,
                  in das die Zahl umgerechnet werden soll: ")
if modus > 10:
  mehrAlsHex = ['0','1','2','3','4','5','6','7','8','9',
                          'A','B','C','D','E','F','G','H','I','J','K','L']

restliste = []
while(zahl > 0):
  (zahl,rest) = divmod(zahl, modus)
  if modus > 10:
    restAlpha = mehrAlsHex[rest]
    restliste.append(restAlpha)
else:
  restliste.append(rest)

restliste.reverse()

for i in restliste:
  print i,
```

Mit Uhrzeiten rechnen.

```
#!/usr/bin/python
z1 = [15, 38, 46]
z2 = [12, 47, 58]
sum = [0,0,0]

uebertrag = 0
for i in range(len(z1)-1,-1,-1):
  if i > 0:
    sum[i] = (z1[i] + z2[i] + uebertrag) % 60
    uebertrag = (z1[i] + z2[i]) / 60
  else:
    sum[i] = (z1[i] + z2[i] + uebertrag) % 60
print sum
```

Drucker - Abrechnung.

```
#!/usr/bin/python
# -*- coding: utf8 -*-

print 'Druckkosten-Berechnung'

print 'Menge\tGesamtkosten\tKosten je Blatt'

fixkosten = 840.34
stueckkosten = 0.038
mindestAbnahme = 2000
mwstSatz = 0.19
```

```
for i in range(0, 10001,  500):
  if i == 0:
    print '%-10d %-17.2f' % (i, fixkosten * (1 + mwstSatz))
  elif i <= mindestAbnahme:
    print '%-10d %-17.2f %-12.4f' %
           (i, fixkosten * (1 + mwstSatz),  fixkosten * (1 + mwstSatz)  / i)
  else:
    kosten = (fixkosten + (i - mindestAbnahme)*stueckkosten) * (1 + mwstSatz)
    print '%-10d %-17.2f %-12.4f' %  (i, kosten,  kosten / i)
```

Sierpinski - Pfeilspitze. Zuerst ein Modul, das die Fakultäten und den Binomialkoeffizienten berechnet:

```
#!/usr/bin/python
# -*- coding: utf8 -*-

def fak(zahl):
  fakultaet = 1
  for i in range(zahl, 0,  -1):
    fakultaet *= i
  return fakultaet

def binko(n, k):
  return fak(n) / (fak(n-k) * fak(k))

if __name__ == '__main__':
  z = input('Fakultät bis ... ')
  print fak(z)

  print 'n über k'
  n = input('n = ')
  k = input('k = ')
  print binko(n, k)
```

Und hier das Programm für die Pfeilspitze (es ist wieder peinlich kurz!):

```
#!/usr/bin/python
from fakultaet import binko
for i in range(0,30,1):
  z = ''
  for j in range(i+1):
    if binko(i, j)%2 == 0:
      z += ' '
    else:
      z += 'x'
  print z
```

Staaten mit Hauptstädten (selbstlernend). Hier zuerst das Programm, das die ganze Intelligenz zum Inhalt hat:

```
#!/usr/bin/python
""" Dokumentation von LandHauptst.py
Ein Dictionary Land -> Hauptstadt wird eingerichtet.
```

```
Das gesamte Dictionary wird ausgegeben.
Dann wird ein kleines Raetsel veranstaltet.
"""

def initialisiere():
  global LaHa
  global neuLaHa
  neuLaHa= {}
  LaHa = {'Italien':'Rom',
               'Frankreich':'Paris',
               'Deutschland':'Berlin',
               'Belgien':'Bruessel',
               'Spanien':'Madrid'}

def hilfe():
  for land in LaHa:
    print land, '\t', LaHa[land]

def raetsel():
  punkte = 0
  fehler = 0
  for land in LaHa:
    frageHText = 'Wie heisst die Hauptstadt von '+land+'?'

    stadt = raw_input(frageHText)
    if LaHa[land] == stadt:
      punkte += 1
      print "Gut!!"
    else:
      print 'so nicht'
      print 'schau im Lexikon nach!'
      tippfText = '(falls Deine Eingabe "'+stadt
                        + '" nur ein Tippfehler war, gib jetzt das Wort "Tippfehl

      frageLText =  'von welchem Land ist '+stadt+' die Hauptstadt?'
                          + tippfText

      neuesLand = raw_input(frageLText)
      if neuesLand == 'Tippfehler':
        pass
      else:
        neuLaHa[neuesLand] = stadt
      fehler += 1

  print "Du hast ", punkte, " von ",punkte+fehler," richtige Antworten"
  if fehler > 0:
    print "also nochmal!!!"
  LaHa.update(neuLaHa)
  return (fehler)
```

Und hier das aufrufende Programm:

```
#!/usr/bin/python
import LandHauptst

print LandHauptst.__doc__
```

```
LaHa = {}
neuLaHa = {}
LandHauptst.initialisiere()
while LandHauptst.raetsel() > 0:
  1
```

Caesar-Verschlüsselung.

```
#!/usr/bin/python
# -*- coding: utf8 -*-

dateiName = raw_input('Welche Datei soll verschlüsselt werden? ')
schlBuchst = raw_input('Schlüssel-Buchstabe, um den verschoben werden soll? ')
ausDatName = dateiName+'ver'
leseDat = open(dateiName,'r')
schreibDat = open(ausDatName,'w')

for zeile in leseDat.readlines():
  ausZeile = ''
  for zeichen in zeile:
    if zeichen == ' ':
      ausZeile += '#'
    elif zeichen == '\n':
      ausZeile += zeichen
    else:
      ausZeile += chr(((ord(zeichen) - ord('A') + ord(schlBuchst))%26)+ord('A'))
  schreibDat.write(ausZeile)
schreibDat.close()
leseDat.close()
```

... und auch Entschlüsseln.

```
#!/usr/bin/python
# -*- coding: utf8 -*-

dateiName = raw_input('Welche Datei soll entschlüsselt werden? ')
schlBuchst = raw_input('Schlüssel-Buchstabe, um den verschoben werden soll? ')
ausDatName = dateiName+'ent'
leseDat = open(dateiName,'r')
schreibDat = open(ausDatName,'w')

for zeile in leseDat.readlines():
  ausZeile = ''
  for zeichen in zeile:
    if zeichen == '#':
      ausZeile += ' '
    elif zeichen == '\n':
      ausZeile += zeichen
    else:
      ausZeile += chr(((ord(zeichen) - ord('A') - ord(schlBuchst))%26)+ord('A'))
print ausZeile
schreibDat.close()
leseDat.close()
```

Monoalphabetische Verschlüsselung mit Schlüsselwort.

```
#!/usr/bin/python
# -*- coding: utf8 -*-

dateiName = raw_input('Welche Datei soll verschlüsselt werden? ')
schlWort = raw_input('Schlüsselwort? ')
ausDatName = dateiName+'ver'
leseDat = open(dateiName,'r')
schreibDat = open(ausDatName,'w')

zz = 0
for zeile in leseDat.readlines():
  ausZeile = ''
  for zeichen in zeile:
    ausZeile += chr((ord(zeichen) + ord(schlWort[zz % len(schlWort)]))% 255)
    zz += 1
  schreibDat.write(ausZeile)
schreibDat.close()
leseDat.close()
```

A.3. aus Kapitel 10: Funktionen

Newton'sches Näherungsverfahren.

```
#!/usr/bin/python

# Funktion fest eingebaut
def f(x):
  fwert = x**3 - 2
  return fwert

def ableitung(x):
  abl = 3 * x**2
  return abl

fwert = 100
startwert = 1.0
schwelle = 0.0000001
x = startwert

while abs(fwert) > schwelle:
  fwert = f(x)
  abl = ableitung(x)
  print x,'\t',fwert,'\t',abl
  x = x - fwert / abl
```

Römische Zahlen.

```
#!/usr/bin/env python
# -*- coding: utf8 -*-

roemZahlen = [{1:'I', 5:'V', 10:'X'},
                        {1:'X', 5:'L', 10:'C'},
                        {1:'C', 5:'D', 10:'M'}]
```

```
def einsFuenfZehn(zehnerPotenz,  zahl):
  global roZa
  if zahl == 9:
    roZa += roemZahlen[zehnerPotenz][1]+roemZahlen[zehnerPotenz][10]
  elif zahl == 4:
    roZa += roemZahlen[zehnerPotenz][1]+roemZahlen[zehnerPotenz][5]
  elif zahl >= 5:
    roZa += roemZahlen[zehnerPotenz][5]+(zahl-5)*roemZahlen[zehnerPotenz][1]
  else:
    roZa += zahl*roemZahlen[zehnerPotenz][1]


decZahl = input('Natürliche Zahl im Dezimalsystem eingeben: ')
roZa = ''
while decZahl > 1000:
  decZahl -= 1000
  roZa += roemZahlen[2][10]

(decHun,  decHunRest) = divmod(decZahl,  100)
(decZeh,  decEin) = divmod(decHunRest,  10)

einsFuenfZehn(2, decHun)
einsFuenfZehn(1, decZeh)
einsFuenfZehn(0, decEin)

print roZa
```

B. GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or

discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C State on the Title page the name of the publisher of the Modified Version, as the publisher.

D Preserve all the copyright notices of the Document.

E Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H Include an unaltered copy of this License.

I Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties — for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License,

and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See Copyleft <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

C. GNU Free Documentation License

This is an unofficial translation of the GNU Free Documentation License into German. It was not published by the Free Software Foundation, and does not legally state the distribution terms for documentation that uses the GNU FDL—only the original English text of the GNU FDL does that. However, we hope that this translation will help German speakers understand the GNU FDL better.

Dies ist eine inoffizielle Übersetzung der GNU Free Documentation License (FDLv1.3) ins Deutsche. Sie wurde nicht von der Free Software Foundation veröffentlicht, und legt keine rechtsgültigen Bestimmungen zur Verteilung für Dokumentation fest, welche die GNU FDL verwendet – nur der englische Originaltext der GNU FDL gewährleistet dies. Dennoch hoffen wir, dass diese Übersetzung Deutschsprechenden dazu verhilft, die GNU FDL besser zu verstehen. Urheberrechtshinweis Deutsche Übersetzung © 2008 Ingo Rybaczyk Die unveränderte Vervielfältigung und Verteilung des gesamten Artikels in beliebiger Form ist gestattet, vorausgesetzt, dass dieser Hinweis erhalten bleibt.

GNU-Lizenz für freie Dokumentation Version 1.3, 3 November 2008 Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. Jedem ist es gestattet dieses Lizenzdokument zu kopieren und unveränderte Kopien zu verteilen, aber seine Änderung wird nicht erlaubt.

0. EINLEITUNG

Der Zweck dieser Lizenz ist es, ein Handbuch, Fachbuch, oder ein anderes sachliches sowie nützliches Dokument „frei“ im Sinne von Freiheit anzufertigen: um jedermann die wirkliche Freiheit zuzusichern, es zu kopieren und neu zu verteilen, mit oder ohne Modifikation daran, entweder kommerziell oder nicht kommerziell. Zweitrangig erhält diese Lizenz für den Autor und Herausgeber die Möglichkeit aufrecht, Anerkennung für ihr Werk zu bekommen, während Sie nicht als verantwortlich betrachtet werden für vorgenommene Modifikationen anderer.

Diese Lizenz ist eine Art „Copyleft“, was bedeutet, dass abgeleitete Werke des Dokuments ihrerseits im selben Sinne frei sein müssen. Sie ergänzt die GNU General Public License, welche eine Copyleft-Lizenz darstellt, die für freie Software vorgesehen ist.

Wir haben diese Lizenz zur Verwendung für Handbücher von freier Software konzipiert, da freie Software freie Dokumentation benötigt: Ein freies Programm sollte mit Handbüchern erhältlich sein, welche dieselben Freiheiten zur Verfügung stellen wie es die Software tut. Doch diese Lizenz ist nicht auf Software-Handbücher beschränkt; sie kann für jedes textliche Werk verwendet werden, unabhängig vom Thema oder ob es als ein gedrucktes Buch veröffentlicht wird. Wir empfehlen diese Lizenz vorwiegend für Werke, deren Einsatzzweck Anleitung oder Referenz ist.

1. ANWENDBARKEIT UND DEFINITIONEN

Diese Lizenz trifft auf jedes Handbuch oder sonstiges Werk zu, in beliebiger Form, das einen vom Urheberrechtsinhaber untergebrachten Hinweis mit den Worten enthält, dass es unter den Bestimmungen dieser Lizenz verteilt werden kann. Solch ein Hinweis gewährt eine weltweite, vergütungsfreie Lizenz von unbefristeter Dauer, um dieses Werk unter den hier festgelegten Bedingungen zu verwenden. Das „Dokument“, nachstehend, bezieht sich auf jedes derartige Handbuch oder Werk. Jedes Mitglied der Öffentlichkeit ist ein Lizenznehmer und wird als „Sie“ angesprochen. Sie akzeptieren die Lizenz, wenn Sie die Werke kopieren, modifizieren oder verteilen, was gewissermaßen unter dem Urheberrechtsgesetz die Erlaubnis erfordert.

Eine „modifizierte Version“ des Dokumentes bedeutet, dass jedes Werk das Dokument selbst oder einen Teil davon beinhaltet, entweder unverändert kopiert, oder mit Modifikationen und/oder übersetzt in eine andere Sprache. Ein „untergeordneter Abschnitt“ ist ein benannter Anhang oder ein Titelei-Abschnitt des Dokuments, der sich ausschließlich mit dem Verhältnis der Herausgeber oder Autoren des Dokuments zum Gesamtthema des Dokumentsbefasst (oder damit in Verbindung stehende Bewandtnisse), und nichts beinhaltet was direkt innerhalb des Gesamtthemas fallen könnte. (Wenn das Dokument zu einem Fachbuch über Mathematik gehört, kann ein untergeordneter Abschnitt folglich nichts Mathematisches erläutern.) Die Beziehung könnte ein Anliegen mit historischer Verbindung zum

Thema oder ähnlicher Angelegenheiten, oder bezüglich ihrer rechtlichen, kommerziellen, philosophischen, ethischen oder politischen Position sein.

Die „unveränderlichen Abschnitte“ sind bestimmte untergeordnete Abschnitte, deren Titel zum Beispiel in dem Hinweis, der besagt, dass das Dokument unter dieser Lizenz freigegeben ist, als jene unveränderlichen Abschnitte gekennzeichnet sind. Wenn ein Abschnitt nicht zur obigen Definition von untergeordnet passt, dann ist es nicht erlaubt ihn als unveränderlich zu kennzeichnen. Das Dokument kann null unveränderliche Abschnitte enthalten. Wenn das Dokument keine unveränderlichen Abschnitte kennzeichnet, dann gibt es keine.

Die „Umschlagtexte“ sind bestimmte kurze Textpassagen die als vordere Umschlagtexte oder hintere Umschlagtexte in dem Hinweis, der besagt, dass das Dokument unter dieser Lizenz freigegeben ist, verzeichnet sind. Ein vorderer Umschlagtext darf höchstens 5 Wörter lang sein, und ein hinterer Umschlagtext darf höchstens 25 Wörter lang sein.

Eine „transparente“ Kopie des Dokumentes bedeutet eine maschinenlesbare Kopie, in einem ansehnlichen Format, dessen Spezifikation für die Allgemeinheit verfügbar ist, welches geeignet ist das Dokument unkompliziert mit allgemeinen Texteditoren oder (für aus Pixeln bestehende Bilder) allgemeinen Malprogrammen oder (für Zeichnungen) irgendeinem weit verbreiteten Zeicheneditor zu überarbeiten, und das geeignet ist zur Eingabe in Textformatierer oder zur automatischen Übersetzung in eine Variante von geeigneten Formaten zur Eingabe in Textformatierer.

Eine Kopie, erstellt in einem ansonsten transparenten Dateiformat, dessen Auszeichnung oder fehlende Auszeichnung derart ausgestaltet wurde, um nachträgliche Modifikation durch Leser zu behindern oder zu verhindern, ist nicht transparent.

Ein Bildformat ist nicht transparent, wenn es für irgendeine beträchtliche Menge von Text verwendet wird. Eine Kopie, die nicht „transparent“ ist, wird „undurchlässig“ genannt.

Beispiele von geeigneten Formaten für transparente Kopien beinhalten einfachen ASCII ohne Auszeichnung, Texinfo Eingabeformat, LaTeX Eingabeformat, SGML oder XML unter Verwendung einer öffentlich zugänglichen DTD, und standardkonformes einfaches HTML, PostScript oder PDF, vorgesehen für humane Modifikation. Beispiele für transparente Bildformate beinhalten PNG, XCF und JPG.

Undurchlässige Formate beinhalten proprietäre Formate die nur mit proprietären Textverarbeitungssystemen gelesen und bearbeitet werden können, SGML oder XML für welche die DTD und/oder Bearbeitungswerkzeuge nicht allgemein verfügbar sind, und das maschinengenerierte HTML, PostScript oder PDF, erzeugt mit irgendwelchen Textverarbeitungssystemen, nur für Ausgabezwecke.

Das „Titelblatt“ bedeutet, für ein gedrucktes Buch, das Titelblatt an sich zzgl. solcher nachfolgenden Seiten die notwendig sind, die Lesbarkeit des Materials beizubehalten, wie von dieser Lizenz erfordert, um im Titelblatt zu erscheinen.

Für Werke in Formaten, welche kein Titelblatt als solches haben, bedeutet „Titelblatt“ der Text nahe dem bedeutendsten Auftreten des Titels dieses Werkes, ausgehend vom Anfang des Textkörpers.

Der „Herausgeber“ ist jede Person oder Instanz, welche Kopien des Dokuments an die Öffentlichkeit verteilt.

Ein Abschnitt „Mit dem Titel XYZ“ bedeutet eine benannte Untereinheit des Dokuments, dessen Titel entweder genau XYZ ist, oder XYZ in runden Klammern, gefolgt von Text, welcher XYZ in eine andere Sprache übersetzt. (Hier steht XYZ für einen spezifischen Abschnittsnamen, weiter unten erwähnt, wie zum Beispiel „Danksagungen“, „Widmungen“, „Befürwortungen“ oder „Verlauf“.) Den „Titel“ eines solchen Abschnitts „zu erhalten“, wenn Sie das Dokument modifizieren, bedeutet, dass dieser ein Abschnitt „Mit dem Titel XYZ“ gemäß seiner Definition bleibt.

Das Dokument kann neben dem Hinweis, der festlegt, dass diese Lizenz auf das Dokument zutrifft, Garantie-Ausschlussklauseln beinhalten. Diese Garantie-Ausschlussklauseln werden als aufgenommen betrachtet, durch Verweis in dieser Lizenz, aber nur was die Ausschließung von Garantien betrifft: Jede andere Auswirkung, die diese Garantie-Ausschlussklauseln haben können, ist ungültig und hat keine Wirkung auf die Bedeutung dieser Lizenz.

2. UNVERÄNDERTE VERVIELFÄLTIGUNG

Sie dürfen das Dokument in beliebiger Form kopieren und verteilen, entweder kommerziell oder nicht kommerziell, vorausgesetzt, dass diese Lizenz, die Urheberrechtshinweise und der Lizenzhinweis mit den Worten, dass diese Lizenz auf das Dokument zutrifft, in allen Kopien wiedergegeben werden, und dass Sie keine anderen Bedingungen, welcher Art auch immer, zu jenen dieser Lizenz hinzufügen.

Sie dürfen keine technischen Maßnahmen anwenden die das Lesen oder weitere Vervielfältigung von den Kopien, die Sie erstellen oder verteilen, behindern oder kontrollieren. Allerdings können Sie Bezah-

lung im Austausch für Kopien entgegennehmen. Wenn Sie eine ausreichend große Anzahl von Kopien verteilen, müssen Sie außerdem die Bedingungen in Abschnitt 3 beachten.

Sie dürfen außerdem Kopien verleihen, unter den selben, oben angegebenen Bedingungen, und Sie dürfen Kopien öffentlich auslegen.

3. VERVIELFÄLTIGUNG GROßER MENGEN

Wenn Sie, mehr als 100 umfassend, gedruckte Kopien eines Dokuments veröffentlichen (oder Kopien in Medien, die üblicherweise gedruckte Umschläge haben) und der Lizenzhinweis des Dokuments Umschlagtexte erfordert, müssen Sie die Kopien in Umschläge beifügen, welche eindeutig und leserlich alle diese Umschlagtexte tragen: vordere Umschlagtexte auf vordere Umschläge, und hintere Umschlagtexte auf hintere Umschläge. Beide Umschläge müssen, ebenso eindeutig und leserlich, Sie, als den Herausgeber dieser Kopien, identifizieren.

Der vordere Umschlag muss den vollständigen Titel, mit allen Wörtern des Titels, in gleicher Weise auffallend und leicht erkennbar darstellen. Sie dürfen darüber hinaus sonstiges Material auf die Umschläge anbringen. Vervielfältigung mit Änderungen, begrenzt auf die Umschläge, sofern sie den Titel des Dokuments erhalten und diese Bedingungen erfüllen, können ansonsten als unveränderte Vervielfältigung behandelt werden.

Wenn die erforderlichen Texte für beide Umschläge zu umfangreich sind, um lesbar zu passen, sollten Sie die ersten verzeichneten (so viele, um angemessen zu passen) auf den aktuellen Umschlag setzen, und den Rest auf den nachfolgenden Seiten fortsetzen.

Wenn Sie, mehr als 100 umfassend, undurchlässige Kopien des Dokuments veröffentlichen oder verteilen, müssen Sie entweder eine maschinenlesbare transparente Kopie zusammen mit jeder undurchlässigen Kopie einbeziehen, oder in bzw. mit jeder undurchlässigen Kopie eine Computer-Netzwerkadresse angeben, von welcher die allgemeine netzwerknutzende Öffentlichkeit, unter Verwendung von Netzwerkprotokollen öffentlicher Standards, Zugang hat, um eine vollständig transparente Kopie, frei von hinzugefügtem Material, herunterzuladen. Falls Sie die letzte Option verwenden, müssen Sie angemessen überlegte Schritte unternehmen, wenn Sie mit der Verteilung von undurchlässigen Kopien in großen Mengen beginnen, um sicherzustellen, dass diese transparente Kopie unter der angegebenen Adresse auf diese Weise zugänglich bleibt, bis mindestens ein Jahr nachdem Sie zum letzten Mal eine undurchlässige Kopie (unmittelbar oder durch Ihre Vertreter oder Einzelhändler) in dieser Ausgabe an die Öffentlichkeit verteilen.

Es ist erwünscht, aber nicht erforderlich, dass Sie in Kontakt mit den Autoren des Dokuments treten, kurz bevor Sie irgendeine große Anzahl von Kopien neu verteilen, um ihnen die Möglichkeit zu geben, Sie mit einer aktualisierten Version des Dokuments zu versorgen.

4. MODIFIKATIONEN

Sie dürfen eine modifizierte Version des Dokuments unter den oben erwähnten Bedingungen der Abschnitte 2 und 3 kopieren und verteilen, vorausgesetzt, dass Sie die modifizierte Version unter genau dieser Lizenz freigeben, mit der modifizierten Version wird die Rolle des Dokuments besetzt, und somit der Lizensierung von Verteilung und Modifikation der modifizierten Version, für jeden der eine Kopie davon besitzt. Zusätzlich müssen Sie diese Sachen in der modifizierten Version erledigen:

A Verwenden Sie auf dem Titelblatt (und gegebenenfalls auf den Umschlägen) einen Titel der sich von dem des Dokuments unterscheidet, und von jenen der vorhergehenden Versionen (die, wenn es irgendwelche gab, in dem Verlaufs-Abschnitt des Dokuments verzeichnet sein sollten). Sie dürfen denselben Titel wie den einer vorhergehenden Version verwenden, wenn der ursprüngliche Herausgeber dieser Version die Erlaubnis gibt.

B Verzeichnen Sie auf dem Titelblatt, als Autoren, eine oder mehrere Personen oder Organe, verantwortlich für die Autorschaft der Modifikationen in der modifizierten Version, zusammen mit mindestens fünf der Hauptautoren des Dokuments (alle seine Hauptautoren, wenn es weniger als fünf hat), es sei denn, dass sie Sie von dieser Anforderung befreien.

C Geben Sie auf dem Titelblatt den Namen des Herausgebers der modifizierten Version als den des Herausgebers an.

D Erhalten Sie alle Urheberrechtshinweise des Dokumentes.

E Fügen Sie einen entsprechenden Urheberrechtshinweis für Ihre Modifikationen, angrenzend zu den anderen Urheberrechtshinweisen, hinzu.

F Nehmen Sie, direkt nach den Urheberrechtshinweisen, einen Lizenzhinweis auf, der die öffentliche Erlaubnis gibt, die modifizierte Version unter den Bestimmungen dieser Lizenz zu verwenden, in der Form, wie weiter unten im Anhang gezeigt.

G Erhalten Sie in diesem Lizenzhinweis die vollständigen Listen der unveränderlichen Abschnitte und erforderlichen Umschlagtexte, aufgeführt in dem Lizenzhinweis des Dokuments.

H Nehmen Sie eine ungeänderte Kopie dieser Lizenz auf.

I Erhalten Sie den Abschnitt mit dem Titel „Verlauf“, erhalten Sie seinen Titel und fügen Sie ihm einen Punkt hinzu, der mindestens den Titel, das Jahr, neue Autoren und Herausgeber der modifizierte Version angibt, wie auf dem Titelblatt aufgeführt. Wenn es keinen Abschnitt „Verlauf“ in dem Dokument gibt, erzeugen Sie einen, der den Titel, das Jahr, die Autoren und Herausgeber des Dokuments angibt, wie auf seinem Titelblatt aufgeführt, dann fügen Sie einen Punkt hinzu, der die modifizierte Version beschreibt, wie in dem vorhergehenden Satz angegeben.

J Erhalten Sie gegebenenfalls die Netzwerkadresse, aufgeführt in dem Dokument, für den öffentlichen Zugang zu einer transparenten Kopie des Dokuments, und ebenso die in dem Dokument aufgeführten Netzwerkadressen, für vorhergehende Versionen, auf dem es beruht. Diese können in den Abschnitt „Verlauf“ untergebracht werden. Sie dürfen eine Netzwerkadresse für ein Werk auslassen, das mindestens vier Jahre vor dem Dokument selbst veröffentlicht wurde, oder der ursprüngliche Herausgeber der Version, auf die es sich bezieht, die Erlaubnis gibt.

K Für jeden Abschnitt mit dem Titel „Danksagungen“ oder „Widmungen“, erhalten Sie den Titel des Abschnitts, und erhalten Sie in dem Abschnitt vollständig den wesentlichen Gehalt und Umgangston der, von jeglichen Mitwirkenden darin aufgeführten, Danksagungen und/oder Widmungen.

L Erhalten Sie alle unveränderlichen Abschnitte des Dokuments, ungeändert in ihrem Text und ihren Titeln. Abschnittsnummern oder Entsprechendes werden nicht als Teil der Abschnittstitel betrachtet.

M Löschen Sie jeden Abschnitt mit dem Titel „Befürwortungen“. Solch ein Abschnitt darf nicht in die modifizierte Version aufgenommen werden.

N Betiteln Sie keinen Titel eines vorhandenen Abschnitts neu, der mit „Befürwortung“ betitelt ist oder in Konflikt zum Titel irgendeines unveränderlichen Abschnitts steht.

O Erhalten Sie jegliche Garantie-Ausschlussklauseln.

Wenn die modifizierte Version neue Titelei-Abschnitte oder Anhänge beinhaltet, die als untergeordnete Abschnitte qualifiziert sind und kein kopiertes Material aus dem Dokument enthalten, können Sie Ihrer Wahl nach einige oder alle diese Abschnitte als unveränderlich bestimmen. Um dies zu tun, fügen Sie ihre Titel zur Liste der unveränderlichen Abschnitte in den Lizenzhinweis der modifizierten Version hinzu. Diese Titel müssen sich von allen anderen Abschnittstitel unterscheiden.

Sie können einen Abschnitt mit dem Titel „Befürwortungen“ hinzufügen, vorausgesetzt, dass dieser nichts enthält als nur Befürwortungen Ihrer modifizierten Version durch verschiedene Parteien – zum Beispiel Aussagen der Begutachtung oder dass der Text von einer Organisation als maßgebliche Definition eines Standards anerkannt worden ist.

Sie können eine Textpassage von bis zu fünf Wörtern als einen vorderen Umschlagtext, und eine Textpassage von bis zu 25 Wörtern als hinteren Umschlagtext in der modifizierten Version hinzufügen. Nur eine Textpassage des vorderen Umschlagtextes und eine des hinteren Umschlagtextes kann von (oder durch, von ihr, angefertigte Zusammenstellung) irgendeiner Person hinzugefügt werden. Wenn das Dokument bereits einen Umschlagtext für denselben Umschlag beinhaltet, zuvor von Ihnen hinzugefügt oder durch Zusammenstellung, angefertigt von derselben Person, in dessen Namen Sie handeln, können Sie keinen weiteren hinzufügen; aber Sie dürfen den alten ersetzen, mit ausdrücklicher Erlaubnis des vorherigen Herausgebers, welcher den alten hinzufügte.

Der/die Autor(en) und Herausgeber des Dokuments geben durch diese Lizenz für keine modifizierte Version die Erlaubnis ihre Namen für Werbung zu verwenden oder Befürwortung zu behaupten oder anzudeuten.

5. DOKUMENTE KOMBINIEREN

Sie dürfen das Dokument mit anderen, unter dieser Lizenz freigegeben, Dokumenten kombinieren, unter den Bestimmungen, definiert in Abschnitt 4 für modifizierte Versionen, vorausgesetzt, dass Sie in die Kombination alle unveränderlichen Abschnitte aller Originaldokumente, nicht modifiziert, einbeziehen und sie alle als unveränderliche Abschnitte Ihres kombinierten Werkes in dessen Lizenzhinweis verzeichnen, und dass Sie alle ihre Garantie-Ausschlussklauseln erhalten.

Das kombinierte Werk muss nur eine Kopie dieser Lizenz enthalten, und mehrfach identische unveränderliche Abschnitte können gegen eine einzelne Kopie ausgetauscht werden. Wenn es mehrfach identische unveränderliche Abschnitte mit demselben Namen, aber unterschiedlichem Inhalt gibt, erzeugen Sie den Titel eines jeden solchen Abschnitts an seinem Ende eindeutig durch Hinzufügen, in runden Klammern, des Namens, falls bekannt, des ursprünglichen Autors oder Herausgebers dieses Abschnittes, oder andernfalls einer eindeutigen Nummer. Nehmen Sie dieselbe Anpassung bei den Abschnittstiteln in der Liste von unveränderlichen Abschnitten in dem Lizenzhinweis des kombinierten Werkes vor.

In der Kombination müssen Sie jegliche Abschnitte mit dem Titel „Verlauf“ in den verschiedenen Originaldokumenten kombinieren und einen Abschnitt mit dem Titel „Verlauf“ bilden; ebenso kombinieren Sie jegliche Abschnitte mit dem Titel „Danksagungen“, und jegliche Abschnitte mit dem Titel „Widmungen“. Sie müssen alle Abschnitte mit dem Titel „Befürwortungen“ löschen.

6. SAMMLUNGEN VON DOKUMENTEN

Sie dürfen eine Sammlung erzeugen, bestehend aus dem Dokument und anderen Dokumenten, freigegeben unter dieser Lizenz, und die Einzel-Kopien dieser Lizenz in den verschiedenen Dokumenten gegen eine einzelne Kopie, die in die Sammlung aufgenommen wird, austauschen, vorausgesetzt, dass Sie die Regeln dieser Lizenz für unveränderte Vervielfältigung jedes Dokuments in jeder anderen Hinsicht befolgen.

Sie dürfen ein einzelnes Dokument aus solch einer Sammlung entnehmen, und es individuell unter dieser Lizenz verteilen, vorausgesetzt, dass Sie eine Kopie dieser Lizenz in das extrahierte Dokument einfügen, und diese Lizenz in jeder anderen Hinsicht, bezüglich der unveränderten Vervielfältigung dieses Dokuments, befolgen.

7. GRUPPIERUNG MIT UNABHÄNGIGEN WERKEN

Eine Zusammentragung des Dokuments oder seiner Ableitungen mit anderen gesonderten und unabhängigen Dokumenten oder Werken, in oder zu einem Band auf einem Speicher- oder Verteilungsmedium, wird eine „Gruppierung“ genannt, wenn das Urheberrecht, das sich aus der Zusammentragung ergibt, nicht verwendet wird, um die gültigen Rechte der Benutzer dieser Zusammentragungen darüber hinaus, was die Einzel-Werke gestatten, zu beschränken. Wenn das Dokument in eine Gruppierung aufgenommen ist, trifft diese Lizenz nicht auf die anderen Werke in der Gruppierung zu, welche nicht selbst abgeleitete Werke des Dokuments sind.

Wenn die Anforderung für Umschlagtext von Abschnitt 3 auf diese Kopien des Dokuments anwendbar ist, wenn außerdem das Dokument weniger als eine Hälfte der gesamten Gruppierung darstellt, kann der Umschlagtext des Dokuments auf Umschläge, die das Dokument innerhalb der Gruppierung einklammern, untergebracht werden, oder der elektronischen Entsprechung von Umschlägen, wenn das Dokument in elektronischer Form vorliegt. Andernfalls müssen sie auf gedruckten Umschlägen vorkommen, welche die vollständige Gruppierung einklammern.

8. ÜBERSETZUNG

Übersetzung wird als eine Art von Modifikation betrachtet, also dürfen Sie Übersetzungen unter den Bestimmungen von Abschnitt 4 verteilen. Das Austauschen unveränderlicher Abschnitte mit Übersetzungen erfordert besondere Erlaubnis von ihren Urheberrechtsinhabern, aber Sie können Übersetzungen von einigen oder allen unveränderlichen Abschnitten aufnehmen, zusätzlich zu den Originalversionen dieser unveränderlichen Abschnitte. Sie können eine Übersetzung dieser Lizenz aufnehmen, und alle Lizenzhinweise in dem Dokument, und jegliche Garantie-Ausschlussklauseln, vorausgesetzt, dass Sie außerdem die englische Originalversion dieser Lizenz und die Originalversionen jener Hinweise und Ausschlussklauseln aufnehmen.

Im Falle eines Widerspruchs zwischen der Übersetzung und der Originalversion dieser Lizenz oder eines Hinweises oder einer Ausschlussklausel, wird sich die Originalversion durchsetzen.

Wenn ein Abschnitt in dem Dokument mit „Danksagungen“, „Widmungen“ oder „Verlauf“ betitelt ist, wird die Anforderung (Abschnitt 4), seinen Titel (Abschnitt 1) zu erhalten, normalerweise die Änderung des tatsächlichen Titels erfordern.

9. SCHLUSSBESTIMMUNG

Sie dürfen das Dokument nicht kopieren, modifizieren, unterlizensieren oder verteilen, außer, als ausdrücklich unter dieser Lizenz zur Verfügung gestellt. Jeder andere Versuch es zu kopieren, zu modifizieren, zu unterlizensieren oder zu verteilen ist ungültig, und wird automatisch Ihre Rechte unter dieser Lizenz aufkündigen.

Jedoch, wenn Sie jeglichen Verstoß gegen diese Lizenz beenden, dann ist ihre Lizenz eines bestimmten Urheberrechtsinhabers wieder (a) vorläufig, sofern und solange der Urheberrechtsinhaber nicht ihre Lizenz beendet, und (b) dauerhaft, wenn der Urheberrechtsinhaber es versäumt, Sie über den Verstoß zu benachrichtigen, mit angemessenen Mitteln, vorzeitig 60 Tage nach der Beendigung.

Darüber hinaus ist Ihre Lizenz eines bestimmten Urheberrechtsinhabers wieder dauerhaft, wenn der Urheberrechtsinhaber Sie über den Verstoß mit angemessenen Mitteln benachrichtigt, es das erste Mal ist, dass Sie eine Benachrichtigung über den Verstoss dieser Lizenz (für jedes Werk) vom Urheberrechtsinhaber erhalten, und Sie den Verstoß vorzeitig 30 Tage nach Erhalt der Benachrichtigung beseitigen.

Die Beendigung Ihrer Rechte unter diesem Abschnitt beendet nicht die Lizenzen der Parteien, welche Kopien oder Rechte von Ihnen unter dieser Lizenz erhalten haben. Wenn Ihre Rechte aufgekündigt und nicht wieder dauerhaft geworden sind, gibt Ihnen der Erhalt einer Kopie von Teilen oder Sämtlichem des gleichen Materials keine Rechte, diese zu verwenden.

10. ZUKÜNFTIGE ÜBERARBEITUNGEN DIESER LIZENZ

Die Free Software Foundation kann von Zeit zu Zeit neue, überarbeitete Versionen der GNU-Lizenz für freie Dokumentation veröffentlichen. Solche neuen Versionen werden im Geist zur gegenwärtigen Version gleich sein, aber können sich im Detail unterscheiden, um neue Probleme oder Anliegen anzusprechen. Siehe „http://www.gnu.org/copyleft/“.

Jeder Version der Lizenz wird eine charakteristische Versionsnummer gegeben. Wenn das Dokument spezifiziert, dass eine besonders nummerierte Version dieser Lizenz „oder jede spätere Version“ auf sich zutrifft, haben Sie die Wahl den Bestimmungen und Bedingungen entweder dieser spezifizierten Version oder jeder späteren Version, die (nicht als Entwurf) durch die Free Software Foundation veröffentlicht worden ist, zu folgen. Wenn das Dokument spezifiziert, dass ein Bevollmächtigter entscheiden darf welche zukünftigen Versionen dieser Lizenz verwendet werden können, berechtigt Sie das dauerhaft, durch die öffentliche Aussage desjenigen Bevollmächtigten über die Akzeptanz einer Version, diese Version für das Dokument zu wählen.

11. RELIZENZIERUNG

„Massive Multiauthor Collaboration Standort“ (oder „MMC-Standort“) meint jeglichen World Wide Web Server, welcher urheberrechtsfähige Werke veröffentlicht und ferner bedeutende Möglichkeiten für jeden bietet, solche Werke zu editieren. Ein öffentliches Wiki, welches jeder bearbeiten kann, ist ein Beispiel eines solchen Servers. Eine „Massives Multiautor Collaboration“ (oder „MMC“), enthalten im Standort, bedeutet jegliche Zusammenstellung urheberrechtsfähiger Werke, die folglich auf dem MMC-Standort veröffentlicht werden.

„CC-BY-SA“ meint die Creative Commons Namensnennung-Weitergabe unter gleichen Bedingungen 3.0 Lizenz, veröffentlicht von der Creative Commons Corporation, einem gemeinnützigen Unternehmen mit Hauptgeschäftssitz in San Francisco, Californien, sowie zukünftige Copyleft-Versionen dieser Lizenz, welche durch dieselbe Organisation veröffentlicht wird.

„Einbeziehen“ bedeutet, ein Dokument zu veröffentlichen oder neu zu veröffentlichen, ganz oder teilweise, als Bestandteil eines anderen Dokuments.

Eine MMC ist „geeignet zur Relizenzierung“ wenn sie unter dieser Lizenz lizensiert wird, und wenn alle Werke, die zuerst unter dieser Lizenz irgendwo anders als auf der MMC veröffentlicht, und nachträglich ganz oder teilweise in der MMC einbezogen wurden, (1) keine Umschlagtexte oder unveränderliche Abschnitte hatten, und (2) folglich vor dem 1. November 2008 einbezogen wurden.

Der Betreiber eines MMC-Standortes kann eine MMC, enthalten im Standort unter CC-BY-SA des gleichen Standortes, jederzeit vor dem 1. August 2009 neu veröffentlichen, vorausgesetzt, die MMC ist für die Relizenzierung geeignet.

D. Literatur zu Python

Literatur

[ABOP] *A Byte of Python*, C.H. Swaroop, Copyright © 2003-2005 Creative Commons Attribution-NonCommercial-ShareAlike License 2.0, .

[Compiler] <http://de.wikipedia.org/wiki/Compiler> , , Copyright © , . 1

[DocBook-XML] *DocBook-XML*, Thomas Schraitle, Copyright © 2004 SuSE Press, 3-89990-078-2. (document)

[FSF] <http://www.gnu.org/philosophy/free-sw.de.html>, , Copyright © , . 6

[Freie Software] *Freie Software*, Volker Grassmuck, Copyright © 2002 Bundeszentrale für politische Bildung Bonn, 3-89331-432-6. 5

[Friedl] *Mastering Regular Expressions* , Jeffrey E.F. Friedl, Copyright © 2002 O'Reilly, 0-596-00289-0. 5.5

[Interpreter] <http://de.wikipedia.org/wiki/Interpreter> , , Copyright © , . 2

[Java] *Thinking in Java*, Bruce Eckel, Copyright © 2003 Prentice Hall, 3-8272-6896-6. (document)

[Lusth] *The Alchemy of Programming: Python* <http://beastie.cs.ua.edu/cs150/book/index.html> , John C. Lusth, Copyright © 2012 , . 3.3.5

[Lutz/Ascher] *Einführung in Python*, Mark LutzDavid Ascher, Copyright © 2007 O'Reilly, Köln, 3-89721-488-0.

[Programming Python] *Programming Python*, Mark Lutz, Copyright © 2006 O'Reilly, Sebastopol, CA 95472, USA, 0-596-00925-9. 2, 5, 2.6.1, 3, 20.6

[Programming Python2] *Programming Python*, Mark Lutz, Copyright © 19966 O'Reilly, Sebastopol, CA 95472, USA, 1-56592-1975-6. 7

[Python] *Python*, Peter Walerowski, Copyright © 2009 Addison-Wesley, 978-3-8273-2517-4. 4

[Python and XML] *Python and XML*, Christopher A. JonesFred L. Jr. Drake, Copyright © 2002 O'Reilly, Köln, 3-89721-175-0. (document)

[dummdeutsch] *Dummdeutsch*, Eckard Henscheid, Copyright © 1993 Reclam, Stuttgart, 3-15-008865-8. 9

E. Index

Algorithmus, 29
Alternative, 34, 84
Array, 65
Attribut, 119
Aufgaben
    zu Alternative, 88
    zu Dateien, 160
    zu Dictionaries, 79
    zu Funktionen, 108
    zu Klassen, 146
    zu Programmierung allgemein, 39
    zu Schleifen, 96
    zu Sequenzen, 83
    zu Variablen, 51
Ausdrücke
    reguläre, 69
Ausnahme, 151
Auswahl, 84

Betriebssystem, 33
    Befehle, 33

C, 30
Cleese, John, 25
COBOL, 30
Compiler, 29

Dateien, 155
Dateisystem, 33
Datenbank, 161
    MySQL, 161
Datenstrukturen
    Dictionary, 78
    Liste, 73
    Tupel, 79
Debugger, 39
Dezimalpunkt, 46
Dictionary, 78
divmod, 47
DocBook-XML, 18
docstring, 36

Editor, 31
    Syntax highlighting, 31
    Wortvervollständigung, 31
Entscheidung, 34
exception, 151
expression
    regular, 69

Fehler, 38
    Debugger, 39
Fehlerarten
    Laufzeitfehler, 39
    semantischer Fehler, 39
    Syntaxfehler, 39
Feld, 65
for-Schleife, 89
Fortran, 30
Free Documentation Licence, 17
Free Software Foundation, 17
Freie Software, 30
FSF, 17
Funktionen, 98

Gültigkeitsbereich, 102

Hilfetext, 36

IDE, 32
idle, 43
Idle, Eric, 25, 43
Import, 111
input, 60
Interpreter, 29
Iteration, 34, 84

Kamel-Schreibweise, 119
Klasse
    Attribut, 119
    Methode, 119
Kodierung, 68
Kommentar, 35
Konstruktor, 121
Konventionen, 36

Liste, 73
    flache Kopie, 77
    kopieren, 77
lokale Variable, 102

Methode, 119
Modul, 111
Monty Python, 25
MySQL, 161

Namensraum, 102

Operatoren
    arithmetische, 47
    Shortcut, 55
    Vergleichs-, 52
    Verschiebe-, 54

Parameter, 98
PEP 8, 37
private, 126
Programmaufruf, 59
Programmierrichtlinien, 37
Programmiersprachen
    C, 30
    COBOL, 30
    Fortran, 30
Programmierung
    strukturierte, 83
protected, 126
public, 126

Quellcode, 27, 57
Quelltext, 29

range, 89
raw strings, 67
raw_input, 60
Regeln für Bezeichner
    Klassen und Objekte, 119
    Variablen, 49
reguläre Ausdrücke, 69
regular expression, 69
Reservierte Wörter, 38
reservierte Wörter, 38

Schlüsselwörter, 38
Schleife, 34, 89
    Zähl-, 89
Schleifen-Beispiele, 93
Selbstreferenz, 121
self, 121
Sequenz, 34, 83
sha-bang, 59
Shortcut-Operatoren, 55
Sichtbarkeit
    private, 126
    protected, 126
    public, 126
Slicing, 66
Steuerzeichen, 64
Stil, 36
strings, 63
strukturierte Programmierung, 83
    Alternative, 34, 84
    Entscheidung, 34
    For-Schleife, 89
    Iteration, 34, 84
    Schleife, 34, 89
    Sequenz, 34, 83, 84
    While-Schleife, 92
Syntax highlighting, 31

Tupel, 79

UML, 122
    Umbrello, 122

Variable, 49
    lokale, 102
    Wertzuweisung, 50
Variablennamen, 49
Vergleichsoperatoren, 52
Verschiebeoperatoren, 54

Wörter;
    reservierte, 38
Wertzuweisung, 50
while-Schleife, 89
Wiederholung, 84
Wortvervollständigung, 31

XML
    DocBook, 18

Zählschleife, 89
Zahlen
    binäre, 55
    hexadezimale, 55
    komplexe, 54
    Zahlsysteme, 55
Zeichenketten, 63
    Teilbereiche von, 66
Zeilenvorschub, 64
Zuweisungsmuster, 51
Zuweisungsoperator, 50