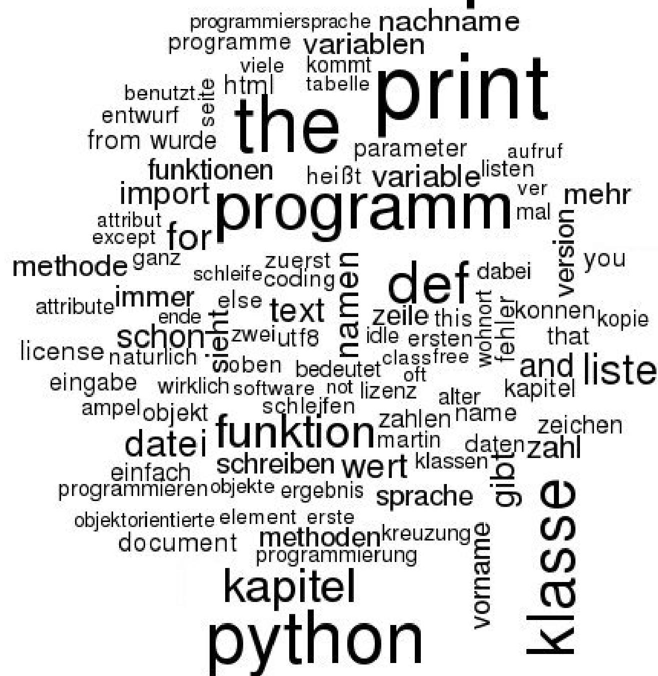
Einführung in Python



Martin Schimmels

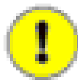
9. März 2024

Gesetzt mit LaTeX und KOMA-Script von ms

©2009, 2018, 2022 Martin Schimmels

Fehler, Verbesserungen, Ergänzungen etc. an martin(@)mschimmels.de
Mit Arbeitsblättern von Frank Krafft und Jens Stephan
Danke an Jürgen Adam für Korrekturen, Ergänzungen, Tips.
Die Wort-Wolke auf der Titelseite wurde mit `R` und dem Paket `wordcloud` erstellt.

Wichtig

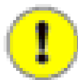


Druck dieses Buch nicht aus!!!

Im Online-Dokument kann man suchen, springen, ... Alle Verweise etwa in Literatur-, Tabellen-, Inhaltsverzeichnis oder auf externe Seiten sind farblich hinterlegt.

Es wird (hoffentlich) ständig verändert und verbessert. Und es benötigt nicht eine wichtige natürliche Ressource: Papier.

Wichtig



Das ist die erste Version, die nicht mit DocbookXML verfasst wurde, sondern direkt in LaTeX. Die aus den XML-Dateien generierten LaTeX-Dateien mussten nachbearbeitet werden. Was ich festgestellt habe: bei der Generierung sind nicht alle Referenzen richtig übertragen haben. Da bin ich für Hinweise dankbar, wenn irgendein Link nicht funktioniert!!!

Dieses Skript steht unter der Creative Commons Lizenz BY-NC-SA, auf Deutsch, „Creative Commons Namensnennung-Keine kommerzielle Nutzung-Weitergabe unter gleichen Bedingungen.“
Diese Lizenz erlaubt es, das Skript zu benutzen, zu ergänzen, zu verändern, Inhalte umzustellen. Diese Lizenz erlaubt es **nicht**, das Skript für kommerzielle Zwecke zu nutzen. Der Link zu dieser Lizenz ist
https://creativecommons.org/licenses/by-nc-sa/3.0/de/
Nachfolgend steht die Zusammenfassung der wichtigsten Teile der Lizenz:

- Das Skript darf vervielfältigt, weitergegeben und öffentlich zugänglich gemacht werden.
- Das Skript darf verändert werden.
- Ich als Autor muss in Kopien genannt werden.
- Das Skript darf nicht kommerziell verwendet werden.
- Veränderte oder bearbeitete Versionen dieses Skriptes müssen unter der selben Lizenz stehen. Das heißt auch, dass solche Versionen genau wie dieses Skript einen Link zur Lizenz enthalten müssen.

Inhaltsverzeichnis

I Vorgeplänkel 17

1 Danke schön an . . . 19

2 Vorbemerkungen 21
2.1 Typographische Konventionen . . . 21
2.2 Freiheit . . . 23
2.3 Gleichberechtigung . . . 24
2.4 Wer? Wo? Wie? . . . 24
2.5 Eine Begründung . . . 25
2.5.1 Andere Sprachen . . . 25
2.5.2 Entscheidungskriterien . . . 27
2.5.3 Entscheidung . . . 28
2.5.4 Schön einfach? Einfach ist schön . . . 28
2.5.5 Lernen durch Nachmachen? Lernen durch Selbertun? . . . 29

II Grundlagen 31

3 Was ist Python? 33
3.1 Der Name . . . 33
3.2 Was für eine Sprache? . . . 34
3.3 Warum Python und nicht eine andere Sprache? . . . 35
3.4 Ausführung von Python-Programmen . . . 37

4 Programmieren 41
4.1 Was heißt „Programmieren“ ? . . . 41
4.1.1 Warum soll man überhaupt programmieren lernen? . . . 42
4.1.2 Sprache lernen . . . 44
4.1.3 Übersetzung ... in welche Sprache denn? . . . 44
4.2 Was ist also ein Programm? . . . 45
4.3 Geschichte . . . 45
4.4 Was braucht man, um mit Python zu arbeiten? . . . 46
4.5 Hilfsprogramme . . . 47
4.6 Voraussetzungen . . . 48
4.6.1 Das Dateisystem . . . 48

4.6.2 Regeln für Dateinamen . . . 49
4.6.3 Datei-Operationen . . . 50
4.7 Was heißt „Programmieren“ ? Fortsetzung! . . . 51
4.7.1 Regeln für die Sprache Python: PEP8 . . . 51
4.7.2 Strukturierte Programmierung: Ein Überblick . . . 52
4.8 Objektorientierung . . . 56
4.9 Programmierstil und Konventionen . . . 57
4.10 Reservierte Wörter . . . 59
4.11 Fehler (zum ersten) . . . 60
4.12 Fehler finden . . . 61
4.13 . . . und das fehlt . . . 62
4.14 Aufgaben . . . 62

III IDE 63

5 Die Entwicklungsumgebung IDLE: Zahlen und Variable 65
5.1 . . . und bevor es losgeht 65
5.2 Rechnen in der Entwicklungsumgebung . . . 65
5.2.1 Ein Vorgriff: Decimal . . . 75
5.2.2 Aufgaben zu Zahlen und elementaren Berechnungen . . . 75
5.3 Ein Speicherplatz mit einem Namen: Variable . . . 76
5.3.1 Variable in Mathematik und in der Programmierung . . . 76
5.3.2 Variablennamen . . . 79
5.3.3 Wertzuweisung . . . 81
5.3.4 Aufgaben zu Variablen . . . 82
5.3.5 Zuweisungsmuster . . . 83

6 Richtig programmieren 85
6.1 Entwicklungsumgebungen . . . 85
6.1.1 Die Entwicklungsumgebung Eric . . . 85
6.1.2 IDE unter Windows . . . 86
6.1.3 So sieht’s aus . . . 87
6.1.4 Der Programm-Rahmen . . . 87
6.2 Das absolute Muss: Hallo Ihr alle . . . 88
6.3 Die ersten einfachen Programme . . . 89

IV Texte und andere Daten 91

7 Texte 93
7.1 Grundlegendes zu Texten . . . 93
7.1.1 Lange Texte . . . 97
7.1.2 Operationen auf Texten . . . 98

7.1.3 Methoden von Zeichenketten 100
7.1.4 Wie und wo gespeichert wird 104
7.2 Kodierungen 104
7.2.1 Unicode in Python 3 106
7.3 Formatierung von Zeichenketten 108
7.3.1 Die C-ähnliche Formatierung mit dem %-Operator 108
7.3.2 Die Formatierung mit Hilfe der `format`-Methode 109
7.4 Reguläre Ausdrücke 117
7.4.1 Allgemeines zu regulären Ausdrücken 118
7.4.2 Wortteile aus einem Text herausfiltern 120
7.4.3 gpx-Datei eines Fitness-Trackers 122
7.4.4 Beispiel für Zahlen 123
7.4.5 Ein komplizierterer Text 124
7.4.6 Wörter mit Doppelbuchstaben finden 125
7.5 Aufgaben zu Texten (Strings) 127

8 Strukturierte Daten **129**
8.1 Überblick 129
8.2 Listen 129
8.2.1 Definition von Listen und Listenelemente 129
8.2.2 Erzeugung von Listen 130
8.2.2.1 Durch Angabe der Elemente 130
8.2.2.2 Als Objekt der Klasse `list` 130
8.2.2.3 Mittels `range` 130
8.2.2.4 Mit Hilfe der `list comprehension` 131
8.2.2.5 Mittels `input` 132
8.2.3 Anzeigen von Listenelementen 134
8.2.3.1 Beispiele von Listen 135
8.2.4 Operationen auf Listen 136
8.2.4.1 Wie funktioniert das „Enthaltensein“? 136
8.2.4.2 Erzeugung einer Liste durch eine Filterfunktion 137
8.2.5 Veränderung von Listen 137
8.2.5.1 Ein Element anhängen 137
8.2.5.2 bisect oder „divide et impera“ 139
8.2.5.3 Ein gern gemachter Fehler 140
8.2.5.4 Mehrere Elemente anhängen 141
8.2.5.5 Ein Element einfügen 144
8.2.5.6 Ein bestimmtes Element entfernen (nicht so schön) 145
8.2.5.7 Ein bestimmtes Element entfernen (schöner) 145
8.2.5.8 Ein bestimmtes Element bearbeiten und entfernen 145
8.2.5.9 Eine Teilmenge der Liste bearbeiten 145
8.2.5.10 Eine Liste sortieren 146
8.2.6 Tricks mit Listen 147
8.2.6.1 Matrizen 147

8.2.6.2 Listenelemente mit Namen ansprechen . . . 149
8.2.6.3 Bestimmte Elemente aus Liste herausziehen . . . 150
8.2.6.4 Richtige Datentypen aus einer Zeichenkette . . . 151
8.2.6.5 Mehrere Listen auf einmal bearbeiten . . . 152
8.2.7 Kopie einer Liste . . . 153
8.2.8 Nicht mehr ganz so wilde Listen . . . 154
8.2.9 Wie und wo gespeichert wird (jetzt bei Listen) . . . 157
8.3 Dictionaries . . . 158
8.3.0.1 Die Frage kommt immer wieder 158
8.3.1 Zugriff auf Dictionary-Elemente . . . 159
8.3.1.1 Dictionary lesen . . . 159
8.3.1.2 Dictionary schreiben . . . 163
8.3.2 Dictionaries sortiert ausgeben . . . 164
8.3.3 Elemente von Listen oder Dictionaries . . . 165
8.3.4 Tricks mit Dictionaries . . . 166
8.3.4.1 zippen von Dictionaries . . . 166
8.3.4.2 Dictionary Comprehension . . . 166
8.3.4.3 Dictionaries und Listen und `bisect` . . . 167
8.4 Tupel . . . 168
8.4.1 Allgemeines zu Tupeln . . . 168
8.4.1.1 Erzeugen von Tupeln . . . 169
8.4.2 Benannte Tupel . . . 171
8.4.3 ...so ähnlich wie Splitten . . . 171
8.4.4 Tupel mit Komfort . . . 172
8.5 Zusammenfassung . . . 174
8.6 Was bei (fast) allen strukturierten Daten funktioniert . . . 174
8.6.1 Die Funktion `enumerate` . . . 174
8.6.2 Die Funktion `map` . . . 174
8.7 Iteratoren . . . 175
8.8 Mengen (sets) . . . 176
8.8.1 Mengenoperationen mit Rechenzeichen . . . 177
8.8.2 Mengenoperationen als Methoden der Klasse Menge . . . 178
8.8.3 Mengen aus anderen Daten erzeugen . . . 180
8.9 Formatierung der Ausgabe (Fortsetzung) . . . 181
8.9.1 Typen oder Objekte einer Klasse . . . 183
8.10 Aufgaben zu Listen, Dictionaries 183
8.11 Veränderbarkeit von Daten . . . 184
8.12 Generatoren . . . 185

V Strukturen **187**

9 Programmstrukturen **189**
9.1 Gute Programme, schlechte Programme . . . 189

9.1.1 Kontrollfluß . . . 189
9.2 Eins nach dem anderen: Die Sequenz . . . 190
9.2.1 Aufgaben zu Sequenz . . . 190
9.3 Wenn . . . dann: Die Alternative . . . 191
9.3.1 Wahrheit . . . 192
9.3.1.1 Sprachliche Ungenauigkeiten und Fallen . . . 197
9.4 Wahrheit angewandt: die Alternative . . . 198
9.4.1 Blöcke und Einrückungen . . . 200
9.4.2 Beispiele zu Bedingungen . . . 203
9.4.2.1 Kurzschlüsse . . . 205
9.4.3 Beispiel für eine Mehrfach-Entscheidung . . . 205
9.4.4 So sehen Mehrfachentscheidungen schöner aus! . . . 207
9.4.5 Ein Trick für verschachtelte Entscheidungen . . . 208
9.4.6 Aufgaben zu Auswahl . . . 210
9.4.7 Verknüpfte Logik-Operatoren und Reihenfolge . . . 211
9.5 Schleifen . . . 215
9.5.1 Zähl-Schleifen . . . 216
9.5.1.1 Mitzählen in einer Zählschleife . . . 223
9.5.2 While-Schleifen . . . 224
9.5.3 Verschiedene Probleme und ihre Lösung mittels Schleifen . . . 226
9.5.3.1 Eine Liste von Listen abarbeiten . . . 226
9.5.3.2 List Comprehensions . . . 227
9.5.3.3 Die Anzahl der Elemente einer Liste bestimmen . . . 230
9.5.3.4 Elemente mit einer bestimmten Eigenschaft . . . 230
9.5.3.5 . . . wie eben, aber mit Ergebnisliste . . . 231
9.5.3.6 Das größte Element einer Menge finden . . . 231
9.5.3.7 An welcher Stelle steht das größte Element einer Menge? 232
9.5.4 Ein bißchen Klassik . . . 232
9.5.5 Eigentlich keine Schleife . . . 234
9.5.6 Eingriffe in das Durchlaufen von Schleifen . . . 235
9.5.7 Was schief gehen kann . . . 237
9.5.8 Aufgaben zu Schleifen . . . 238
9.6 Funktionen . . . 242
9.6.1 Allgemeines zu Funktionen . . . 242
9.6.2 Eingebaute Funktionen . . . 243
9.6.3 Funktionen selbstgebaut . . . 243
9.6.3.1 Definition und Aufruf von Funktionen . . . 243
9.6.3.2 Rückgabewerte und Parameterlisten . . . 245
9.6.3.3 Nachtrag zu Listen . . . 249
9.6.3.4 Verkettung von Funktionen . . . 251
9.6.3.5 Gültigkeitsbereiche und Namensräume . . . 252
9.6.4 Funktionen wiederverwenden . . . 254
9.6.5 Ein Trick, um Funktionen zu testen . . . 255
9.6.6 Globale Variable . . . 256

9.6.7 Rekursive Funktionen . . . 259
9.6.8 Funktionen als Parameter von Funktionen . . . 259
9.6.8.1 Seiteneffekte . . . 261
9.6.9 Aufgaben zu Funktionen . . . 261
9.6.10 lambda-Funktionen . . . 262
9.6.11 Sicherheit, Effizienz, Eleganz . . . 264

VI Programmentwicklung und Modularisierung 267

10 Programmentwicklung mit Test 269
10.1 Der Doctest . . . 269
10.1.1 Aufgaben zu doctest . . . 272

11 Module und Pakete 273
11.1 Mehr Mathematik . . . 273
11.2 Eigene Module . . . 276
11.3 Pakete . . . 278

VII Objektorientierte Programmierung 279

12 Klassen 281
12.1 Ist schon klasse, so eine Klasse! . . . 281
12.2 Objekte, Objekte, Objekte . . . 288
12.2.1 Objektvariable und Klassenvariable . . . 292
12.3 Kapselung . . . 292
12.4 Setter und Getter? Oder doch lieber nicht? . . . 295
12.5 Statische Methoden . . . 298
12.6 Vererbung (ohne Erbschaftssteuer) . . . 300
12.6.1 Klassen neuen Stils . . . 303
12.6.2 Weiter mit der Klasse `Mensch` . . . 304
12.7 Methoden überschreiben . . . 305
12.7.1 Selbstgeschriebene Methoden . . . 305
12.7.2 „Magische“ Methoden . . . 306
12.8 Gute Programme . . . wann schreibt man objektorientiert? . . . 310

13 Ein etwas längeres Programm 313
13.1 Objektorientierter Entwurf . . . 313
13.2 Objektorientierte Programmierung . . . 314
13.2.1 Die Klasse `Konto` . . . 314
13.2.2 Das aufrufende Programm . . . 314
13.2.3 Realisierung der Methoden „Einzahlen“ und „Auszahlen“ . . . 315
13.2.4 Die Verzinsung . . . 317
13.2.5 Die ersten Verbesserungen . . . 319

13.2.6 Aufgaben zu Klassen 321
13.3 Ein klassisches Beispiel: Stack und Queue 322
13.3.1 Die Queue 322
13.3.2 Der Stack 323

VIII Fehler und Ausnahmen **325**

14 Fehler . . . **327**
14.1 . . . macht jeder 327
14.2 Fehler werden abgefangen 329
14.3 Spezielle Fehler 332
14.3.1 Ein Menu-Schnipsel mit Fehlerbehandlung 335

IX Permanente Speicherung **337**

15 Dateizugriff **339**
15.1 Dateien allgemein 339
15.1.1 Weitere Datei-Zugriffsmodi 343
15.1.2 Zippen 343
15.1.3 **print** funktioniert auch! 344
15.1.4 Textanalyse 344
15.2 Fortführung des Konten-Programms 345
15.3 In die Datei damit!!! 348
15.3.1 Der Modul `pickle` 348
15.3.2 Der Modul `shelve` 349
15.4 Aufgaben zu Dateien 351

16 Datenbanken **353**
16.1 MySQL 353
16.2 . . . mit Python-Bordmitteln 356

X Noch ein paar Beispiele zur OOP **357**

17 Ein weiteres Projekt: ein Getränkeautomat **359**
17.1 Objektorientierter Entwurf und Realisierung der Klasse 359
17.2 Die Waren kommen in den Automaten 362
17.3 Geld regiert die Welt 363
17.4 Jetzt kann gekauft werden! 363
17.5 To Do!! 364

18 Noch ein Beispiel: Zimmerbuchung in einem Hotel **365**
18.1 Vorstellung des Projekts 365

18.2 Der erste Entwurf: Eine Klasse für Hotelzimmer 365
18.3 Der zweite Entwurf: ein aufrufendes Programm 367
18.4 Der dritte Entwurf: das Hotel 368
18.5 Der vierte Entwurf: Methoden werden ergänzt 370

19 Und noch ein Beispiel: ein Adressbuch 373
19.1 Vorstellung des Projekts 373
19.2 Der erste Entwurf: eine Liste von Listen 373
19.3 Der zweite Entwurf: ein fauler Trick 375
19.4 Der dritte Entwurf: eine Liste von Dictionaries 375
19.5 Der vierte Entwurf: Aktionen (aber nur angedeutet)!! 376
19.6 Der fünfte Entwurf: Aktionen (jetzt tut sich wirklich etwas)!! 379
19.7 Der sechste Entwurf: persistente Speicherung 381
19.8 Jetzt wird es objektorientiert 383
19.8.1 Der Klassenentwurf 383
19.8.2 Zur Klasse `Adresse` kommt die Klasse `Adressbuch` hinzu 385
19.8.3 Dauerhafte Speicherung 386

20 Immer noch das Adressbuch — nur schöner 391
20.1 Das Adressbuch kommt in die Datenbank 391

21 Eine Ampel 397
21.1 Der Entwurf 397
21.2 Die Realisierung 397
21.3 Persistente Speicherung 399
21.4 Eine Kreuzung hat 4 Ampeln! Der Entwurf. 400
21.5 Die Realisierung 403

XI Grafik! Internet! 405

22 CGI-Programme 407
22.1 HTML und Kollegen 407
22.2 Allgemeines zu HTML 407
22.2.1 Was ist HTML? 407
22.2.2 Grundlagen 408
22.2.3 Obligatorische HTML-Befehle 408
22.2.4 Struktur eines HTML-Dokuments 408
22.2.5 Absätze und Leerzeichen 408
22.2.6 Hervorhebungen 409
22.2.7 Listen 409
22.2.8 Links 409
22.2.9 Formulare 410
22.3 CGI (Das Common Gateway Interface) 411

22.4 Die Pflicht: hallo, ihr alle da draußen 412
22.5 Hello world als dynamische Web-Seite 412
22.6 Ein eigener Mini-Webserver 414
22.7 Dynamik durch Schleifen 415
22.7.1 Vorwärtszählen 415
22.7.2 . . . und rückwärtszählen 416
22.7.3 Das kleine 1 x 1 417
22.8 Formular-Eingabe und Auswertung 420
22.9 Eine Rechnung 423
22.10Jetzt aber wirkliche Dynamik 430

23 Eine Kreuzung auf meiner Web-Seite **433**
23.1 Die Ampel kommt ins Netz 433
23.2 Die Kreuzung im Netz ist auch nicht schwerer 435

24 Programme mit grafischer Oberfläche **441**
24.1 Tkinter 441
24.2 Problem: Temperaturen umrechnen (zuerst ohne grafische Oberfläche) . . . 442
24.3 . . . und jetzt mit grafischer Oberfläche 443

XII Python für Verwaltungsaufgaben **449**

25 Python als Sprache für den eigenen Rechner **451**
25.1 Das Betriebssystem 451
25.2 Verzeichnisse und Dateien 453
25.2.1 Die Grundlagen 453
25.2.2 Das Modul `os` 454
25.2.2.1 Beispiel: Massenumbenennung von Dateien 456
25.2.3 Modul `subprocess` 456
25.3 Zeit und Datum 458
25.3.1 Das Modul `time` 458
25.3.1.1 Zeitmessung beim Programmlauf 459
25.3.2 Das Modul `datetime` 460
25.3.2.1 Die Klasse `timedelta` 461
25.4 Aufruf von Programmen 461
25.5 Weiter mit Verzeichnissen: das Modul `glob` 464
25.6 Das Modul `shutil` 465
25.7 Das Modul `os.walk` 465

XIIITexte für Fortgeschrittene **467**

26 Texte bearbeiten für Fortgeschrittene **469**
26.1 Natural Language Toolkit (NLTK) 469

26.1.1 NLTK installieren und einrichten 469
26.1.2 Einfache Bearbeitung von Texten 469

XIV Mathematik (Forts.) 473

27 Mathematik (Forts.) 475
27.1 Das Paket `numpy` 475
27.1.1 Matrizen 475
27.1.1.1 Erzeugung von Matrizen 476
27.1.1.2 Matrizen umformen 482
27.1.1.3 Elementare Zeilenumformungen 484
27.1.1.4 Rechnen mit Matrizen 486
27.1.1.5 Matrizengleichungen 491
27.2 Symbolische Mathematik (und andere schöne Dinge) 491
27.3 Pandas 492
27.3.1 Series 492
27.3.2 Dateien, die Pandas lesen kann 503
27.3.3 Data Frames 505
27.3.4 Indizierung und andere Möglichkeiten der Auswahl 505
27.3.4.1 `Series` 505

XV Verzeichnisse 509

28 Glossar 511

29 Lösungen zu Aufgaben 517
29.1 aus Kapitel 7: Programmstrukturen 517
29.2 aus Kapitel 8: Schleifen 518
29.3 aus Kapitel 10: Funktionen 525

Abbildungsverzeichnis

2.1 6 einfache quadratische Spiralen 21
2.2 6 verdrehte quadratische Spiralen 23
2.3 3 sechseckige Spiralen 24

3.1 Monty Python 34
3.2 Interpreter und Compiler 38
3.3 Compiler 38
3.4 Interpreter 39

5.1 Eric Idle (CC BY 2.0) 66
5.2 Die IDE Idle 67
5.3 Erste Berechnung in der IDE Idle 68
5.4 Rechnungen in Idle 70
5.5 Ein Variablen-Karton mit Inhalt 76

6.1 Die IDE Eric 85
6.2 Die IDE PyScripter 86

7.1 Texte in der IDE Idle 95
7.2 Multiplikation von Texten 96

8.1 Struktogramm binäre Suche 137

12.1 Klassendiagramm `Mensch` 287
12.2 Klassendiagramm `Schueler` 300

13.1 Konten 313

17.1 Klassendiagramm `Automat` 359
17.2 Klassendiagramm `Ware` 362

18.1 Klassendigramm `Hotelzimmer` 365
18.2 Klassendiagramm `Hotel` 368

19.1 Klassendiagramm `Adresse` 383

21.1 Klassendiagramm `Ampel` 397
21.2 Eine Kreuzung 400

21.3 Kreuzung: Freie Fahrt von Nord nach Süd 401
21.4 Kreuzung: Alle warten 401
21.5 Kreuzung: Freie Fahrt von West nach Ost 402
21.6 Kreuzung: Wieder warten alle 402

22.1 Eingabe-Formular in HTML(Namen) 421
22.2 Eingabe-Formular für eine Rechnung in HTML 424
22.3 Ausgabe einer Rechnung in HTML (aber wirklich nicht schön) 427
22.4 Ausgabe einer Rechnung in HTML (so kann man das akzeptieren) 430

23.1 Web-Seite Ampel 434
23.2 Kreuzung im Browser 437

24.1 Klassendiagramm `Temperatur` 442
24.2 Klassendefinition: GUI für die Temperatur 444
24.3 Temperaturprogramm 446
24.4 Beendigung des Temperaturprogramms 447

Teil I.

Vorgeplänkel

1. Danke schön an ...

- Donald Knuth, Lesley Lamport, Markus Kohm ... ist ja klar, warum
- alle, die sich für freie Software einsetzen und freie Software schreiben.
- alle, 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“ [2]
- Benjamin Franklin: „That as we enjoy great Advantages from the Inventions of Others, we should be glad of an Opportunity to serve others by any Invention of ours, and this we should do freely and generously.“ [3]

[1] Richard Stallman, Datum und Ort unbekannt
[2] in: Me and Bobby McGee
[3] Benjamin Franklin, Autobiographie [Part III, p. 98], zitiert nach http://en.wikiquote.org/wiki/Benjamin_Franklin

2. Vorbemerkungen

2.1. Typographische Konventionen

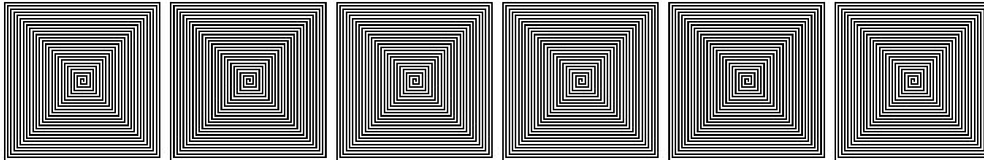


Abbildung 2.1.: 6 einfache quadratische Spiralen

Bei manchen Kapiteln stehen solche Bilder voran. Sie sind mit Python und der Bibliothek „turtle“ gemalt. Wer sich über die Schildkröte informieren möchte, sollte einfach im Internet suchen. Vorschläge für Suchbegriffe: turtle, logo, Seymour Papert

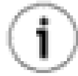
Jetzt fängt das Buch aber richtig an. Zuerst muss hier aber einmal geklärt werden, was die verschiedenen Schriftarten in diesem Buch bedeuten:

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

Listing 2.1: Schriftarten im Buch

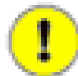
```
1 >>> print('Hallo')
2 Hallo
```

TIPP



Randnotizen stehen im gedruckten Text in einem Rahmen mit einem eingekreisten großen „i“ am Rand.

WICHTIG!



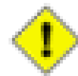
Wenn etwas besonders wichtig ist, steht im gedruckten Text am Rand ein Ausrufezeichen in einem Kreis.

ANMERKUNG

Auf eine Anmerkung wird im gedruckten Text durch den ausgestreckten Zeigefinger hingewiesen.

Anmerkungen werden insbesondere auch benutzt, um auf Änderungen in Python 3.x im Verhältnis zu Python 2.x hinzuweisen.

ACHTUNG



Das Achtung-Zeichen weist auf eine Situation hin, bei der leicht ein Fehler gemacht werden kann. Wenn etwas besonders wichtig ist, steht im gedruckten Text am Rand ein Ausrufezeichen in einer Raute.

2.2. Freiheit

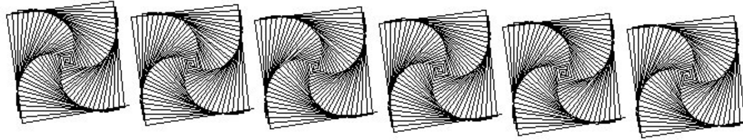


Abbildung 2.2.: 6 verdrehte quadratische Spiralen

Dieser gesamte Text ist frei im Sinne der „Freien Software“ .
Er steht unter der „CC BY-NC-SA“ , die unter dem Link https://creativecommons.org/licenses/by-nc-sa/3.0/de/ nachzulesen ist. Das bedeutet, dass dieser Text beliebig kopiert und verbreitet werden darf. Er darf auch modifiziert werden.

Das Urheberrecht dieses Textes bleibt bei mir. Die wichtigste Bestimmung ist, dass dieser Text immer unter der CC BY-NC-SA bleibt, auch wenn er von Dir verändert wird, und dass die CC BY-NC-SA 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 CC BY-NC-SA 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 CC BY-NC-SA und enthält die CC BY-NC-SA.
3. Der Text darf nicht in ein proprietäres Format umgewandelt werden.
4. Eine Kopie des veränderten Textes wird mir zugesandt.

Dieser Text ist in LaTeX unter Benutzung des Paketes KOMA-Script geschrieben.

Da ich selber es absolut unerträglich finde, wenn Fachbücher der Informatik in schlechtem Deutsch geschrieben sind,[1] [2] bitte ich auch alle Leser, dass sie mir eventuelle sprachliche Fehler mitteilen, gerade auch wenn es nur Leichtsinnsfehler oder kleinere Ungenauigkeiten sind.[3]

[1] Es ist leider betrüblich zu sehen, wie viele Autoren gerade 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?“

[2] Von Frau A.F. habe ich gelernt, dass Schüler inzwischen in Bezug auf die Zeichensetzung nur noch die 5-cm-Regel kennen: Setze alle 5 Zentimeter ein Komma.

[3] Ich bitte wirklich ausdrücklich darum, mir Dinge NICHT durchgehen zu lassen, die heutzutage an

2.3. 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 LaTeX mit KOMA-Script so abzuändern, dass solche generischen Begriffe in eine nach seinen Vorstellungen politisch korrekte Form umwandelt werden.[4]

2.4. Wer? Wo? Wie?

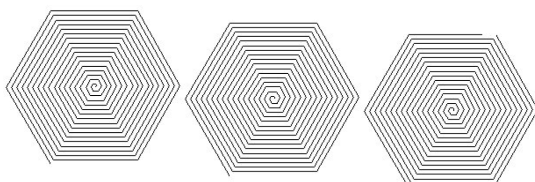


Abbildung 2.3.: 3 sechseckige Spiralen

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 . . .

Schulen viel zu vielen Schülern nachgesehen werden.

[4] Beim ersten Schreiben war noch das grosse Binnen-I die korrekte Schreibweise, inzwischen ist es das Gendersternchen. Ich habe einfach keine Lust, hier Moden hinterherzulaufen und alle paar Jahre diesen Text zu ändern.

[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.

2.5. Eine Begründung

> Ich höre und vergesse.
> Ich sehe und erinnere mich.
> Ich tue und ich verstehe.
>
> *(Konfuzius [6])*

Aber warum sollen wir eigentlich programmieren lernen? Computer sind inzwischen zu einem Werkzeug geworden, von dem man wie etwa von dem Werkzeug „Staubsauger“ nicht mehr wissen muss, wie es funktioniert, um es zu bedienen. So jedenfalls wird das uns in der Werbung immer wieder gesagt. Ist das wirklich so?

Nun, zu welcher Gruppe von Menschen willst Du gehören? Der Gruppe derjenigen, die bedienen können, ohne zu verstehen? Oder zu der Gruppe derjenigen, die verstehen und die Maschine beherrschen? Man muss sich klar machen: Computer machen genau das, was man ihnen „sagt“ , und das machen sie richtig gut, ohne Widerspruch, skrupellos. Vielleicht will ich doch lieber zu der Gruppe gehören, die dem Computer sagen kann, was er tun soll.

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.

2.5.1. Andere Sprachen

Fast alle Programmiersprachen können fast alles! Zumindest können fast alle Programmiersprachen das, was wir in einem Anfängerkurs in Python lernen werden. Aber nicht alle Sprachen machen das elegant, nicht alle Sprachen haben Sprachkonstrukte, die sofort einsichtig sind, nicht alle Sprachen erfordern wenig Vorwissen.

Deswegen soll an einem alltäglichen Beispiel demonstriert werden, wie dieses in verschiedenen Sprachen gelöst wird. Vor allem wird das oben genannte Vorgehen realisiert, also auf allen Schnickschnack verzichtet, der heutzutage zu einem „richtigen“ Programm gehört: eine grafische Oberfläche fehlt, Eingabemasken fehlen. Das Programm benutzt nur die Shell (die bei anderen Betriebssystemen „Eingabeaufforderung“ heißt.)

Ein Programm soll einen Text von der Standard-Eingabe entgegennehmen. 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:

[6] singemäß unter: <http://www.bk-luebeck.eu/zitate-konfuzius.html>

1. In perl gibt es die ganz kurze Version:

   Listing 2.2: Ausgabe eines Textes in perl

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

   oder die fast schon geschwätzige Version:

   Listing 2.3: Ausgabe eines Textes in perl (geschwätzig)

   ```
   $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:

   Listing 2.4: Ausgabe eines Textes in python

   ```
       print(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:

   Listing 2.5: Ausgabe eines Textes in python (geschwätzig)

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

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

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

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

   Listing 2.6: Ausgabe eines Textes in Java

   ```
   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)
   ```

```
9             System.out.println(s);
10        }
11    }
```

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.

4. Und zum Schluß kommt das entsprechende C++-Programm.

Listing 2.7: Ausgabe eines Textes in C++

```
1  #include<iostream>
2  #include<string>
3  using namespace std;
4
5  int main()
6  {
7      string t1;
8      cin >> t1;
9      cout << t1 << endl;
10     return 0;
11 }
```

Kein Kommentar! Aber falls jemand Lust verspürt: einfach mal ein paar Kommentare sammeln von absoluten Programmieranfängern.

2.5.2. Entscheidungskriterien

Man kann die Entscheidung für eine Programmiersprache nach vielen Kriterien treffen. Ein Unternehmen wird die Wirtschaftlichkeit einer Sprache weit oben ansiedeln. Eine interessante (wenn auch schon etwas ältere) Gegenüberstellung findet man bei Steve Ferg unter http://de.wikipedia.org/wiki/Compiler. Aber Wirtschaftlichkeit sollte nicht nur unter dem Kostenaspekt gesehen werden. Auch in der Bildung ist Wirtschaftlichkeit ein Kriterium, nämlich wenn man der Frage nachgeht, wieviele Fehlermöglichkeiten eine

Sprache bei verschiedenen Konstrukten enthält. Hier ist eine knappe Sprache wie Python deutlich im Vorteil. Auch hierzu gibt es in oben genanntem Artikel von Ferg einige Beispiele.

Für den Unterricht, sei es in der Schule oder sei es an der Hochschule, stellen sich die Fragen nach der Lernkurve (steil oder flach), nach der Modellhaftigkeit der Sprache und der Mächtigkeit der Sprache. Die Mächtigkeit der Sprache ist allerdings in der Schule von untergeordneter Bedeutung. Jede Programmiersprache ermöglicht es, alle Probleme, die etwa in einem 2- oder 4-stündigen Kurs auftreten, zu lösen.

2.5.3. Entscheidung

Hier sollte sich jeder Lehrende kurz überlegen, wie er vorgehen würde, wenn er seinen Schülern erklären sollte, was die Befehle bedeuten. Würde er seinen Schülern sagen: **#include<iostream>**, **#include<string>**, **using namespace std;**, **cin** usw. ignorieren wir einfach mal, das kommt später? Oder würde er wirklich die einzelnen Sprachelemente erklären? Wie lange bräuchte er wohl dazu? (Eine kleine Entscheidungshilfe, aber leider nur für die Sprache Java: In einem der Standardbücher zu Java, in [10] steht ein ähnliches Programm-Fragment, das dieses Problem löst, auf Seite 612.) Wenn er die erste Methode wählt: wieviele Zuhörer würden Programmierung als etwas sehen, das in der Nähe von obskuren Voodoo-Kulten anzusiedeln ist, aber für Normalsterbliche völlig unverständlich ist. Und wenn er die zweite Methode wählt: wieviele Programmieranfänger würden diesen Erklärungen gerne folgen, sie verstehen und sagen: mehr davon!

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]

2.5.4. Schön einfach? Einfach ist schön

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

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

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.

Die Programmierung graphischer Oberflächen erfolgt heute im kommerziellen Bereich in der Regel mit Hilfe sogenannter „Interface-Builder“ . Auch wenn es Sinn machen kann, [8] 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!** [9]

2.5.5. Lernen durch Nachmachen? Lernen durch Selbertun?

> Man kann eine Sprache erst dann, wenn man in ihr kommunizieren kann.
>
> Man kann eine Programmiersprache erst dann, wenn man selbständig in ihr Programme schreiben kann.

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.

Wer Programmieren lernt, der wird Probleme als solche erkennen lernen. Fortschritte dabei bedeuten, dass man immer mehr Lösungsstrategien sich erarbeitet und verstehen kann. Nach einer Weile hat man dann einen ganzen Stapel Vorgehensmuster, unter denen man dann bei einem neuen Problem einige auswählen kann, die Effizienz beurteilen kann und das Problem löst, indem man aus seinem Erfahrungsschatz das Richtige auswählt.

> Bereits beim Zusammenspiel zwischen Variablen und Schleifen zeigen sich die Probleme vieler SchülerInnen[10] im Umgang mit Steuerstrukturen.

Ein einfaches Beispiel dazu: Wie bestimmt man das Maximum aus einer Menge von fünf eingegebenen Zahlen?[11]

---

[8] das ist auch nicht von mir; ich weiß, dass es in gutem Deutsch „Auch wenn es sinnvoll wäre, . . . “ heißt

[9] aus: http://www.b.shuttle.de/b/humboldt-os/python/

[10] Ganz ehrlich: ich war das nicht! Das große „I“ ist im Originaltext, und bei dem Zitat muss ich das übernehmen.

[11] aus: http://de.wikibooks.org/wiki/Programmieren_leicht_gemacht_-_adäquate_Modelle_für_den_Einsatz_im_Unterricht#Einleitung

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.

Listing 2.8: Viele Parameter in Python

```
>>> print(max(3,4,12,67,34,8))
67
```

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.

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.

Teil II.
Grundlagen

3. Was ist Python?

Always look on the bright side of life.

(Eric Idle [1])

3.1. 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.** „Die Ritter der Kokosnuss" oder „Das Leben des Brian" kennt wahrscheinlich wirklich jeder, genauso wie wohl die meisten die bekanntesten Schauspieler dieser so britischen Komikertruppe kennen: John Cleese, Michael Palin, Eric Idle.

Wie Mark Lutz in seinem Buch „Programming Python" [31, S. 15] 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. [2]

[1] Life of Brian

[2] Bild lizensiert unter

Abbildung 3.1.: Monty Python

3.2. 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
- 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 ein 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

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

Überzeugt?

Dazu schreibt Mark Lutz: Mancher stellt da die Frage „Wer benutzt überhaupt Python?“ Aber die richtige Frage ist „Wer benutzt denn Python nicht?“ [31, S. 9]

3.3. 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 [38, S. 13] beginnt: „Python-Programme kann man lesen!“ Damit sind Python-Programme gut nachvollziehbar, damit gut zu verändern und zu verbessern (Wartbarkeit).

   Für den Programmieranfänger ist das nicht einzusehen, dass das tatsächlich ein wichtiges Kriterium ist. Wer länger programmiert, weiß es: ein Programm wird einmal geschrieben, hundertmal getestet, tausendmal verbessert und verändert. Die Zeitersparnis, die sich ergibt, wenn ein Programm gut lesbar ist und damit beim ersten Durchlesen (oder auch beim zweiten) verständlich ist, ist enorm, und Zeit ist Geld.

2. Python-Code unterstützt die Wiederverwertbarkeit von Code durch Objektorientiertheit und die Aufteilung von Code in Module.

   Das Argument aus dem vorigen Punkt der Aufzählung gilt auch hier wieder: time is money.

3. Python-Code ist plattformunabhängig. Ein auf einem Rechner geschriebenes Programm in Python ist meistens ohne Veränderung auf einem anderen Rechner lauffähig.

   Selbstverständlich müssen die betriebssystembedingten Befehle jeweils angepasst werden, so zum Beispiel Dateinamen, Verzeichnisnamen, die Trennzeichen zwischen Dateinamen und Verzeichnisnamen und vieles mehr. Aber auch dazu gibt es in Python Bibliotheken, die das machen. Wahrscheinlich wird es vielen Programmierern so gehen, dass sie, wenn sie solche „Systemaufrufe“ in Python benutzen, es sich durch den Kopf gehen lassen, ob sie einen für alle Betriebssysteme gültigen Code schreiben könnten, und stellen dann fest: ja, in Python könnte ich das wahrscheinlich selber schreiben.

4. Python bringt schon eine große Zahl von Bibliotheken mit, viele weitere können aus dem Internet besorgt werden. „Batteries included“ ist ein immer wieder gehörter oder gelesener Spruch, wenn es um Python geht. Das meiste, was man für die tägliche Arbeit braucht, ist im Standard-Umfang der Sprache dabei.

5. Python-Code ist im Verhältnis zu Java-Code relativ kurz, im Verhältnis zu perl-Code sehr klar. Wie Lutz in [31], 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 auf S. 25 zum Vergleich verschiedener Sprachen wie perl, Python oder 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.
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.
11. Seit kurzer Zeit ist ein weiterer wichtiger Punkt dazugekommen: der Raspberry Pi. Wer sich diesen Zigarettenschachtel-PC kauft, will ihn auch bedienen. Der wird in Python programmiert.

(Womit ich es geschafft habe, von der seriösen Zahl 10 (10 Gebote . . .) wegzukommen, und auf der närrischen Zahl 11 gelandet bin. Geschrieben im Juli 2014, zu der Zeit der Monty Python Reunion Show.)

Inzwischen habe ich von C. Severance das Buch „Python for Informatics“ durchgeblättert. In seinem Vorwort schreibt er:

> In 2003 I started teaching at Olin College and I got to teach Python for the first time. The contrast with Java was striking. Students struggled less, learned more, worked on more interesting projects, and generally had a lot more fun.

Bill Lubanovic schreibt in seinem „Introducing Python“

> Python is the most popular language for introductory computer science courses at the top American colleges.

Warum nicht in Deutschland?

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!!

3.4. 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[4] und das Kompilat wird dann ausgeführt. Die unterschiedliche Verarbeitung wird in dem Bild dargestellt.

[4]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!

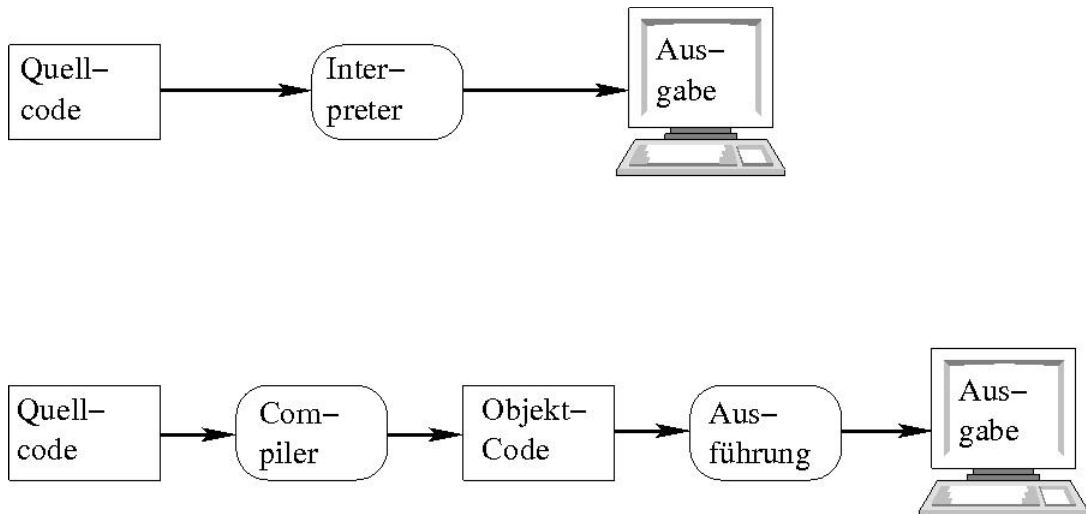


Abbildung 3.2.: Interpreter und Compiler

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.

```
#include "stdio.h"

main() {
    printf("Hallo Welt");
}
```

```
7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
02 00 03 00 01 00 00 00 10 83 04 08 34 00 00 00
94 17 00 00 00 00 00 00 34 00 20 00 09 00 28 00
25 00 22 00 06 00 00 00 34 00 00 00 34 80 04 08
34 80 04 08 20 01 00 00 20 01 00 00 05 00 00 00
04 00 00 00 03 00 00 00 54 01 00 00 54 81 04 08
54 81 04 08 13 00 00 00 13 00 00 00 04 00 00 00
01 00 00 00 01 00 00 00 00 00 00 00 00 80 04 08
00 80 04 08 c0 04 00 00 c0 04 00 00 05 00 00 00
00 10 00 00 01 00 00 00 14 0f 00 00 14 9f 04 08
14 9f 04 08 04 01 00 00 08 01 00 00 06 00 00 00
00 10 00 00 02 00 00 00 28 0f 00 00 28 9f 04 08
28 9f 04 08 c8 00 00 00 c8 00 00 00 06 00 00 00
04 00 00 00 04 00 00 00 68 01 00 00 68 81 04 08
68 81 04 08 20 00 00 00 20 00 00 00 04 00 00 00
04 00 00 00 04 00 00 00 88 01 00 00 88 81 04 08
88 81 04 08 18 00 00 00 18 00 00 00 04 00 00 00
```

Abbildung 3.3.: 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 [5] , kann man diesen Interpreter sofort aufrufen.[6] Das machen wir jetzt! Nachdem der Interpreter sich mit dem Python-Prompt (»>) arbeitsbereit gemeldet hat, geben wir dem Interpreter die erste Aufgabe:

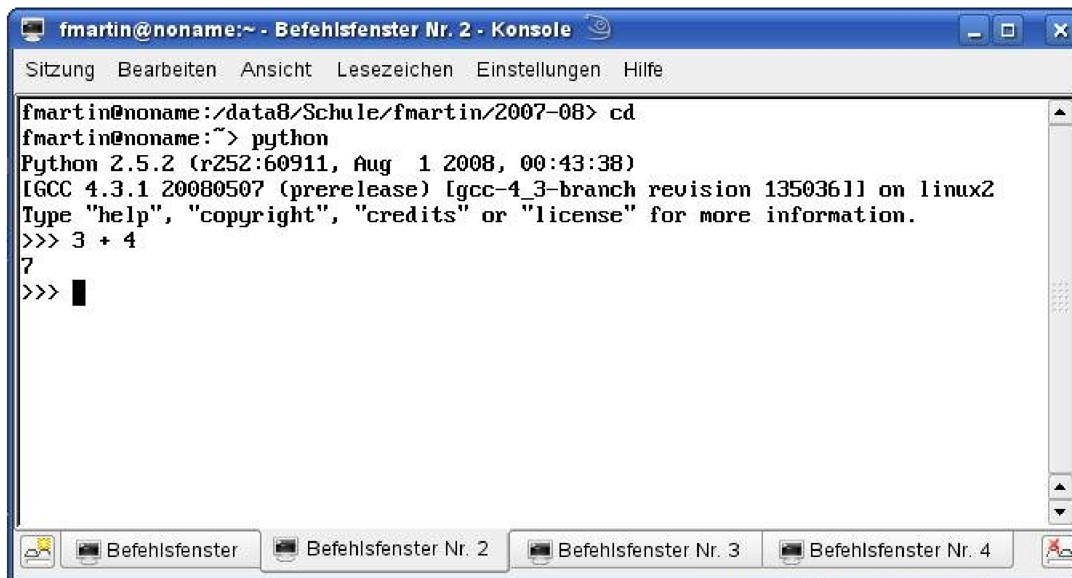


Abbildung 3.4.: Interpreter

Brav, der Interpreter hat die Zeile richtig gelesen und die Aufgabe zu unserer Zufriedenheit gelöst. Was passiert aber, wenn wir etwas ganz anderes, weniger sinnvolles, schreiben, oder mit Monty Python „And now for something completely different“ ?

Listing 3.1: Fehlerhafte Ausgabe eines Textes in python

```
1 >>> The Spanish Inquisition
2     File "<stdin>", line 1
3     The Spanish Inquisition
4               ^
5     SyntaxError: invalid syntax
```

Python beschwert sich hier, weil hier etwas auftaucht, was es nicht erwartet.[7] Diese 3 Wörter gehören nicht zum Sprachumfang von Python.

[5] das heißt auch, dass die Pfade alle richtig gesetzt sind
[6] 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.
[7] Wer Monty Pythons Flying Circus gesehen hat, weiß es: „Nobody expects the Spanish Inquisition!“

4. Programmieren

Computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity,
and especially because it produces objects of beauty.

(Donald E. Knuth[1])

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[2])

4.1. 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).[3] 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.[4]

[1] zitiert nach: http://en.wikiquote.org/wiki/Donald_Knuth
[2] zitiert nach: http://en.wikiquote.org/wiki/C._A._R._Hoare
[3] 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.
[4] So wie ich das gerade auch mache (obwohl Programmieren mir jetzt mehr Spaß machen würde).

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“ .[5] 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.

Ein Algorithmus besteht aus einzelnen Anweisungen. Oft benützt man statt Anweisung das Wort „Befehl“ oder das englische Wort „statement“ Ein Befehl ist eine Wortfolge, die aus Wörtern besteht, die der Computer, genauer die Programmiersprache, kennt, die nach Regeln aneinandergereiht werden, die den Regeln der Programmiersprache genügen.

4.1.1. Warum soll man überhaupt programmieren lernen?

> Most of the good programmers do programming not because they expect to get paid or get adulation by the public, but because it is fun to program.
>
> *(Linus Torvalds[6])*

Für das Lernen einer natürlichen Sprache gibt es meistens ganz natürliche Gründe: man lernt Spanisch, weil man im nächsten Urlaub nach Spanien fahren will, man lernt Französisch, weil man Sartres „Huis Clos“ im Original lesen will, man lernt Italienisch, weil man eine nette Italienerin oder einen netten Italiener kennengelernt hat, mit dem man sich auch manchmal in deren bzw. dessen Sprache unterhalten möchte.

Aber was gibt es für Gründe, eine Programmiersprache zu lernen?

- Will ich den Computer bedienen oder beherrschen?
- Will ich verstehen, wie ein so komplexes Programm wie eine Textverarbeitung aufgebaut ist?
- Will ich wissen, warum ein Computer kein Schwäbisch versteht? Und warum er überhaupt keine natürliche Sprache versteht?
- Habe ich ein spezielles Problem, von dem ich „im Prinzip “ weiß, wie man das lösen könnte, aber noch nirgends eine Lösung für den Computer gefunden habe?

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

[6] zitiert nach: https://en.wikiquote.org/wiki/Linus_Torvalds , dort: Rishab Aiyer Ghosh, interviewer (1998-03-02). First Monday Interview with Linus Torvalds: What motivates free software developers?

- Muss ich am Computer immer wieder die selben stupiden Arbeiten erledigen, von denen ich denke, dass sie so stupide sind, dass der Computer das alleine können sollte?

Vielleicht weil ich eine dieser Fragen beantworten will!

Wenn man dann einen Einstieg gefunden hat, fallen einem auch ganz schnell Dinge ein, die für einen Menschen langweilig sind, weil sie die Intelligenz eines Menschen unterfordern, um nicht zu sagen: manche Arbeiten sind eine Beleidigung für intelligente Menschen. Maschinen sind da nicht so schnell beleidigt, und vor allem sind Maschinen auch nicht dadurch aus der Fassung zu bringen, dass man sie auffordert, immer wieder das selbe zu machen.

Nehmen wir ein zugegeben sehr unrealistisches Beispiel (obwohl vielleicht die Linguisten das nicht uninteressant finden): suche in einem etwas längeren Text, nehmen wir also mal von Marx „Das Kapital“ oder die Bibel, das Wort mit 5 Buchstaben, das am häufigsten auftritt. Wer wollte das von Hand machen? Und wer wäre sicher, dass er keinen Fehler, weder beim Buchstabenzählen der einzelnen Wörter noch beim Zählen der verschiedenen 5-buchstabigen Wörter macht?

Nach einigen Stunden Programmierung wäre das Problem selbst für einen Anfänger kein Problem mehr. Es wäre vielleicht noch eine Herausforderung, aber in Python ist ein Programm, das genau die oben geschilderte Aufgabe löst

- kurz
- schnell zu schreiben
- und absolut sicher

Dem Computer ist es dabei völlig egal, welche Bedeutung der Text hat, den er gerade verarbeitet, „Das Kapital“ und die Bibel sind für ihn nur Mengen von Zeichen, die er nach bestimmten von uns vorgegebenen Regeln zu durchsuchen hat. Er wird nicht müde dabei, ihn langweilt es nicht, er muss keine Pausen machen, er arbeitet einfach, bis er fertig ist.

Beim Schreiben des Programms tut man etwas sehr menschliches, man trainiert seine Intelligenz, und während das Programm läuft, kann man etwas machen, was dem Menschen angemessener ist, als auf Hunderten von Seiten Buchstaben zu zählen (übrigens nicht sehr lange, denn das Programm wäre sehr schnell mit seiner Arbeit fertig).

Und man ist kreativ! Wir erinnern uns vielleicht an die ersten Holzbauklötzchen, kleine, bunte Quader, mathematisch eine ziemlich langweilige Form. Trotzdem konnten wir als kleine Kinder damit Häuser, Drachen, Ritter, Bäume und vieles anderes bauen. Die Phantasie macht es! Später kamen dann Lego-Steine, die flexibler waren und besser zusammenhielten, und auch hier konnte man mit einfachen Formen recht komplexe Gebilde machen.[7] Programmieren ist irgendwie ähnlich: aus einfachen (programmier-)sprachlichen Strukturen kann man mit viel Kreativität komplexe Programme schreiben.

Programmieren lehrt Lernen. Wo sonst kann ich Fehler machen, die zwar gleich bestraft werden (dadurch, dass das Programm nicht läuft), aber die ich verbessern kann, ohne

[7] Ob die Lego-Baukästen, die heutzutage viele Spezialsteine enthalten, dafür aber nur noch ermöglichen, ein bestimmtes Objekt zu bauen, die Kreativität genauso fördern?

dass jemand anderer mich deswegen tadelt. Dabei lerne ich, dass Lernen nicht nach dem ersten Versuch, ob erfolgreich oder nicht, endet, sondern dass Lernen daraus besteht, etwas aufzunehmen, es anzuwenden, festzustellen, dass ich es doch noch nicht beherrsche, und es dann nochmals zu versuchen.

Kapiert? Fehler sind in Ordnung!!! ... wenn man bereit ist, sie zu verbessern.[8]

4.1.2. Sprache lernen

Eine Programmiersprache zu lernen ist gar nicht so verschieden vom Lernen einer natürlichen Sprache. Man muss Vokabeln lernen und die Grammatik beherrschen. Das Vokabelnlernen ist in einer Programmiersprache aber schön überschaubar: schau doch mal bei den reservierten Wörtern nach, wie viele Vokabeln das sind! Die Grammatik ist ein bißchen aufwendiger. Hier geht es darum, zu lernen, wie man in der Programmiersprache korrekte „Sätze“ bildet. Das wird mit „Syntax“ bezeichnet, und hier ist die große Hürde, dass die Syntax einer Programmiersprache im Gegensatz zu der einer natürlichen Sprache keine Varianten zulässt. Wenn in einer natürlichen Sprache die Syntaxregel „Subjekt - Prädikat - Objekt“ gilt, dann ist ein „Satz“ zwar nicht korrekt, aber oft noch verständlich, der sich nicht an diese Regel hält. Ein Satz in einer Programmiersprache, der sich nicht an eine Syntax-Regel hält ist falsch **und** unverständlich, was bedeutet, dass ein Programm mit einem solchen falschen Satz einfach nicht ausgeführt wird.

4.1.3. Übersetzung ... in welche Sprache denn?

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.

Man kann es auch so beschreiben: nachdem ein Compiler den Quellcode in ein ausführbares Programm umgewandelt hat, kann man dieses Programm immer wieder laufen

[8] Ja, ich bin für Verbesserungen bei Klassenarbeiten, Hausaufgaben etc.!!!

lassen, ohne es neu zu kompilieren, auf jeden Fall auf dem selben Computer, aber auch auf jedem Rechner mit dem selben Betriebssystem. Ein interpretiertes Programm benötigt aber für jeden weiteren Programmlauf wieder den Interpreter, um den Quellcode Zeile für Zeile in einen ausführbaren Befehl umzuwandeln.

4.2. 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.

4.3. 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.

4.4. 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.[9] „Freie Software“ hat etwas mit Freiheit zu tun, nicht mit dem Preis. Um das Konzept zu verstehen, ist an „frei“ wie in „freier Rede“[10] und nicht wie in „Freibier“ zu denken.[11]

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 [12] muss Python gesondert besorgt werden.

Falls dieser Text auf CD zu Dir gekommen ist, ist die Chance groß, dass auch in einem

[9] hierzu sollte man sich das gute Buch von Grassmuck[16] besorgen, das — das Thema erfordert es ja schon — auch fast „frei“ ist, nämlich bei der Bundeszentrale für politische Bildung für einen geringen Unkostenbeitrag zu erhalten ist. siehe: `www.bpb.de`

[10] 2.

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

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

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.

4.5. 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.

Wohl aber 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.

Zwei Vorschläge mache ich trotzdem:

1. atom erfüllt diese Bedingungen, und wer seinen Editor nach seinen Bedürfnissen einrichten möchte, liegt hier richtig. [13]

[13] Nicht nur, wenn sie oder er programmieren will.

2. 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 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.

4.6. Voraussetzungen

4.6.1. 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 [14] 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 [31] auf Seite 266,

> In the first edition of the book *Learning Python*„, 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

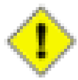
[14] Glück gehabt! Das dummdeutsche Wort [24] „Benutzeroberfläche" konnte ich im Text gerade noch vermeiden. Der Benutzer bin ich, meine Oberfläche nenne ich Haut, und was die auf einem Computer-Bildschirm soll, habe ich noch nie verstehen können.

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

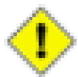


Wer unter Windows ernsthaft arbeiten will, sollte sich von Microsoft nicht bevormunden lassen und sofort den Schalter umlegen, um grundsätzlich alle Dateiendungen anzeigen zu lassen.

4.6.2. 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[15] 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.

[15] also der Einstellung der Sprache der Darstellung

4.6.3. 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?)[16]
- 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.

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 verz-name	md verz-name
Verzeichnis löschen	rmdir verz-name	rmdir verz-name
Link auf eine Datei anlegen	ln originalName neuerName	
Zugriffsberechtigung lesen	ls -l dateiname	attrib dateiname
Zugriffsberechtigung setzen	chmod xxxx dateiname	attrib xxxx dateiname

Tabelle 4.1.: Betriebssystem-Befehle

[16]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.

4.7. Was heißt „Programmieren“ ? Fortsetzung!

> Without a program, a computer is an overpriced door stopper.
>
> *(David Evans [17])*

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.

4.7.1. Regeln für die Sprache Python: PEP8

Das Nachschlagewerk für Regeln, die man bei der Programmierung einhalten sollte, ist PEP8 (von van Rossum[37] und von Reitz[36]). Die wichtigsten Regeln für Programmier-Anfänger sind hier zusammengetragen:

- In Python sind Leerzeichen(engl.: whitespace) syntaktisch relevant. Python Programmierer sind besonders empfindlich gegenüber den Auswirkungen von whitespace auf die Klarheit eines Programms.
- Benutze Leerzeichen anstelle von Tabulatoren für die Einrückung. [18]
- Für jede Ebene der syntaktisch relevanten Einrückung benutze 4 Leerzeichen
- Die Länge einer Zeile sollte maximal 79 Zeichen betragen.
- Fortsetzungszeilen sollten wieder durch 4 zusätzliche Leerzeichen eingerückt werden.
- In einer Datei sollten Funktionen und Klassen durch 2 leere Zeilen voneinander getrennt sein.
- Innerhalb einer Klasse sollten Methoden durch eine Leerzeile voneinander getrennt sein.
- Setze keine Leerzeichen vor und nach Listen-Indices, Funktionsaufrufe oder Schlüsselwort-Argumente.
- Setze genau ein Leerzeichen vor und nach einer Variablen-Zuweisung.
- Funktionsnamen, Variablennamen und Attribute sollen im mit Kleinbuchstaben beginnen, Teile des Namens durch Unterstriche voneinander getrennt sein.
- Klassennamen und Ausnahmen sollen im GrossbuchstabenWort-Format geschrieben werden.

[17] aus:[11], S. 35

[18] Viele IDE erlauben die Tabulatortaste, die dann Leerzeichen generiert.

4.7.2. Strukturierte Programmierung: Ein Überblick

> Lerne gehen, bevor du rennen lernst.
>
> *(unbekannt)*

Gerade im Hinblick auf Programmieranfänger, vor allem Schüler, wurde die objektorientierte Programmierung als die Errungenschaft gefeiert, die einen intuitiveren Zugang zur Programmierung verschafft und so die Einstiegshürden senkt. Leider wurde dabei übersehen, dass das beste (objektorientierte) Paradigma nichts nützt, wenn Lernende nicht wissen, was an der Basis passiert: was eine Variable ist, was „WENN - DANN" bedeutet, was eine Wiederholung ist und vieles mehr. Nein, das ist noch nicht die Basis: dazu gehört auch, dass man weiß, was Bedingungen sind und wie man diese formuliert, wie man einer Variablen einen Wert zuweist, wie eine Wiederholung beendet wird usw.

Lehrer, die an Schulen unterrichten, wissen, was ich meine. Da wird etwa in der Mittelstufe ein schlichtes Anwendungsprogramm zur Tabellenkalkulation benutzt, und man muss froh sein, wenn die Schüler mit viel Anleitung verstehen, wie man darin die „WENN-DANN" -Funktion benutzt. Und jeder Lehrer weiß, wieviel davon im darauffolgenden Schuljahr noch vorhanden ist.

Ich bin überzeugt, dass der Einstieg in die (objektorientierte) Programmierung nicht geht, wenn man nicht die oben beschriebenen Grundlagen beherrscht: wenn man nicht die elementaren Regeln der strukturierten Programmierung gelernt hat. Und so ist dieses Buch auch aufgebaut: die Objektorientierung kommt ziemlich weit hinten.

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.[19] Dies wird in der Datenverarbeitung eine Schleife, mit einem Fremdwort Iteration 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

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

intelligent! Diese Entscheidung wird in der Programmierung eine Alternative genannt.

Das sind also die Elemente der strukturierten Programmierung:

- die Sequenz
- die Alternative
- die Iteration

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 `gegenkatete`, `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, welche Bedeutung die Variable 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:

Listing 4.1: Kommentar

```
1 # zz ist der Zeilenzaehler fuer die Matrix
2 # sz ist der Spaltenzaehler fuer die Matrix
3 zz = 1
4 sz = 1
```

Eine besonders hilfreiche Eigenschaft von Kommentaren in Python 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.

Listing 4.2: Hilfetext

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

    return zahl * zahl
```

Der Hilfetext kann jetzt auf zwei Arten aufgerufen werden:

1. Mit help(quadrate) wird ein gegliederter Hilfetext ausgegeben. Das sieht so aus:

   Listing 4.3: Ausgabe eines Hilfetextes

   ```
   >>> help(quadrate)
   Help on function quadrate in module __main__:

   quadrate(zahl)
       Der Funktion quadrate wird als
       Parameter eine Zahl mitgegeben,
       die Funktion liefert das Quadrat der
       Zahl zurueck
   ```

2. Mit der print-Funktion, indem man den Namen der Funktion, gefolgt von zwei Unterstrichen, gefolgt vom Text `doc`, gefolgt von nochmals zwei Unterstrichen eingibt.

   Listing 4.4: Ausgabe eines Hilfetextes mit print

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

Die **print**-Funktion wird erst weiter hinten auf Seite 73 angesprochen, aber es ist wahrscheinlich klar, was sie macht: sie gibt etwas aus, in diesem Fall auf den Bildschirm.

Und die Funktion funktioniert so wie gewünscht:

Listing 4.5: Beispiel für eine einfache Funktion

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

4.8. Objektorientierung

Dieses Kapitel greift vor: über Objektorientierung wird erst wieder im Kapitel 12 gesprochen. Aber bereits in den Kapiteln zuvor wird immer wieder eine Notation benutzt, die aus dem Bereich der Objektorientierung stammt.

In (fast) allen Programmiersprachen muss man beim Anlegen einer Variablen angeben, welchen Typ diese Variable hat. Man muss etwa festlegen, dass die Variable `zahl1` eine Variable des Typs `int` ist, wobei `int` die Abkürzung für das englische Wort für „ganze Zahl“ nämlich `integer` ist; oder eine `vorname` soll vom Typ `string` sein, also eine Zeichenkette zum Inhalt haben.

In Python muss das nicht festgelegt werden. Bei der ersten Wertzuweisung an eine Variable merkt Python selbst, zu welcher Klasse (gemerkt? nicht zu welchem Typ) diese Variable gehört. Klassen sind Modelle für Variablen, die nicht nur bestimmte Eigenschaften haben, sondern auch festgelegte Fähigkeiten; das heißt, dass durch die Klassenzugehörigkeit einer Variablen festgelegt wird, was diese Variable kann und darf.

Die Eigenschaften bzw. die Fähigkeiten einer Variablen, also eines Objektes einer Klasse, werden durch einen Punkt an den Namen der Variablen angehängt. Ein Beispiel: wenn ich also eine Variable mit Namen `name` habe, die den Wert „Meier“ enthält, ist dies ein Text, und dieser Text hat die Fähigkeit, sich selber in reinen Großbuchstaben schreiben zu lassen:

Listing 4.6: Texte: groß schreiben

```
1 >>> name = 'meier'
2 >>> name.upper()
3 'MEIER'
```

Fähigkeiten sind Funktionen, und die Schreibweise ähnelt auch der Schreibweise einer Funktion in der Mathematik: hinter den Funktionsnamen wird dort in Klammern die Variable der Funktion angegeben. Das ist hier auch so, aber die Funktion `upper` benötigt keine Variable, also bleibt hier die Klammer leer.

4.9. Programmierstil und Konventionen

> The ideal programmer would have
> the vision of Isaac Newton,
> the intellect of Albert Einstein,
> the creativity of Miles Davis,
> the aesthetic sense of Maya Lin,
> the wisdom of Benjamin Franklin,
> the literary talent of William Shakespeare,
> the oratorical skills of Martin Luther King,
> the audacity of John Roebling,
> and the self-confidence of Grace Hopper.
>
> *(David Evans [20])*

Solange man an seinem eigenen Rechner sitzt und nur für das eigene Vergnügen programmiert, kann man das machen, wie man will. Viele Programmierer sind auf diese Art groß geworden. Wenn man aber aus seinem stillen Kämmerlein herauskommt, wird das eigene Werk auf einmal kritisch beäugt. Und das ist gut so. Auch wenn D. Knuth sein Werk „The Art of Computer Programming“ genannt hat, sollte man sich nicht zuviel künstlerische Freiheiten herausnehmen — vor allem dann, wenn man unter künstlerischer Freiheit hauptsächlich Chaos und Unordnung versteht. Jede Kunst besitzt auch Struktur.

Struktur entsteht, indem man das Handwerkszeug beherrscht, das für die Beherrschung der Kunst(-richtung) nötig ist. So sollte man sich nach den ersten paar Programmen diese noch einmal anschauen und sich die Variablennamen vornehmen: sind die alle nach einer Regel (die ich mir selber gegeben habe und in Worte fassen kann) aufgebaut? Oder gibt es da Inkonsistenzen in der Schreibweise? Sind logische Zusammenhänge durch die Schreibweisen sichtbar gemacht?

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. In jedem Text in 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:

[20] [11], S. 35

Listing 4.7: 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.[21]

Dieses perl-Programm ist lauffähig. [22] Aber wirklich nicht gut lesbar. Eine Gliederung ist nicht zu erkennen. In Python ginge so etwas nicht!

[21] Ist es auch. Die Gliederungsmerkmale sind die Strichpunkte und die geschweiften Klammern.
[22] Perl: The only language that looks the same before and after RSA encryption. – Keith Bostic

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 der Variablenname von Variablen, 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

Da Python-Entwickler und Python-Programmierer eine gewisse Neigung zu dummen Sprüchen haben[23], 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.

4.10. 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. [24] 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 lässt man sich anzeigen durch

Listing 4.8: Befehl, um reservierte Wörter anzuzeigen

```
1 >>> import keyword
2 >>> print(keyword.kwlist)
3 ['False', 'None', 'True',
4 'and', 'as', 'assert', 'break', 'class', 'continue',
5 'def', 'del', 'elif', 'else', 'except',
6 'finally', 'for', 'from', 'global',
7 'if', 'import', 'in', 'is', 'lambda',
8 'nonlocal', 'not', 'or', 'pass', 'raise',
9 'return', 'try', 'while', 'with', 'yield']
```

[23] Du weißt noch, woher die Sprache ihren Namen hat?

[24] 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

Wir, die Programmierer, erweitern durch unsere Programme das Vokabular, indem wir neue Vokabeln für neue Sachverhalte erfinden.

4.11. Fehler (zum ersten)

> She knows there's no success like failure
> And that failure's no success at all
>
> *(Bob Dylan [25])*

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 „erhoehen“ 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.

Was also in einem Aufsatz die Note 2-3 ergibt, führt in der Programmierung zur Note „ungenügend“.

Jetzt also zu den verschiedenen Arten von Fehlern:

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

- 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.

4.12. 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.

4.13. . . . und das fehlt

Vor allem jüngere Lernende vermissen vielleicht Spiele. Auch hierzu gibt es natürlich viele schöne Beispiele in Python, denn Python ist auch eine Sprache, in der man gut Spiele programmieren kann. Aber die Programmierung von Spielen setzt viel mehr voraus, als hier in diesem Skript beschrieben wird. Man wird auch kaum ein sinnvolles Buch zum Einstieg in die Programmierung finden, das sich mit Spiele-Programmierung befasst. Wer sich trotzdem darin probieren will, sollte zuerst auf grafische Oberflächen verzichten. Ein Vorschlag, den man in Erwägung ziehen könnte, wenn man die ersten objektorientierten Programme erstellt hat, ist „Schiffe versenken“ . Try it!

4.14. 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 III.

IDE

5. Die Entwicklungsumgebung IDLE: Zahlen und Variable

> A ship in port is safe;
> but that is not what ships are built for.
> Sail out to sea and do new things.
>
> *(Admiral Grace Hopper [1])*

5.1. ... 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.

5.2. 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

[1] (Erfinderin der Programmiersprache COBOL)

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 5.1.: Eric Idle (CC BY 2.0)

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

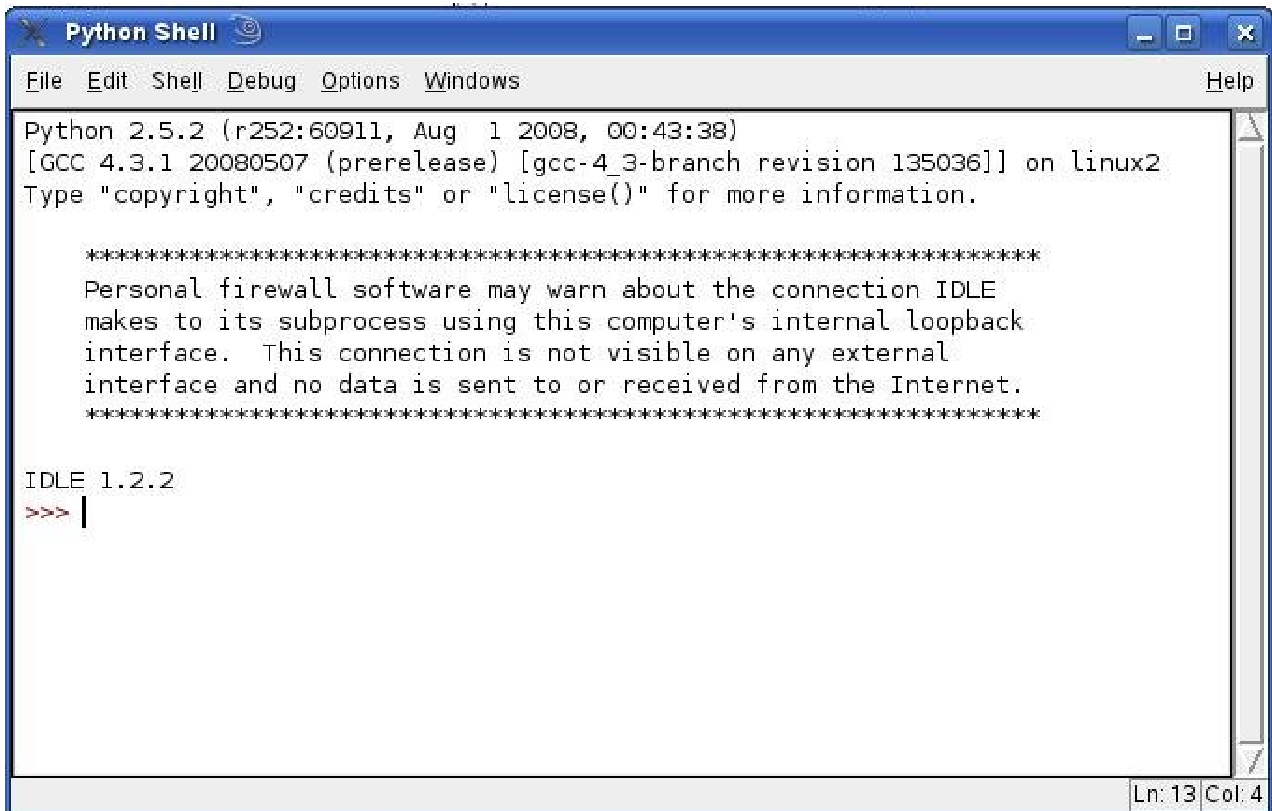


Abbildung 5.2.: Die IDE Idle

Jetzt geben wir einfach mal etwas in IDLE ein, in Erinnerung dessen, dass ein Computer ein Rechner ist, eine komplizierte Mathematik-Aufgabe. Zuerst müssen den Variablen, hier der Einfachheit halber `x1` und `x2` genannt, Werte zugewiesen werden. Der Zuweisungsoperator in Python ist das Gleichheitszeichen.

Listing 5.1: Zuweisungsoperator

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

Jede einzelne Zeile, die wir oben geschrieben haben, ist eine Anweisung (oder ein Befehl) in Python. Manchmal wird dafür auch das englische Wort „statement“ benutzt. Damit wird Python gesagt: mach dies, mach das. Die obigen 3 Zeilen geben Python genau 3 Befehle, und diese Befehle heißen in menschlicher Sprache:

1. Gib einem Speicherplatz den Namen `x1` und weise diesem Speicherplatz den Wert 3 zu.
2. Gib einem anderen Speicherplatz den Namen `x2` und weise diesem Speicherplatz den Wert 4 zu.

3. Gib einem weiteren Speicherplatz den Namen `ergebnis` und weise diesem Speicherplatz die Summe der Werte zu, die auf den Speicherplätzen mit den Namen `x1` und `x2` gespeichert sind.

Das ist jeweils eine sehr umständliche Sprech- und Schreibweise. Man drückt das deswegen oft einfacher aus durch:

1. Weise der Variablen `x1` den Wert 3 zu.
2. Weise der Variablen `x2` den Wert 4 zu.
3. Weise der Variablen `ergebnis` die Summe der Werte der Variablen `x1` und `x2` zu.

Das IDLE-Fenster sieht danach so aus:

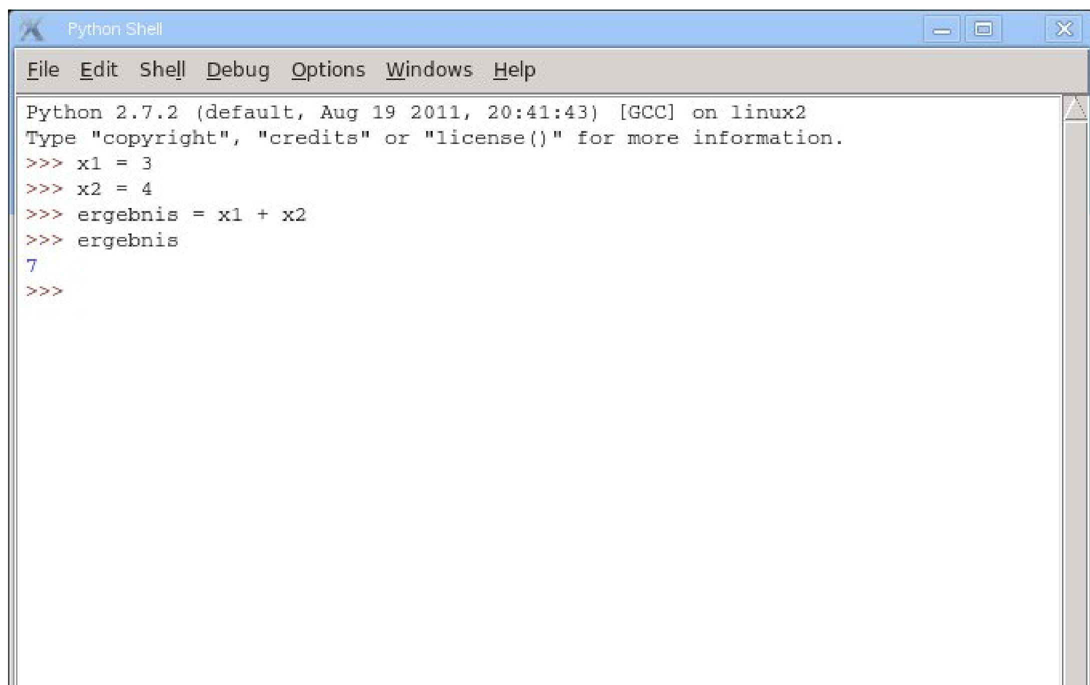


Abbildung 5.3.: Erste Berechnung in der IDE Idle

Es ist wohl fast selbstverständlich, wie eine Anweisung abgearbeitet wird: von links nach rechts; dabei gelten aber die Regeln, die aus der Mathematik bekannt sind: was in Klammern steht, wird zuerst verwertet, danach kommen (im Falle von mathematischen Anweisungen) die „Punkt-“ Rechnungen, dann die „Strich-“ Rechnungen.

Anmerkung

Wer sich für diese schrittweise Ausführung von Befehlen interessiert, aber auch für alle Anfänger, für die das oben beschriebene völliges Neuland ist: es gibt eine nette Entwicklungsumgebung mit Namen `thonny`, die genau das macht. Diese IDE ist auch freie Software und existiert wohl für alle Betriebssysteme.

Im Menu unter `Run` → `Debug(Nicer)` werden die einzelnen Aktionen von Python Schritt für Schritt angezeigt.

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.[2]

Das Ergebnis überrascht uns nicht. Der Rechner kann rechnen.

[2] 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!!!

Abbildung 5.4.: Rechnungen in Idle

Wenn man im obigen Bild genau hinschaut, erkennt man 2 Dinge:

1. In der ersten steht die Version des Python-Interpreters. Hier ist es Version 3.6.5. Die Division von 9 durch 3 ergibt 3.0 Das Ergebnis einer Division ist immer eine Division.
2. Die ganzzahlige Division, die als Ergebnis auch eine ganze Zahl liefert und diese beim Dezimalkomma abschneidet wird durch den doppelten Schrägstrich erledigt. (Wie damals in der Grundschule: $9 : 4 = 2$ (Rest 1))

In Python 2 sähe das etwas anders aus. Die Division von 9 durch 3 ergibt hier das Ergebnis 3 (als ganze Zahl). Also stellen wir hier einen gravierenden Unterschied zwischen Python 2.x und Python 3.x fest. Das stelle ich hier in einer Tabelle gegenüber:

Python 3.x	Python 2.x

`>>> 9 / 3`
3.0
`>>> 9 // 4`
2
`>>> 9 / 4`
2.25
	`>>> 9 / 3`
	3
	`>>> 9 / 4`
	2
	`>>> 9.0 / 4`
	2.25

Der Divisionsoperator / steht für die Division von beliebigen Zahlen. Das Ergebnis ist immer eine Dezimalzahl. Der Divisionsoperator // steht für die Division von ganzen Zahlen. Das Ergebnis ist immer eine ganze Zahl.	Der Divisionsoperator / steht für die Ganzzahl-Division. Wenn Divisor und Dividend ganze Zahlen sind, ist das Ergebnis eine ganze Zahl. Andernfalls ist das Ergebnis eine Dezimalzahl (zur Erinnerung: Divisor: die Zahl, durch die geteilt wird; Dividend: die Zahl, die geteilt wird)

Tabelle 5.1.: Division in Python 3.x bzw. Python 2.x

Wenn wir einer Variablen einen Wert zuweisen, kann es passieren, dass wir sehr bald nicht mehr wissen, ob wir eine ganze Zahl oder eine Dezimalzahl zugewiesen haben. Das können wir in Python aber leicht rauskriegen.

Listing 5.2: Zahlentypen

```
1 >>> x = 3
2 >>> y = 123.156156
3 >>> type(x)
4 <class 'int'>
5 >>> type(y)
6 <class 'float'>
```

Und manchmal ist es nötig, eine Variable, die den Typ `int` hat, in den Typ `float` umzuwandeln, oder umgekehrt.

Listing 5.3: Umwandlung in einen anderen Zahlentyp

```
1 >>> x_float = float(x)
2 >>> y_int = int(y)
3 >>> print(x_float)
4 3.0
5 >>> print(y_int)
6 123
```

Eine wichtige Kleinigkeit muss an dieser Stelle festgehalten werden: die Grundlage bei der Entwicklung von Python (wie von fast allen Programmiersprachen) ist englisch, also gibt es kein Dezimalkomma, sondern einen Dezimalpunkt.

Anmerkung

In Python 3 ist die Division (ausgeführt durch den Schrägstrich /) immer eine Fließkomma-Division. Wenn man in Python 3 eine Ganzzahl-Division durchführen will, also eine Division, bei der das Ergebnis eine ganze Zahl ist, muss man den Doppel-Schrägstrich // benutzen.

Das probieren wir gleich aus, indem wir in die entsprechenden Befehle in IDLE eingeben:

Listing 5.4: Division

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

Ebenso gilt:

Listing 5.5: Division (Forts.)

```
>>> x1 = 3
>>> x2 = 4
>>> ergebnis = x2 / x1
>>> ergebnis
1.3333333333

```

Für die Division, wie sie in der Grundschule gelernt wird, nämlich

Listing 5.6: Division mit Rest

```
4 : 3 = 1 (Rest 1)
```

muss eingegeben werden:

Listing 5.7: Ganzzahlige Division

```
>>> 4 // 3
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:

Listing 5.8: Modulus

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

und das liefert uns den Rest 2 bei der Division von 8 durch 3.

Das, was wir im vorigen Absatz Python vorgeworfen haben, ist kein Befehl. Es ist ein Ausdruck, auf englisch eine „expression“. Auf der Python-Konsole, im Python-Interpreter, kann man auch Ausdrücke auswerten lassen, ohne einen Befehl auszuführen. Das macht man manchmal, um zu testen, welchen Wert ein Ausdruck hat. Allerdings ist der Wert des Ausdrucks sofort verloren, wenn ich etwas anderes mache, da er nicht gespeichert wird, nicht ausgegeben wird ... Deswegen lasse ich Ausdrücke so gut es geht weg.

Python bietet uns sogar noch mehr:

Listing 5.9: divmod

```
1 >>> divmod(13,3)
2 (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.

Operation	math. Zeichen	Beispiel math. Schreibweise	Beispiel in Python	Ergebnis
Addition	+	6 + 3	6 + 3	9
Subtraktion	-	6 - 3	6 - 3	3
Multiplikation	·	6 ·3	6 * 3	18
Division	:	6 : 3	6 / 3	2
Modulus	mod	8 mod 3	8 % 3	2
Exponentiation	Hochstellung	2^3	2 ** 3	8

Tabelle 5.2.: Arithmetische Operatoren in Python

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**. **print** ist eine Funktion in Python, und Funktionen werden geschrieben, wie das aus der Mathematik bekannt ist. $f(x) = 2x^2 - 3$ ist eine solche Funktion aus der Mathematik: vorne steht der Funktionsname, das f, gefolgt von ein Klammer, in der die Argumente der Funktion stehen, das (x) ... und in Mathematik, zumindest in der Schule, steht dann nach dem Gleichheitszeichen der Funktionsterm.

Anmerkung

In Python2 war **print** noch keine Funktion, sondern ein Ausdruck. Dort mussten die Argumente von **print** nicht in Klammern geschrieben werden. In Python3 ist das jetzt viel ordentlicher!

In Python ist das genauso, bis zu den 3 Punkten „. . . " im vorigen Satz. **print** nimmt die Argumente, die in der Klammer stehen, wandelt jedes einzelne in eine Zeichenkette (einen String) um, setzt die einzelnen Zeichenketten zusammen und gibt diese zusammengesetzte Zeichenkette aus. Dabei wird die Zeichenkette „vollständig" , das heißt, dass ein Zeilenvorschub hinzugefügt wird. [3] Dieses Verhalten, dass jeder Aufruf der **print**-Funktion eine vollständige Zeile liefert, kann man dadurch ändern, dass man der **print**-Funktion den Parameter **end=' '** mitgibt. Hier ist der Parameter standardmäßig mit einem Zeilenvorschub **'\n'** belegt.

Das scheint keinen Unterschied zu machen, ob man in IDLE

```
1 >>> 3 + 4
```

oder

```
1 >>> print(3 + 4)
```

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

Listing 5.10: Rundungsfehler

```
1 >>> 2 / 5.0
2 0.40000000000000002
3 >>> print(2 / 5.0)
4 0.4
```

Man sieht: die Ausgabe mit „print" rundet intelligent! [4]

Klar: wenn ich zu der Zahl 0 in einer Schleife 10 mal 0,1 addiere, ergibt das 1. Aber nicht beim Programmieren mit Dezimalzahlen. (Ein kleiner Vorgriff auf Programmschleifen! Geduld, die kommen in einem der nächsten Kapitel!)

[3] Das entspricht einem **println** in Pascal.

[4] 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.

Listing 5.11: Dezimalzahlen und Rundungsfehler

```
1  >>> summe = 0
2  >>> for i in range(10):
3  ...     summe += 0.1
4  ...
5  >>> print(summe)
6  0.9999999999999999
```

Das sollte man sich immer klar machen, wenn man viele Rechnungen mit Dezimalzahlen zu machen hat.

5.2.1. Ein Vorgriff: Decimal

Python wäre nur halb so schön, wenn es nicht eine Möglichkeit gäbe, diese Rundungsfehler zu unterbinden: das klappt mit dem Import von `Decimal` aus dem Modul `decimal`. Die Dezimalzahl 0.1 wird dann als Decimal('0.1') geschrieben; das verwirrt im ersten Augenblick, weil das nicht mehr wie eine Zahl, sondern wie ein Text aussieht, aber diese Objekte verhalten sich wie exakte Dezimalzahlen.

Listing 5.12: Dezimalzahlen ohne Rundungsfehler

```
1  >>> from decimal import Decimal
2
3  >>> summe = 0
4  >>> for i in range(10):
5          summe += Decimal('0.1')
6  >>> print(summe)
7  1.0
```

Wenn man mit Brüchen rechnet, passiert das nicht. Auf Seite 308 kommt weiter hinten in diesem Kapitel das Thema Bruchrechnen, und darin wird dieses nochmals mit Brüchen gerechnet.

5.2.2. Aufgaben zu Zahlen und elementaren Berechnungen

1. Weise mindestens 5 Variablen irgendwelche numerische Werte zu, und zwar sowohl ganze Zahlen als auch Dezimalzahlen. Führe alle möglichen Rechenoperationen mit diesen Zahlen aus, gib die Ergebnisse aus und halte Auffälligkeiten in den Ergebnissen fest.

5.3. Ein Speicherplatz mit einem Namen: Variable

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

(Bob Dylan [5])

5.3.1. 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 Wert wird einer Variablen zugewiesen durch den Zuweisungsoperator „=“ . Durch `x = 3` wird der Variablen `x` der Wert 3 zugewiesen. (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. Dazu später mehr, wenn Objektorientierte Programmierung besprochen wird.) Vorerst benutzen wir nur einfache Typen, das heißt Zahlen-Typen und Text-Typen.

Man kann sich eine Variable wie einen Karton vorstellen, der ordentlich beschriftet ist. Die Beschriftung gibt dabei an, was in dem Karton ist. Damit man sich das nicht nur vorstellen muss, sondern auch sehen kann, gibt es hier ein Bild: der Karton ist mit dem Namen `zahl` versehen und hat als Inhalt einen Zettel, auf dem der Wert „15“ steht.

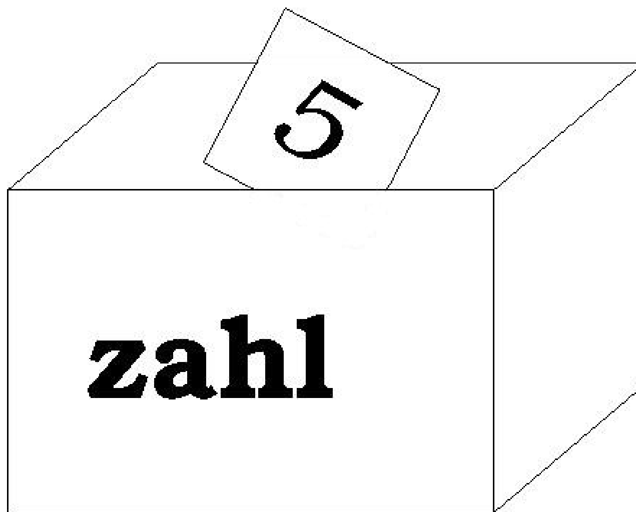


Abbildung 5.5.: Ein Variablen-Karton mit Inhalt

Die Beschriftung sollte nicht nur gut lesbar sein, sondern auch etwas über den Inhalt

[5] Man gave names to all the animals *auf:* Slow Train Coming

aussagen. Stell Dir vor, Du hast vor ein paar Monaten Deinen Keller aufgeräumt, alles was herumliegt in Kisten gepackt und diese alle fein säuberlich beschriftet: „Kruscht“ steht auf allen drauf!!! Und jetzt such mal in den Kisten den USB-Stick mit den Bildern von der Geburtstagsfeier: er ist in einer der Kisten, aber wo?

Das im vorhergehenden Abschnitt beschriebene Verhalten von Python wird in der Informatik mit dem Begriff „dynamische Typisierung“ bezeichnet. Bei der dynamischen Typisierung wird einer Variablen ein Typ zugeordnet, und damit verhält sich die Variable so wie es in der Typ-Beschreibung vorgegeben ist. Das hat gewisse Vorteile, weil sich der Programmierer zum Beispiel keine Gedanken darüber machen muss, wie groß mögliche Werte für diese Variable sein können (so kann man also einer Variablen `textbeispiel` sowohl den Wert `Python` als auch den Wert `Donaudampfschifffahrtsgesellschaftskapitän` zuweisen). Python kümmert sich automatisch darum, dass ausreichend Speicher zur Verfügung gestellt wird und dass, nachdem die Variable ihren Dienst getan hat, auch der Speicher wieder freigegeben wird.[6] Auf jeden Fall muss der Programmierer hier darauf achten, dass zum Beispiel in der Variablen `erg` das Ergebnis `2.71828` steht. Das Ergebnis `Always look on the bright side of life` ist vielleicht als mentale Aufmunterung erfreulicher, aber mit diesem Ergebnis kann niemand, auch nicht Python, weiterrechnen.

Das Gegenteil der dynamischen Typisierung ist die „statische Typisierung“ . Hier muss der Programmierer von vorneherein festhalten, welche Art von Information in einer Variablen gespeichert werden soll (eine Zahl, ein Text, eine Liste, eine Matrix), und meistens auch noch, wieviel Platz diese Information (maximal) einnehmen kann.

Wir müssen noch festhalten, dass in der Informatik Ausdrücke benutzt werden, die einen Mathematiker den Kopf schütteln lassen: es kann nämlich auf der rechten und der linken Seite des Zuweisungsoperators, also des Gleichheitszeichens, die selbe Variable stehen. In Mathematik ist das folgende sinnlos:

Listing 5.13: zweimal dieselbe Variable

```
1 x = 7
2 x = 2 + x
```

In Mathematik widersprechen sich die beiden Anweisungen: in der ersten steht, dass die Variable `x` den Wert 7 hat, die zweite Anweisung ist für keinen Wert von `x` richtig.

In der Programmierung ist das sehr sinnvoll. In der ersten Anweisung wird der Variablen `x` der Wert 7 zugewiesen, d.h. ab jetzt hat die Variable `x` den Wert 7. In der zweiten Anweisung steht (in einer Art Pseudocode formuliert): nimm das, was in der Variablen `x` steht, zähle 2 dazu und weise das Ergebnis der Variablen `x` zu; ab jetzt hat also `x` den Wert 9.

Einem Mathematiker oder Programmierer erscheint es selbstverständlich, aber es wird trotzdem hier erwähnt: man kann einer Variablen beliebig oft einen Wert zuweisen.

Listing 5.14: Mehrere Werte einer Variablen

```
1 >>> x = 7
```

[6] Ob das ein Vorteil oder ein Nachteil ist, darüber können sich Programmierer prima streiten, vor allem zwischen Mötzingen und Baisingen.

```
2 >>> print(x)
3 7
4 >>> x = 2 + 7
5 >>> print(x)
6 9
```

Python erlaubt auch die Mehrfachzuweisung, und zwar in zwei Varianten.

- Es ist möglich, mit einer Anweisung mehreren Variablen den selben Wert zuzuweisen.

Listing 5.15: Mehrfachzuweisung eines Wertes an mehrere Variable

```
1           \xlabel[Mehrfachzuweisung]{lst:mehrfZuw}
2 >>> x = y = z = 7
3 >>> print(x)
4 7
5 >>> print(z)
6 7
7 >>> print(y)
8 7
```

- Mit einer Anweisung kann auch verschiedenen Variablen verschiedene Werte zugewiesen werden.

Listing 5.16: Mehrfachzuweisung mehrerer Werte an mehrere Variable

```
1 >>> a, b, c, d = 3,7,5,9
2 >>> print(a)
3 3
4 >>> print(b)
5 7
6 >>> print(c)
7 5
8 >>> print(d)
9 9
```

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.

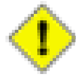
In der (Schul-)Mathematik kennt man in der Analysis eine Variable, die dann fast immer `x` heißt, ganz selten `t`, wenn man Funktionen in der Physik untersucht, bei der die Zeit variabel ist. In der Linearen Algebra gibt es dann in der Schule schon mal 3 Variable, die dann `x`, `y` und `z` heißen, wenn 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`.

5.3.2. 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 zwar eine Variable `länge` nicht ab; besser ist die Variable `laenge`. Denn Python arbeitet mit UTF-8, aber wer weiß, ob ein Entwickler, der mein Programm weiterbearbeitet, auch schon diesen Standard auf seinem Rechner hat.

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. [7] Andererseits ist es auch für 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.

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

Tabelle 5.3.: Variablennamen

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. Am besten mache ich das mit einem prägnanten Beispiel.

[7] siehe hierzu auch PEP 8

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 eines (sehr schlichten) Betriebssystems aus einem inzwischen sehr großen Softwarehaus von der Westküste der USA[8] versuchte man, uns weiszumachen, dass die Datei mit dem Namen `liebe.txt` und die Datei mit dem Namen `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.

5.3.3. 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.

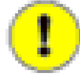
In fast allen Programmiersprachen, auch in Python, ist der Zuweisungsoperator das „=“, das Gleichheitszeichen.[9] Links des Gleichheitszeichens steht der Name der Variablen, der etwas zugewiesen wird, rechts des Gleichheitszeichens steht der Wert, der zugewiesen wird. **a = 17** bedeutet also, dass der Variablen mit dem Namen `a` der Wert 17 zugewiesen wird.

[8] schon rausgekriegt, welches ich meine?

[9] 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

Gültige Zuweisungen sind also:

Listing 5.17: Korrekte Zuweisungen

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

Ungültige Zuweisungen (die dann zu Fehlermeldungen führen) sind:

Listing 5.18: Fehlerhafte Zuweisungen

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

5.3.4. Aufgaben zu Variablen

> And there shall in that time
> be rumours of things going astray,
> and there will be a great confusion
> as to where things really are.
>
> *(Monty Python [10])*

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

[10] in: Life of Brian

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?

5.3.5. Zuweisungsmuster

Aus einer netten Internetseite von John C. Lusth[11] 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:

   Listing 5.19: Transfer-Muster
   ```
   1 >>> a = 5
   2 >>> 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.

   Listing 5.20: Veränderungs-Muster
   ```
   1 >>> a = 5
   2 >>> 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.

[11] . . . die nicht mehr existiert

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

Listing 5.21: Wegschmeiß-Muster

```
1 >>> a = 5
2 >>> 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.

6. Richtig programmieren

6.1. Entwicklungsumgebungen

6.1.1. 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:

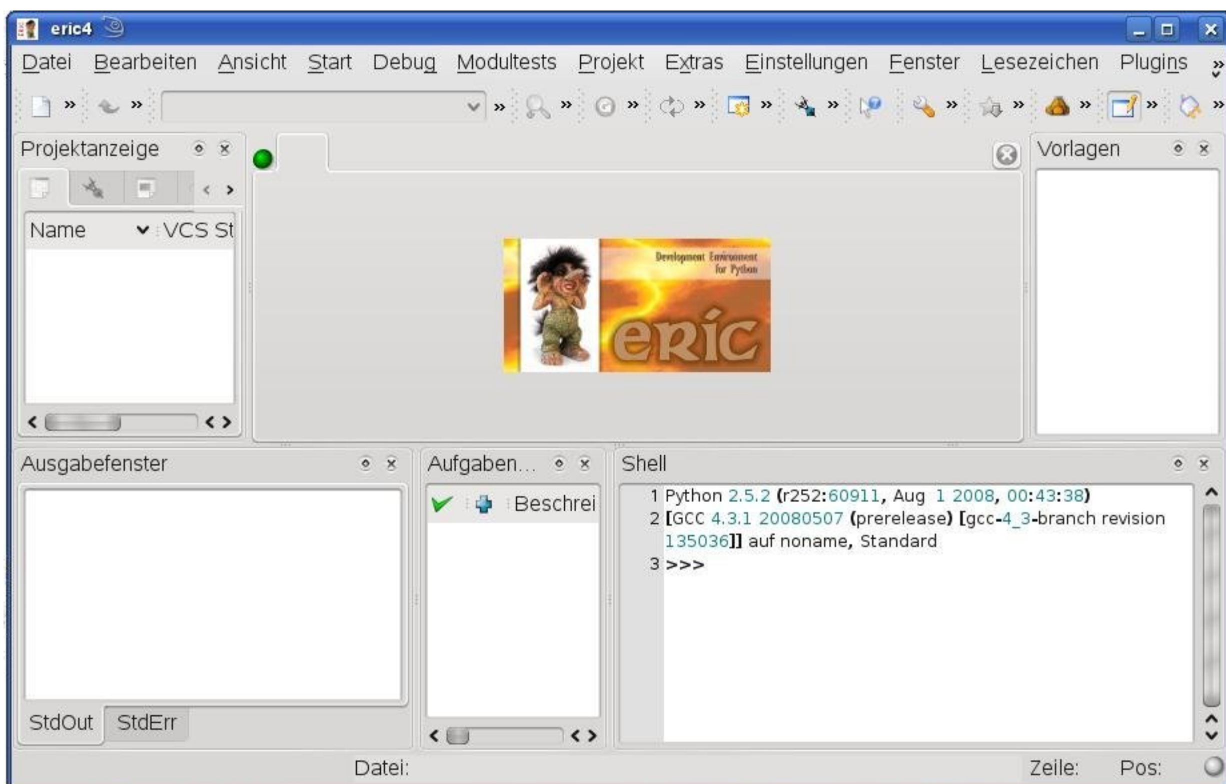


Abbildung 6.1.: Die IDE Eric

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.

6.1.2. 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:

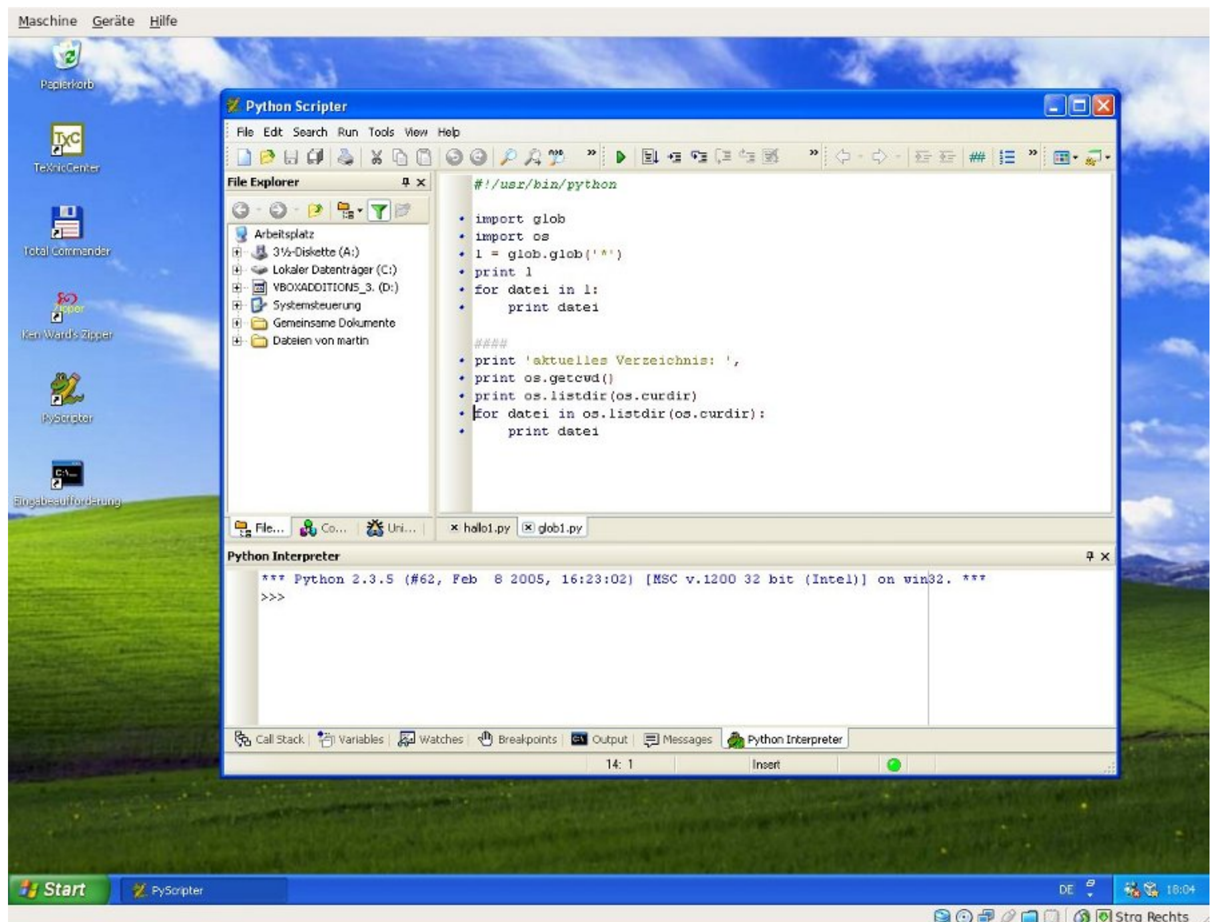


Abbildung 6.2.: Die IDE PyScripter

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.

6.1.3. 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! 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.

6.1.4. 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 einem beliebigen Betriebssystem lauffähig sind, sollten wir uns hier bemühen, das Programm auch so zu schreiben, dass das gewährleistet ist.[1] 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:

Listing 6.1: Sha-Bang

```
1 #!/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:

Listing 6.2: Programmaufruf mit Umgebungsvariable

```
1 #!/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(zur Zeit noch; aber Windows soll das irgendwann auch einmal verstehen) ignoriert, stört also in keinem Fall.

[1] 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.

6.2. 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:

Listing 6.3: Hallo world

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 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 „utf-8“ eingestellt. Unter Windows [2] ist es oft sinnvoller, die Kodierung auf iso8859-1 oder latin1 umzustellen.

Anmerkung

Unter Python 3.x hat sich das geändert. Hier wird als Standard-Kodierung „utf-8“ 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.

[2] wie immer zeitlich ein bißchen hinterher

6.3. 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")
```

Texte müssen in Anführungsstriche geschrieben werden. „input“ fasst die Eingabe als Text auf. Soll das, was eingegeben wird, als Ganzzahl oder als Fließkommazahl verwendet werden, so muss das umgewandelt werden durch `int(input(...))` bzw. durch `float(input(...))`.

Listing 6.4: Saftladen

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  print("Saftladen\n\n")
5  prApfelschorle = 0.70
6  prBananenmilch = 1.20
7  prOrangensaft = 0.80
8
9  anzAschorle = int(input('Wieviel Apfelschorle? '))
10 anzBmilch = int(input('Wieviel Bananenmilch? '))
11 anzOsaft = int(input('Wieviel Orangensaft? '))
12
13 preis = anzAschorle * prApfelschorle + anzBmilch * prBananenmilch
14         + anzOsaft * prOrangensaft
15 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:

Listing 6.5: Rechnen mit Uhrzeiten

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  min1 = int(input('erste Minuten-Zahl: '))
5  min2 = int(input('zweite Minuten-Zahl: '))
6
7  sum = min1 + min2
8  std = sum // 60
9  min = sum % 60
10
11 print(std,' Stunde(n) ',min,' Minuten')
```

Teil IV.

Texte und andere Daten

7. Texte

And now for something
completely different

(John Cleese [1])

7.1. 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.

Die wichtigste Regel für Zeichenketten ist, dass Zeichenketten in Anführungszeichen geschrieben werden. Texte nicht in Anführungszeichen zu schreiben 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" '
```

Eine andere Möglichkeit, einfache Anführungsstriche in einem Text, der von doppelten Anführungsstrichen eingeschlossen ist, zu benutzen (oder umgekehrt), ist das „Maskieren“/ von Anführungsstrichen:

```
>>> text = "Gasthaus zum \"Goldenen Ochsen\""
>>> text
'Gasthaus zum "Goldenen Ochsen"'
```

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

```
4 >>> print(text)
5 Gasthaus zum "Goldenen Ochsen"
```

Maskiert wird ein Zeichen durch das Voranstellen eines Backslashes.

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

```
1 >>>sinn = '''Always look
2 on the bright
3 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

```
1 >>> sinn
```

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

```
1 >>> print(sinn)
```

Zeichenketten (auch Strings genannt) sind wohlgeordnet: ich kann bestimmen, was das 3. Zeichen in der Zeichenkette ist, was das erste, was das letzte. Die Position innerhalb der Zeichenkette bezeichne ich auch gerne als die Hausnummer des Zeichens. Ein bestimmtes Zeichen einer Zeichenkette wird dadurch ausgewählt, dass ich in eckigen Klammern seine Hausnummer angebe.

```
1 >>> text = 'Martin'
2 >>> text[2]
3 'r'
4 >>> text[0]
5 'M'
6 >>> text[-1]
7 'n'
```

Wie man hier sieht (und das gilt in Python für alle Objekte, die man abzählen kann) zählt Python von der Null an. Das 0-te Element der obigen Zeichenkette ist das „M“ . Das letzte Element einer Zeichenkette ist das von hinten gesehen erste. Das wird geschrieben als Hausnummer[-1

Strings sind unveränderlich (engl. immutable), d.h. sie können nicht am ursprünglichen Speicherplatz verändert werden. Das funktioniert also nicht:

```
1 >>> text = 'Martin'
2 >>> text[2]
3 'r'
4 >>> text[2] = '?'
```

Probiere es selbst aus ... und lies und verstehe die Fehlermeldung!

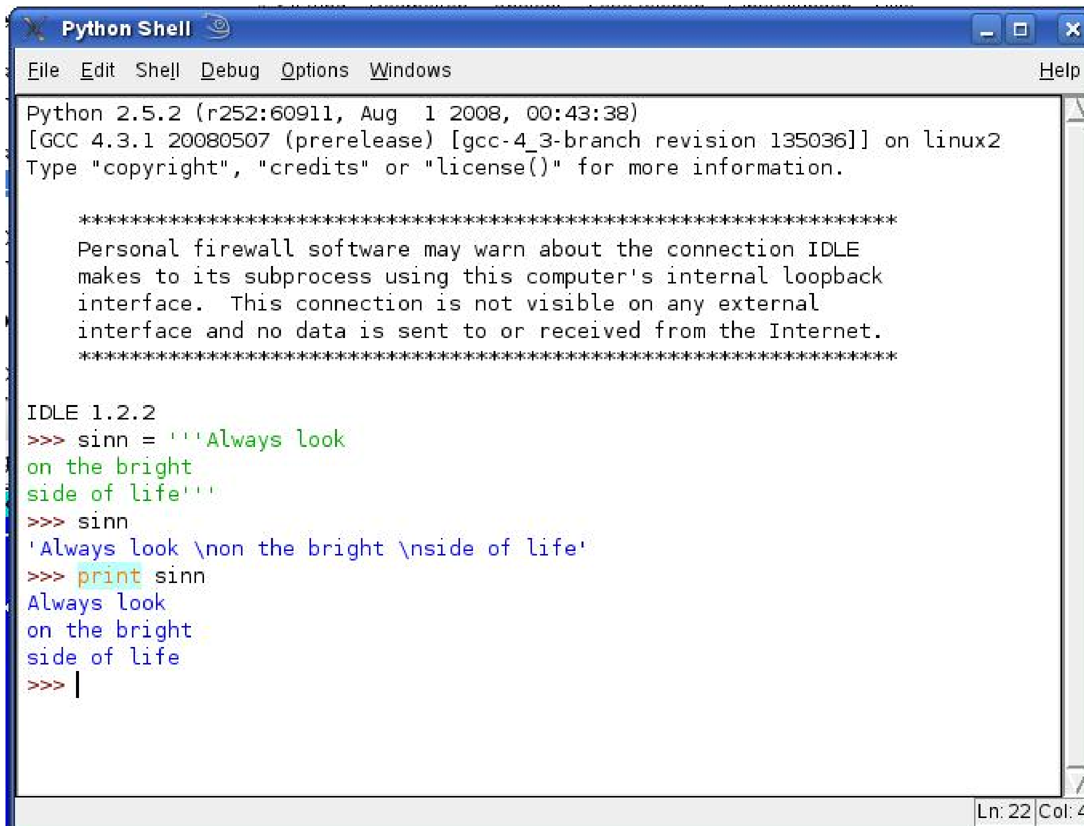


Abbildung 7.1.: Texte in der IDE Idle

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.

Listing 7.1: Zeilenvorschub (Steuerzeichen)

```
1  >>> vierZ = 'Das\nsind\n4\nZeilen'
2  >>> vierZ
3  'Das\nsind\n4\nZeilen'
4  >>> print(vierZ)
5  Das
6  sind
7  4
8  Zeilen
```

Die wichtigsten Steuerzeichen stehen in der folgenden Tabelle:

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

Tabelle 7.1.: Steuerzeichen

Python kann auch mit Texten rechnen.[2] Was dabei herauskommt, wenn man Texte addiert oder einen Text mit einer Zahl multipliziert, sieht man hier

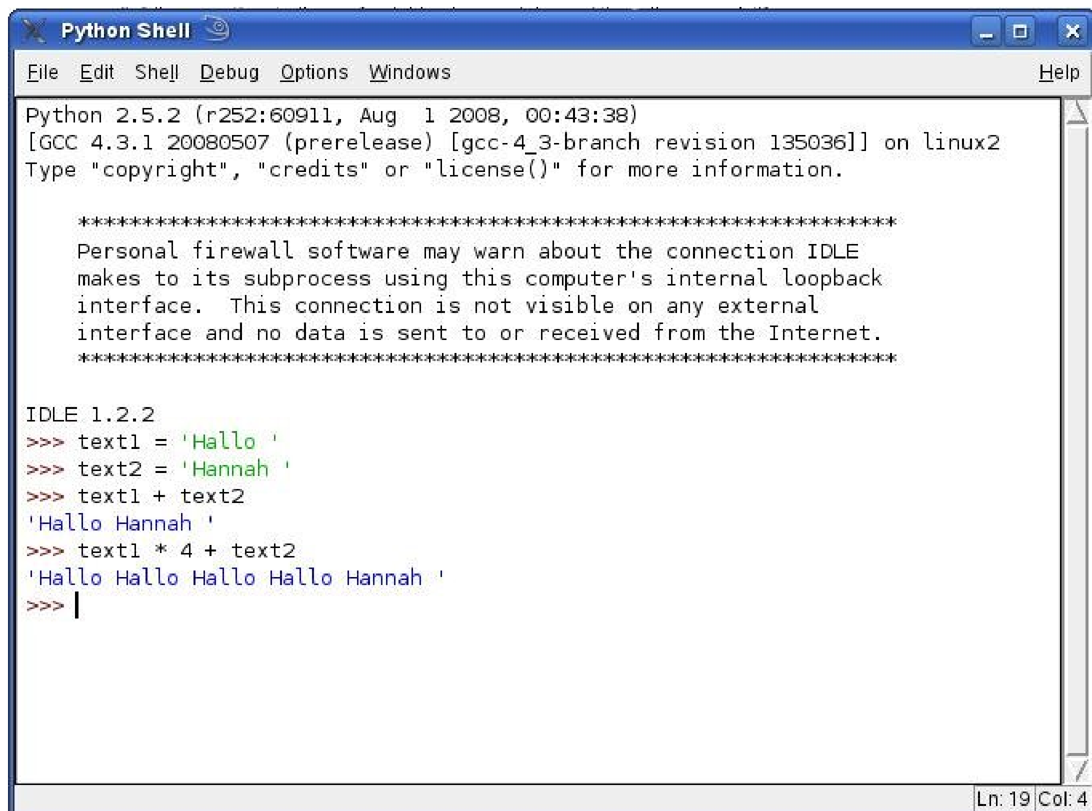


Abbildung 7.2.: Multiplikation von Texten

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

Listing 7.2: Bildschirm löschen

```
1 >>> print(40*"\n")
```

[2] Das hört sich seltsam an. Aber lies einfach weiter!

7.1.1. Lange Texte

Editoren oder IDEs haben einen bestimmte Breite der Darstellung von Ein- und Ausgabe eines Programms. Standard sind 80 Zeichen. Aber es kann passieren, dass Texte länger als 80 Zeichen sind. Es gibt zwei verschiedene Arten, Texte einzugeben, die länger als 80 Zeichen sind, und diese beiden Arten bewirken auch verschiedene Ausgaben.

Hier kommt eine Eingabe des obigen Absatzes in eine Variable und die Ausgabe dieser Variablen:

Listing 7.3: Langer Text ohne Umbrüche

```
1  >>> textOhneUmbruch = 'Editoren oder IDEs haben einen bestimmte Breite der Darstellung von
2  >>> print(textOhneUmbruch)
3  Editoren oder IDEs haben einen bestimmte Breite der Darstellung von Ein- und Aus
4  gabe eines Programms. Standard sind 80 Zeichen. Aber es kann passieren, dass Tex
5  te länger als 80 Zeichen sind. Es gibt zwei verschiedene Arten, Texte einzugeben
6  , die länger als 80 Zeichen sind, und diese beiden Arten bewirken auch verschied
7  ene Ausgaben.
```

Unschön!! Die Eingabe war in einer Zeile möglich, obwohl es mehr als 80 Zeichen waren; hier sieht man nur etwas mehr als 80 Zeichen, dann ist die Breite der Seite voll ausgeschöpft. Die Ausgabe hat stur 80 Zeichen pro Zeile geschrieben.

Ein längerer Text bei der Eingabe kann umbrochen[3] werden, indem man an das Ende einer geschriebenen Zeile einen Backslash „\“ anhängt. Das zeigt das nächste Beispiel:

Listing 7.4: Langer Text mit Umbrüchen durch einen Backslash

```
1  >>> textMitBackslashes = 'Editoren oder IDEs haben einen bestimmte Breite der \
2  Darstellung von Ein- und Ausgabe eines Programms.\
3  Standard sind 80 Zeichen.\
4  Aber es kann passieren, dass Texte länger als 80 Zeichen sind.\
5  Es gibt zwei verschiedene Arten, Texte einzugeben, die länger als 80 Zeichen sind,\
6  und diese beiden Arten bewirken auch verschiedene Ausgaben. '
7  >>> print(textMitBackslashes)
8  Editoren oder IDEs haben einen bestimmte Breite der Darstellung von Ein- und Aus
9  gabe eines Programms.Standard sind 80 Zeichen.Aber es kann passieren, dass Texte
10  länger als 80 Zeichen sind.Es gibt zwei verschiedene Arten, Texte einzugeben, d
11  ie länger als 80 Zeichen sind,und diese beiden Arten bewirken auch verschiedene
12 Ausgaben.
```

Die Eingabe konnte man besser lesen, die Ausgabe hat sich ein bißchen verschoben. Aber nach wie vor teilt Python den Text nach 80 Zeichen, egal wo man den Zeilenvorschub durch einen Backslash gemacht hat; der Backslash als Zeichen wird nicht ausgegeben.

Die zweite Möglichkeit, einen Umbruch zu bewerkstelligen, besteht darin, dass man den gesamten Text in dreifache Anführungsstriche setzt und manuell an den gewünschten Stellen umbricht.

[3] ein Begriff aus dem Buchsatz; umbrechen bedeutet, dass ein Text an einer sinnvollen Stelle in einer Zeile endet und in der nächsten Zeile sinnvoll weitergeht.

Listing 7.5: Langer Text in dreifachen Anführungsstrichen und mit manuellem Umbruch

```
        >>> textMitDreifachen = ''' Editoren oder IDEs haben einen bestimmte Bre
    der Darstellung von Ein- und Ausgabe eines Programms.
    Standard sind 80 Zeichen.
    Aber es kann passieren, dass Texte länger
    als 80 Zeichen sind.
    Es gibt zwei verschiedene Arten, Texte einzugeben,
    die länger als 80 Zeichen sind,
    und diese beiden Arten bewirken auch verschiedene Ausgaben. '''
>>> print(textMitDreifachen)
 Editoren oder IDEs haben einen bestimmte Breite
    der Darstellung von Ein- und Ausgabe eines Programms.
    Standard sind 80 Zeichen.
    Aber es kann passieren, dass Texte länger
    als 80 Zeichen sind.
    Es gibt zwei verschiedene Arten, Texte einzugeben,
    die länger als 80 Zeichen sind,
    und diese beiden Arten bewirken auch verschiedene Ausgaben.
```

Eingabe okay, Ausgabe okay!!!

7.1.2. Operationen auf Texten

And the first one now
Will later be last

(Bob Dylan [4])

Texte, also Zeichenketten, sind aus einzelnen Zeichen zusammengesetzt. Die dafür übliche Datenstruktur ist die „Liste“ (auch „das Feld“ genannt) auf englisch „list“ bzw. „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:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Wert	D	a	s		L	e	b	e	n		d	e	s		B	r	i	a	n	

Tabelle 7.2.: Zeichenkette als Liste betrachtet

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

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

Listing 7.6: 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 End-Element das erste Element ist, das nicht mehr ausgewählt wird.

Listing 7.7: Slicing

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

Für Texte nicht so interessant ist, dass man hier auch noch einen dritten Parameter mitgeben kann. Dieser gibt eine Schrittweite an, in der etwas aus einem Text herausgeschnitten werden soll.

Listing 7.8: Slicing (Forts.)

```
>>> filmDerWoche = 'Das Leben des Brian'
>>> print(filmDerWoche[1:9:2])
a ee
```

Zur Erläuterung: das 3. Zeichen ist ein Leerzeichen!! (Nicht vergessen: man fängt bei 0 an zu zählen!!)

Beachte dabei, dass man beim Slicing von vorne nach hinten die bis zu 3 Teile angeben muss. Wenn man für den ersten Teil keinen Wert (die Start-Hausnummer) angeben will, für die beiden anderen (die End-Hausnummer und die Schrittweite), muss das so geschrieben werden: `[:12:2]`, wenn man den gesamten Text (also Start-Hausnummer ist der Anfang, End-Hausnummer das Ende des Textes) mit einer festen Schrittweite ausgeben will, so: `[::2]`.

Am besten gefällt mir das folgende Problem: einen Text umdrehen (also von hinten nach vorne schreiben), das, was wir als kleine Kinder gespielt haben. „Wie heißt denn Du von hinten nach vorne?“ Mit Hilfe von Python ist das ganz einfach:

Listing 7.9: Umgekehrter Text

```
>>> vorname1 = 'martin'
>>> vorname2 = 'hannah'
>>> print(vorname1[::-1])
nitram
>>> print(vorname2[::-1])
hannah
```

Klar? Nimm den Text von Anfang bis Ende und gib ihn in der Schrittweite -1 aus.

7.1.3. Methoden von Zeichenketten

Der Begriff „Methode“ gehört noch gar nicht hier her. (Und um diesen Begriff auch nur ungefähr zu erklären brauche ich andere noch nicht geklärte Begriffe.) Methode bezeichnet eine Funktion innerhalb einer Klasse. Aber Klassen werden erst im Kapitel Klasse weiter hinten bearbeitet.

Ups! Auch der Begriff „Funktion“ gehört noch gar nicht hier her. Funktion bezeichnet ein Programm innerhalb eines Programms, ein Unterprogramm. Aber Funktionen werden erst im Kapitel 9.6 weiter hinten bearbeitet.

Methoden werden aufgerufen, indem man an den Klassennamen einen Punkt und dann den Methodennamen anhängt.

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.

Listing 7.10: Splitten

```
>>> name = 'Martin Schimmels Oberndorf'
>>> name.split()
['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!

Als Trennzeichen kann ein beliebiges Zeichen fungieren, sogar eine Zeichenkette ist möglich:

Listing 7.11: Splitten bei einer vorgegebenen Zeichenkette

```
>>> text = 'Martinxyzistxyzdoof'
>>> fuellzeichen = 'xyz'
>>> text.split(fuellzeichen)
['Martin', 'ist', 'doof']
```

Die Methode `join` macht das Splitten rückgängig.

Listing 7.12: Splitten und Wiederherstellen

```
>>> name = 'Martin Schimmels Oberndorf'
>>> liste = name.split()
>>> print(liste)
['Martin', 'Schimmels', 'Oberndorf']
>>> ' '.join(liste)
'Martin Schimmels Oberndorf'
```

Aber Vorsicht: `join` ist eine Methode,[5] die mit Texten arbeitet, nicht mit Listen. Deswegen steht ein Text, in diesem Fall der Text, der aus einem einzigen Leerzeichen besteht, voran, gefolgt von einem Punkt und dem Methodennamen. Das Argument der Methode ist in diesem Fall eine Liste.

[5] was das ist, kommt leider erst später, wenn ich Klassen einführe

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:

Listing 7.13: Suchen und ersetzen

```
>>> name = 'Martin Schimmels Oberndorf'
>>> name.find('Schi')
7
>>> name.replace('Oberndorf', 'Rottenburg')
Martin Schimmels Rottenburg'
>>> name
'Martin Schimmels Oberndorf'
>>> name.index('S')
7 # Beachte: man faengt bei 0 an zu zählen!
```

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**.

Listing 7.14: Enthaltensein

```
>>> name = 'Martin Schimmels Oberndorf'
>>> if 'mm' in name:
>>>     print('gefunden')
gefunden
```

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

Listing 7.15: Mehr mit Texten

```
>>> name = 'Martin Schimmels Oberndorf'
>>> name.upper()
'MARTIN SCHIMMELS OBERNDORF'
>>> name.lower()
'martin schimmels oberndorf'
>>> name.center(40)
'       Martin Schimmels Oberndorf       '
>>> name.center(40,'*')
'*******Martin Schimmels Oberndorf*******'
```

Wenn man mit Texten arbeitet, ist es oft nötig, „überflüssige“ Zeichen zu entfernen, so etwa alle möglichen Leerzeichen, auf englisch „whitespaces“, am Anfang oder am Ende eines Textes. Zu diesen Leerzeichen gehören

- das Leerzeichen
- der Tabulator
- der Zeilenvorschub

Dafür (und allgemeiner für die Entfernung beliebiger Zeichen) steht die Methode `strip` zur Verfügung. Ohne einen Parameter werden genau die oben genannten Leerzeichen entfernt, aber man kann auch einen oder mehrere Parameter in der Klammer angeben.

Listing 7.16: Strippen

```
1 >>> text = '\n     Hendrix:Voodoo Chile     \t'
2 >>> text.strip()
3 'Hendrix:Voodoo Chile'
4 >>> text = '****Hendrix:Voodoo Chile****'
5 >>> text.strip('*')
6 'Hendrix:Voodoo Chile'
7 >>> text = '*-*-*Hendrix:Voodoo Chile*-*-*'
8 >>> text.strip('*-')
9 'Hendrix:Voodoo Chile'
```

Ein Sonderfall für das Strippen ist das Entfernen eines Zeilenvorschubs am Ende einer Zeile. Das wird später beim Lesen aus einer Datei relevant. Es wird erledigt durch `zeile.rstrip()`

Weiter oben in der Tabelle Steuerzeichen 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.[6] Das bringt Probleme mit sich, wenn man aus einem Programm heraus auf einen vollqualifizierten Dateinamen zugreifen muss, also auf etwas in der Art

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

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

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

7.1.4. Wie und wo gespeichert wird

Bin ich? Und wenn ja, wieviele bin ich?

(frei nach R.D. Precht)

Geben wir folgende Anweisungen ein:

Listing 7.17: Kopie oder keine Kopie

```
>>> schlange = 'Python'
>>> sprache = 'Python'
>>> schlange == sprache
True
>>> schlange is sprache
True
>>> id(sprache)
140169203725008
>>> id(schlange)
140169203725008
```

Der erste Vergleich mit `==` stellt fest: ja, die beiden Variablen haben den selben Wert. Der zweite Vergleich mit `is` stellt fest: das ist sogar das selbe Objekt, auf das jetzt nur zwei verschiedene Zeiger weisen. Mittels der Funktion `id` sieht man jetzt, dass `schlange` und `sprache` tatsächlich die selbe Adresse haben.

7.2. Kodierungen

Das ist unter Python3 (fast) kein Problem mehr! Zeichenketten in Python3 sind Unicode-Zeichenketten.

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[7], der EBCDIC[8], 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

[7] ASCII: American Standard Code for Information Interchange
[8] Extended Binary Coded Decimals Interchange Code

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 Datei zu packen. Unicode enthält weit über 100 000 Zeichen. Diese sind in Ebenen zusammengefasst, von denen jede 65 536 Zeichen enthält. Ebene 0 enthält die ASCII-Zeichen. Insgesamt enthält Unicode mehr als 1 Million Zeichen.

Die Unicode-Zeichen werden in Blöcken aufgelistet. Man kann sich auf der Homepage des Unicode-Konsortiums die unterschiedlich langen Blöcke anschauen. Man kann sich aber auch mal den einen oder anderen Block durch ein kleines Python-Programm anzeigen lassen. Hier unten steht das Beispiel-Programm, das hier die Schriftzeichen der Cherokee anzeigt.

Wenn ich mal Zeit habe, werde ich LaTeX auch noch dazu überreden, dass es die Zeichen aus nicht-europäischen Zeichensätzen druckt; jetzt muss jeder, der das sehen will, das Programm hier unten laufen lassen. Vorher könnte man mal auf der Seite blättern, und sich einen interessanten Zeichensatz anzeigen lassen. Nicht nur der der Cherokee ist interessant!

Listing 7.18: Cherokee-Schrift

```
1 #!/usr/bin/python3
2 spaltenZaehler = 1
3 for z in range(0x13a0,0x13ff):
4     if spaltenZaehler % 8:
5         print(format(z,"5X"),chr(z), end='   ')
6     else:
7         print(format(z,"5X"),chr(z), end='\n')
8     spaltenZaehler += 1
9 print()
```

In Python3 können Unicode-Zeichen angegeben werden durch

1. \u gefolgt von einer vierstelligen Hexadezimalzahl. Damit werden Zeichen der elementaren Ebenen (0 bis 256, in Hexadezimalschreibweise 00 bis FF) angegeben. Die ersten beiden Stellen der Zahl geben die Ebene an, wobei 00 die ASCII-Ebene ist, die letzten beiden Stellen geben die Position des Zeichens in der Ebene an.
2. \U gefolgt von einer achtstelligen Hexadezimalzahl. Damit werden Zeichen der höheren Ebenen (größer als 256) angegeben.
3. \N{Bezeichnung}, wenn die Bezeichnung des Zeichens bekannt ist.

```
>>> z = '\u0041'
>>> print(z)
A
>>> z = '\u00E9'
```

```
>>> print(z)
é
>>> z = '\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> print(z)
?é?
```

7.2.1. Unicode in Python 3

In einem kleinen Programmschnipsel soll gezeigt werden, dass Python3 Unicode beherrscht und wie man die Byte-Darstellung eines Unicode-Strings erhält. Kommentare im Programmcode erklären, was gemacht wird. (Das muss ich wegen der Sonderzeichen leider anders setzen als die anderen Programmschnipsel.)

```
#!/usr/bin/python3
####
## Texte sind Zeichenketten, Zeichenketten sind Ketten von Zeichen,
## Zeichen ist ein Unicode-Zeichen
##
####
text = 'Ein Café in Århus und ein Wein in A Coruña'

print(text)

woerter = text.split(' ')
for w in woerter:
    print(w, '\tLänge des Wortes:',  len(w))
print('Man sieht: auch Sonderzeichen aus dem Französischen, \
Dänischen, Spanischen werden als Zeichen erkannt.\n')

####
## Unicode-Zeichen sind durchnummeriert.
## Die "Hausnummer" "ist der Code Point" des Zeichens.
## Die Hausnummer beginnt immer mit "U+";
## darauf folgen mindestens 4 hexadezimale Ziffern.
##
## Um als Text dargestellt zu werden, muss die Hausnummer
## in ein Byte umgewandelt werden.
## Die umgewandelten Zeichen stehen in dem "Encoding"
##
## Code Point -> Byte: encoding
## Byte -> Code Point: decoding
####

byteCodeString = []
```

```
for w in woerter:
    print(w, '\tLänge des Wortes:',  len(w), '\tbyteCode', w.encode('utf8'))
    byteCodeString.append(w.encode('utf8'))

print(text)
print(byteCodeString)

for bcw in byteCodeString:
    print(bcw.decode(), end=' ')
```

Die Ausgabe des Programms sieht so aus:

```
Ein Café in Århus und ein Wein in A Coruña
Ein  Länge des Wortes: 3
Café  Länge des Wortes: 4
in  Länge des Wortes: 2
Århus  Länge des Wortes: 5
und  Länge des Wortes: 3
ein  Länge des Wortes: 3
Wein  Länge des Wortes: 4
in  Länge des Wortes: 2
A  Länge des Wortes: 1
Coruña  Länge des Wortes: 6
Man sieht: auch Sonderzeichen aus dem Französischen, Dänischen,
Spanischen werden als ein Zeichen erkannt.

Ein  Länge des Wortes: 3  byteCode b'Ein'
Café  Länge des Wortes: 4  byteCode b'Caf\xc3\xa9'
in  Länge des Wortes: 2  byteCode b'in'
Århus  Länge des Wortes: 5  byteCode b'\xc3\x85rhus'
und  Länge des Wortes: 3  byteCode b'und'
ein  Länge des Wortes: 3  byteCode b'ein'
Wein  Länge des Wortes: 4  byteCode b'Wein'
in  Länge des Wortes: 2  byteCode b'in'
A  Länge des Wortes: 1  byteCode b'A'
Coruña  Länge des Wortes: 6  byteCode b'Coru\xc3\xb1a'
Ein Café in Århus und ein Wein in A Coruña
[b'Ein', b'Caf\xc3\xa9', b'in', b'\xc3\x85rhus', b'und', b'ein', b'Wein',
b'in', b'A', b'Coru\xc3\xb1a']
Ein Café in Århus und ein Wein in A Coruña
```

Listing 7.19: Unicode-Strings (Forts.)

```
1 >>> import unicodedata
2 >>> unicodedata.name('$')
```

```
3  'DOLLAR SIGN'
4  >>> unicodedata.name('€')
5  'EURO SIGN'
6  >>> unicodedata.name('\u2603')
7  'SNOWMAN'
8  >>> unicodedata.name('é')
9  'LATIN SMALL LETTER E WITH ACUTE'
10 >>> p = 'caf\u00e9'
11 >>> print(p)
12 café
13 >>> t = 'th\u00e9 \N{LATIN SMALL LETTER A WITH GRAVE} la menthe'
14 >>> print(t)
15 thé à la menthe
16 >>> t2 = 'th\N{LATIN SMALL LETTER E WITH ACUTE}
17          \N{LATIN SMALL LETTER A WITH GRAVE} la menthe'
18 >>> print(t2)
19 thé à la menthe
```

Leider wird in diesem Skript der Schneemann nicht angezeigt. Aber auch dazu gibt es in `unicodedata` eine Methode: **unicodedata.lookup('SNOWMAN')**

7.3. 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.

7.3.1. Die C-ähnliche Formatierung mit dem %-Operator

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:

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

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

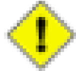
Listing 7.20: Formatierungsanweisungen

```
1  >>> du = 'Hannah'
2  >>> print('Guten Tag, %s, wie geht es Dir?' % du)
3  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 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:

Listing 7.21: Formatierung mit 2 Parametern

```
1 >>> mann = 'Romeo'
2 >>> frau = 'Julia'
3 >>> print('Berühmte Liebespaare: %s und %s' % (mann, frau))
4 Berühmte Liebespaare: Romeo und Julia
5 >>> m = 'Loriot'
6 >>> f = 'Evelyn Hamann'
7 >>> print('Berühmte Liebespaare: %s und %s' % (m, f))
8 Berühmte Liebespaare: Loriot und Evelyn Hamann
```

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

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

Tabelle 7.3.: Format-Codes

Die Format-Codes für Zahlen können noch Optionen bekommen.

7.3.2. Die Formatierung mit Hilfe der `format`-Methode

Seit Python 2.6 existiert eine weitere Möglichkeit, Ausgaben zu formatieren. Die Methode **`format`** bietet ein paar zusätzliche Möglichkeiten, die die Formatierung komfortabler machen. Außerdem ist eine Fehlerquelle, die doppelte Bedeutung des Prozentzeichens innerhalb der Formatierungsanweisung, damit weggefallen. Die allgemeine Syntax einer Formatierung mittels der `format`-Methode sieht so aus:

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.

Tabelle 7.4.: Format-Option-Codes

Listing 7.22: format in Pseudocode

```
1 fs = formatMuster
2 fs.format(Werte)
```

Ein Formatmuster besteht aus

- Zeichenketten der Länge 0 bis n
- Platzhalter für zu formatierende Informationen
- Positionierungsparameter
- Längenangaben

Platzhalter werden durch geschweifte Klammern angegeben. In den jeweiligen geschweiften Klammern können Positionsangaben und Längenangaben stehen. Ein Beispiel für eine Zeichenkette mit einem Platzhalter ohne weitere Angaben:

```
'ich heiße {}'
```

Diese Zeichenkette mit genau einem Platzhalter wird jetzt in einer Variablen gespeichert und ist damit eine Formatierungsanweisung. Danach wird für diese Formatierungsanweisung die Methode `format` mit einem Wert für den Platzhalter aufgerufen und das Ergebnis an die Funktion `print` weitergegeben.

Listing 7.23: einfache Formatierung mit format

```
>>> ihFormat = 'ich heiße {}'
>>> ausgabeText = ihFormat.format('Martin')
>>> print(ausgabeText)
ich heiße Martin
```

Im nächsten Beispiel hat der Formatstring 2 Platzhalter. Bei mehreren Platzhaltern muss der Programmierer beachten, dass die Werte für die Platzhalter in der richtigen Reihenfolge übergeben werden. Das wurde in den ersten 3 Zeilen des folgenden Beispiels richtig gemacht, in den folgenden 2 Zeilen nicht.

```
>>> ngFormat = 'ich heiße {} und habe Schuhgröße {}'
>>> ngFormat.format('Martin',43)
'ich heiße Martin und habe Schuhgröße 43'

>>> ngFormat.format(43, 'Martin')
'ich heiße 43 und habe Schuhgröße Martin'
```

Eine Möglichkeit, solche Probleme zu umgehen, ist die Benutzung von benannten Parametern.

```
>>> ngFormat = 'ich heiße {name} und habe Schuhgröße {sg}'
>>> ngFormat.format(sg=43, name='Martin')
'ich heiße Martin und habe Schuhgröße 43'
```

Noch eleganter ist die Übergabe eines Dictionary an den Formatstring. Dabei werden die Parameter nicht einzeln übergeben, sondern das Dictionary wird durch das Voranstellen der zwei ** entpackt.

```
>>> ich = {'name' : 'Martin', 'sg' : '43'}
>>> ngFormat.format(**ich)
'ich heiße Martin und habe Schuhgröße 43'
```

Wenn in einem Text mehrere Platzhalter angegeben werden, muss dem format-Befehl ein Tupel mit genauso vielen Objekten übergeben werden. Die Platzhalter werden dann durch die Objekte in deren Reihenfolge gefüllt. Dazu gibt es jetzt ein einfaches Beispiel.

Listing 7.24: format mit einfachem Text

```
>>> fA = "{:10}{:40} {:2}"  # fA ist die Format-Anweisung;
>>> print(fA.format('Martin', 'hat die Schuhgröße', 43))
                             # 'Martin' mit Länge 10,
                             # Text mit Länge 40,
                             # Schuhgröße mit Länge 2
Martin    hat die Schuhgröße                       43
>>> print(fA.format('Caro', 'hat die Schuhgröße', 23))
                             # jetzt mit anderen Werten
Caro      hat die Schuhgröße                       23
```

Jetzt soll an einem Beispiel gezeigt werden, dass Formatierung viel Arbeit und Zeit sparen kann. Hier sollen die Zahlen 1 bis 10, ihre Quadrate und ihre Kuben (3. Potenzen) angezeigt werden. Hier muss ich vorgreifen: damit man sieht, wie effektiv Formatierung benutzt werden kann, enthält das nächste Programm (in seinen Variationen) eine Zählschleife: hier wird eine Berechnung mit Ausgabe für alle ganzen Zahlen von 0 bis 9 (jeweils einschließlich) gemacht. (Wer mehr zu Schleifen wissen will, muss sich entweder etwas gedulden oder vorblättern zu Zählschleifen.) Zuerst läuft das Programm ohne Formatierung ab:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

for i in range(10):
    print(i, i**2, i**3)
```

Das sieht nicht gut aus, denn wir erwarten schon die Ausgabe in Spalten untereinander, wobei die Zahlen wie gewohnt rechtsbündig geschrieben sein sollten.

Listing 7.25: Unformatierte Ausgabe

```
>>>
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
```

Jetzt wird formatiert! Das kann auf zwei Arten realisiert werden:

1. indem man die zu formatierenden Werte direkt in die Ausgabe schreibt
2. indem man ein Format-Muster aufbaut und die Format-Methode von Zeichenketten mit dem Format-Muster aufruft.

Zuerst kommt hier also die direkte Ausgabe:

Listing 7.26: Formatierte Ausgabe direkt (Quelltext)

```
>>> for i in range(1,11):
      print('{zahl:>4}{quadr:>6}{kubus:>8}'.format(
                      zahl = i, quadr = i*i, kubus = i*i*i))

   1     1       1
   2     4       8
   3     9      27
   4    16      64
   5    25     125
   6    36     216
   7    49     343
   8    64     512
   9    81     729
  10   100    1000
```

Und jetzt kommt das selbe mittels eines Format-Musters.

Listing 7.27: Formatierte Ausgabe mittels Format-Muster(Quelltext)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

formatMuster = "{:>4} {:>6} {:>8}"

for i in range(10):
    print(formatMuster.format(i, i**2, i**3))
```

In dem Formatmuster steht für jeden Ausdruck, der ausgegeben werden soll, in einem Paar von geschweiften Klammern eine Formatierungsanweisung. Nach dem Doppelpunkt steht dann in jeder geschweiften Klammer in unserem Fall zuerst die Positionierungsanweisung >, die signalisiert, dass die Ausgabe nach rechts geschoben wird, also rechtsbündig stehen soll. Danach steht die Feldbreite für diesen Teil der Ausgabe. Die 3 geschweiften Klammern geben also an, dass z.B. die erste Zahl auf 4 Stellen Breite rechtsbündig geschrieben werden soll. Und die Ausgabe sieht doch gut aus!

Listing 7.28: Formatierte Ausgabe

```
>>>
   0      0        0
   1      1        1
   2      4        8
   3      9       27
   4     16       64
   5     25      125
   6     36      216
   7     49      343
   8     64      512
   9     81      729
```

Auch vor dem Doppelpunkt kann noch etwas stehen. Im einfachsten Fall kann dort eine Positionsangabe der Werte, die ausgegeben werden sollen, gemacht werden. (Nicht vergessen: man fängt bei 0 an zu zählen!!)

Listing 7.29: Formatierte Ausgabe in umgekehrter Reihenfolge (Quelltext)

```
#!/usr/bin/python

formatMuster = "{2:>4} {1:>6} {0:>8}"

print()
for i in range(10):
  print(formatMuster.format(i, i**2, i**3))
```

Listing 7.30: Formatierte umgekehrte Ausgabe

```
>>>
       0      0     0
       1      1     1
       8      4     2
      27      9     3
      64     16     4
     125     25     5
     216     36     6
     343     49     7
     512     64     8
     729     81     9
```

Eine der oben erwähnten Verbesserungen der `format`-Methode ist, dass man Argumente an die Formatierungsanweisung per Name übergeben kann. Das macht ein Programm besser lesbar (auch wenn es für unser einfaches Beispiel noch nicht nötig ist), und deswegen steht der Quelltext hier ohne weitere Erklärungen:

Listing 7.31: Formatierte Ausgabe mit benannten Parametern

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

formatMuster = "{zahl:>4} {quadrat:>6} {kubus:>8}"

print()
for i in range(10):
  print(formatMuster.format(zahl=i, quadrat=i**2, kubus=i**3))
```

In der Spaltenformatierung kann der Bereich mit einem Füllzeichen ausgefüllt werden:

Listing 7.32: Formatierte Ausgabe mit Füllzeichen(Quelltext)

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-

formatKette = "Zahl: {zahl:.>4} Quadrat: {quadrat:_>6} Kubus: {kubus:*>8}"

print()
for i in range(11):
  print(formatKette.format(zahl=i, quadrat=i**2, kubus=i**3))
```

Und so sieht es aus:

Listing 7.33: Formatierte Ausgabe mit Füllzeichen

```
Zahl: ...0 Quadrat: _____0 Kubus: *******0
Zahl: ...1 Quadrat: _____1 Kubus: *******1
Zahl: ...2 Quadrat: _____4 Kubus: *******8
Zahl: ...3 Quadrat: _____9 Kubus: ******27
Zahl: ...4 Quadrat: ____16 Kubus: ******64
Zahl: ...5 Quadrat: ____25 Kubus: *****125
Zahl: ...6 Quadrat: ____36 Kubus: *****216
Zahl: ...7 Quadrat: ____49 Kubus: *****343
Zahl: ...8 Quadrat: ____64 Kubus: *****512
Zahl: ...9 Quadrat: ____81 Kubus: *****729
Zahl: ..10 Quadrat: ___100 Kubus: ****1000
```

Außerdem kann jeder Spaltenformatierung noch ein Vortext vorangestellt werden:

Listing 7.34: Formatierte Ausgabe mit Vortext (Quelltext)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

formatKette = "Zahl: {zahl:>4} Quadrat: {quadrat:>6} Kubus: {kubus:>8}"

print()
for i in range(11):
  print(formatKette.format(zahl=i, quadrat=i**2, kubus=i**3))
```

Und so sieht es aus:

Listing 7.35: Formatierte Ausgabe mit Vortext

```
Zahl:    0 Quadrat:      0 Kubus:        0
Zahl:    1 Quadrat:      1 Kubus:        1
Zahl:    2 Quadrat:      4 Kubus:        8
Zahl:    3 Quadrat:      9 Kubus:       27
Zahl:    4 Quadrat:     16 Kubus:       64
Zahl:    5 Quadrat:     25 Kubus:      125
Zahl:    6 Quadrat:     36 Kubus:      216
Zahl:    7 Quadrat:     49 Kubus:      343
Zahl:    8 Quadrat:     64 Kubus:      512
Zahl:    9 Quadrat:     81 Kubus:      729
Zahl:   10 Quadrat:    100 Kubus:     1000
```

Für Zahlen können verschiedene Darstellungen gewählt werden. Hier folgen ein paar Beispiele, die aber anhand der untenstehenden Tabelle leicht verstanden werden!

Listing 7.36: Formatierte Ausgabe von Zahlen

```
>>> fs = "{:d} {:b} {:x}"
>>> print(fs.format(2,2,2))
2 10 2
>>> print(fs.format(18,18,18))
18 10010 12
>>> print(fs.format(31,31,31))
31 11111 1f
```

Zahlen werden mit dem Parameter `d` im Dezimalsystem, mit dem Parameter `b` im Dualsystem, mit dem Parameter `x` im Hexadezimalsystem ausgegeben, wobei dann die „abcdef" in Kleinbuchstaben geschrieben werden. Und hier kommt dann die ultimative Quizfrage: was bewirkt dann wohl der Parameter `X`?

In einer Tabelle folgen hier die wichtigsten Parameter für die `format`-Methode.

Zeichen	Bedeutung / Erläuterung
Ausrichtung	
>	linksbündig
<	rechtsbündig
ˆ	zentriert
=	rechtsbündig für Zahlen
Vorzeichen	
+	bei allen Zahlen ein Vorzeichen
-	nur bei negativen Zahlen ein Vorzeichen
#	nur bei Zahlen: Zahlsystem(0b für Dualsys., 0x für Hexadezimalsys.)
Typ des Feldes	
b	schreibt eine Zahl im Dualsystem
c	beliebige Zeichen (character)
d	Ganzzahl (dezimal)
e	Zahl in Exponential-Schreibweise
f	Festkommazahl

Tabelle 7.5.: Parameter für die `format`-Methode

Um Zahlen im in Deutschland üblichen Format, d.h. mit Dezimalkomma und Punkten bei der Trennung von großen Zahlen nach jeweils 3 Stellen, gezählt vom Dezimalkomma aus, auszugeben, gibt es die Möglichkeit, Python die `loc ale` „beizubringen“.

Listing 7.37: Dezimalkomma

```
1 >>> import locale
2 >>> locale.setlocale(locale.LC_ALL, "de_DE.UTF-8")
3 'de_DE.UTF-8'
4 >>> "{0:20.16n}".format(12312323456.7878787878)
5 '12.312.323.456,78788'
```

Das Modul `locale` muss hier zuerst importiert, dann die lokale Information auf deutsch „de_DE.UTF-8“ umgestellt werden Danach kann man Zahlen mit Hilfe des Formatparameters `n` in der jetzt gültigen lokalen Schreibweise ausgeben.

Im folgenden Kapitel über Listen und Dictionaries werden noch andere (zum Teil elegantere) Möglichkeiten gezeigt, wie man Ausgaben formatieren kann.

7.4. 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 Programmiersprache 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 könntest 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 [12] empfohlen.

7.4.1. 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.

Listing 7.38: Reguläre Ausdrücke (Meier zum ersten)

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

Um die „Meiers“ noch weiter zu bemühen: es gibt in manchen Gegenden auch noch den Familiennamen „Mair“. Als Regel formuliert heißt das, dass das „e“ fakultativ ist. Dazu benötigen wir einen Quantor[9], der genau 0 oder 1 bestimmtes Zeichen bzw. 0 oder 1 Zeichen aus einer Auswahl erlaubt. Das erledigt das Fragezeichen, das dem bestimmten Zeichen folgt.

Hier wird aber eine andere Methode zum Durchsuchen des Textes benutzt. Während vorher durch die Methode **findall** eine Liste generiert wurde (siehe die Ausgabe oben), erzeugt die Methode **finditer** einen Iterator.

Listing 7.39: Reguläre Ausdrücke (Meier zum zweiten)

```
1  #!/usr/bin/python
2  import re
3
4  text = '''In diesem Text sind jede Menge "Maiers":
5  Herr Maier, Frau Meier, Herr Majer und Frau Mair'''
6
7  gefunden = re.finditer(r'M[ae][ijy]e?r', text)
8  for i in gefunden:
9      print(i.group(0))
10
11 >>> Maier
12 Maier
13 Meier
14 Majer
15 Mair
```

Notwendig ist der Import des Moduls re.

[9] siehe dazu die untenstehende Tabelle

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
?	kein oder ein Auftreten des vorhergehenden Zeichens
ˆ	Anfang der Zeile
$	Ende der Zeile
\d	[0-9]
\w	[0-9a-zA-z_] (also ein Buchstabe, eine Ziffer oder ein Unterstrich)
\s	Leerzeichen, Tabulator, Zeilenvorschub

Tabelle 7.6.: Reguläre Ausdrücke: Zeichen

Beachte dabei: die Zeichen \d, \w und \s gibt es auch mit Großbuchstaben, also \D, \W und \S. Sie bedeuten jeweils „alles, was nicht zum Zeichenbereich des jeweiligen Kleinbuchstaben gehört“.

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

Tabelle 7.7.: Reguläre Ausdrücke: Methoden

7.4.2. Wortteile aus einem Text herausfiltern

Das Problem, das wir als erstes behandeln wollen, ist folgendes: aus einem beliebigen Text sollen alle Wörter herausgefiltert werden, die mit einem „h“ oder einem „H“ anfangen. Die erste Lösung benutzt fast die selben Python-Sprachmittel wie das vorige Beispiel:

Listing 7.40: Wörter, die mit „h“ oder „H“anfangen

```
1 >>> text = 'Hier kommt ein lautes haha, weil Heiner und Harry
2                heute lustig sind.'
3 >>> muster2 = re.compile(r"(H\w*|h\w*)")
4 >>> muster2.findall(text)
5 ['Hier', 'haha', 'Heiner', 'Harry', 'heute']
```

In der runden Klammer des regulären Ausdrucks stehen zwei Teile, die durch den senkrechten Strich voneinander getrennt sind. Das bedeutet, dass alle Zeichenketten gefunden

werden sollen, die entweder auf den regulären Teilausdruck vor dem senkrechten Strich passen oder auf den danach. Das \w ist eine Kurzschreibweise für [A-Za-z0-9_], findet also einen beliebigen Buchstaben, eine Ziffer oder einen Unterstrich. Die Methode `findall` macht genau das, was ihr Name sagt.

Als nächstes sollen aus einem Text bestimmte Teile herausgefiltert werden. Suchen wir also in einem Text nach unanständigen Sachen.

Listing 7.41: Sex, Sex, Sex

```
1  #!/usr/bin/python
2  import re
3  text = 'An der Schule gibt es ein Jazz-Sextett\n
4      Ein Sextaner darf da nicht mitspielen\n
5      In Jazz-Stücken werden oft Sextakkorde verwendet'
6  saetze = text.split('\n')
7  unanstaendigeWoerter = list()
8
9  for satz in saetze:
10     wort = re.search(r"[-\s](Sex[a-z]*)",satz)
11     unanstaendigeWoerter.append(wort.group(1))
12 print(unanstaendigeWoerter)
13
14 >>> ['Sextett', 'Sextaner', 'Sextakkorde']
```

Der Text besteht aus 3 Zeilen, jede abgeschlossen durch einen Zeilenvorschub. Durch `split` wird der Text in 3 Sätze aufgeteilt, die in einer Liste gespeichert werden. `unanstaendigeWoerter` ist eine zu Beginn leere Liste.

Mittels `re.search()` wird nach einem regulären Ausdruck gesucht. Der reguläre Ausdruck setzt sich so zusammen:

1. In eckigen Klammern steht eine Gruppe von Zeichen, aus denen genau eines ausgewählt werden muss, in diesem Fall besteht die Liste aus dem Bindestrich und einem Leerzeichen. Die Variable `\s` bezeichnet die Gruppe der „white spaces“, also aller möglichen Leerzeichen.
2. Es folgt in runden Klammern die Zeichenkette „Sex“.
3. Daran schließt sich wieder eine eckige Klammer an, in der die Menge aller Kleinbuchstaben steht. Das bedeutet, dass von diesen Kleinbuchstaben wieder einer ausgewählt wird. Der Stern nach der schließenden eckigen Klammer ist der Quantor, der besagt, dass null oder beliebig viele Kleinbuchstaben folgen dürfen.
4. Nach dem Stern steht die schließende runde Klammer. Dadurch wird das, was durch den in runden Klammern stehenden Teil des regulären Ausdrucks abgedeckt wird, als (Teil-)Gruppe eines Fundes betrachtet.
5. Denn es soll ordentlich ausgegeben werden: das einleitende Leerzeichen oder der einleitende Bindestrich soll im Ergebnis nicht dargestellt werden, sondern nur das, was durch den in runden Klammern abgedeckten Teil gefunden wird. Darum wird an die Liste der unanständigen Wörter nur der Teil der 1. Gruppe angehängt.

OK. Das Programm hat gefunden, was es sollte. Syntaktisch ist es korrekt, aber naja, das ist nicht das, was wir finden wollten.

7.4.3. gpx-Datei eines Fitness-Trackers

Im nächsten Beispiel geht es darum, aus einer Datei, die von einem Fitness-Tracker erstellt wurde und die eine schöne Rundfahrt mit dem Fahrrad enthält, die Zeilen, die die Herzfrequenz enthalten, zu löschen, denn die Dateien sollen auf einem Blog veröffentlicht werden.[10] Und so persönliche Daten gehören nicht auf eine Internet-Seite. Das ist kein größeres Problem, denn es handelt sich um eine `.gpx`-Datei.[11] Das ganze wird hier einfach mittels eines regulären Ausdrucks gelöst, obwohl das auch über einen XML-Parser ginge. (Und wenn ich mal Lust habe, schreibe ich auch noch ein Kapitel über `xml`-Dateien.)

[10] Danke an Lisa, die genau das macht, und die natürlich auch findet, dass Herzensangelegenheiten Privatsache sind.

[11] die ja eigentlich nur eine `xml`-Datei ist

Listing 7.42: Fitness ohne Herz

```
import re
...
pulsZeileRE = re.compile('''^(\s)*<gpxtpx:hr>\d{1,3}</gpxtpx:hr>(\s)*$''')
aDat = ...
with open(dateiname, 'r') as eDat:
    for eineZeile in eDat:
        if pulsZeileRE.findall(eineZeile):
            pass
        else:
            aDat.write(eineZeile)
aDat.close()
```

Die ersten beiden Zeilen sollten jetzt verständlich sein; der Rest arbeitet mit einer Eingabe- und einer Ausgabedatei. Das Original bleibt, wie es ist, die Ausgabedatei enthält nur Zeilen, in denen der reguläre Ausdruck nicht gefunden wurde. Nach dem weiter hinten folgenden Kapitel über Dateien kannst Du das Drumherum, das, was oben durch die 3 Pünktchen gekennzeichnet ist, leicht selbst schreiben.

Für eine einzelne Datei lohnt sich das Programm ja fast nicht, aber das Programm kann natürlich weiter aufgebohrt werden, so dass es auf einen Schlag alle `gpx`-Dateien eines Verzeichnisses bearbeitet. Und dann lohnt es sich.

7.4.4. Beispiel für Zahlen

Im nächsten 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.

Für nicht triviale Suchmuster ist es gute Sitte, das Suchmuster als ein Objekt der Klasse „regulärer Ausdruck“ zu speichern.[12]

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

[12] Objekte werden später eingeführt im Kapitel Klassen

reicht noch nicht für eine gute Prüfung, denn es 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.

7.4.5. 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

Listing 7.43: Drei Chinesen ... (reguläre Ausdrücke)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# importiere das Modul "Reguläre Ausdrücke"
import re

vokalNeu = input('Welcher Vokal soll es denn sein? ')

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" '''

# regulären Ausdruck aufbauen, das ist ein "raw string"
suchString = re.compile(r"""(ei|au|ä|a|e|i|o|u)""")
neutext = suchString.sub(vokalNeu,text)
print(neutext)
```

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:

```
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!)

Für Texte gibt es noch die Methode `translate`. Mit deren Hilfe kann man das Problem fast lösen. Allerdings benötigt man hierzu Dictionaries, und dieses Thema taucht in diesem Skript erst später auf.

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!

7.4.6. Eine etwas sinnvollere Anwendung: Wörter mit Doppelbuchstaben finden

Du sollst also in einem Text alle Wörter finden, die einen Doppelbuchstaben enthalten; Beispiele dafür sind etwa „See“, „Meer“ und „Leere“, die alle zwei aufeinanderfolgende „e“ enthalten, oder „Wall“, „Qualle“ und „Welle“, die alle zwei aufeinanderfolgende „l“ enthalten. Wie sieht die Regel aus?[13]

Die Regel sagt uns zuerst einmal, dass wir Wörter finden müssen. Wörter sind Zeichenketten, die von Leerzeichen begrenzt werden. Auf sie passt also der reguläre Ausdruck `\w*`. Innerhalb dieses Wortes soll also jetzt ein Buchstabe gefunden werden, der von dem selben Buchstaben gefolgt wird. Das geschieht durch eine Gruppierung, auf die dann Bezug genommen wird. Eine Gruppierung wird durch Einklammern erzeugt: `(\w)` ist eine Gruppe, die genau aus einem Element der Klasse `\w` besteht, also aus einem Buchstaben, einer Ziffer oder einem Unterstrich.[14] Diese Gruppe soll sich also jetzt wiederholen. Da es sich hier um die erste (und bis jetzt einzige) Gruppe handelt, beziehe ich mich auf diese Gruppe durch den regulären Ausdruck „\1“. Damit habe ich meine Regel beschrieben: gesucht sind Wörter, die mit beliebig vielen Buchstaben beginnen, auf die ein Buchstabe folgt, der sofort noch einmal folgt, gefolgt von beliebig vielen Buchstaben: „\w* (\w)\1\w*“.

[13] Ich gehe davon aus, dass der Text vorläufig in einer Variablen steht; die Bearbeitung von Text, der in einer Textdatei enthalten ist, kann später jeder für sich programmieren, wenn der Zugriff auf Dateien bekannt ist.

[14] Seien wir also mal kurzfristig großzügig (oder vielleicht schlampig?), und erlauben auch Ziffern in Wörtern, wodurch auch in Ludwig 11. das „Wort“ „11“ gefunden würde

Der reguläre Ausdruck wird jetzt kompiliert, wodurch ein `re-Objekt` entsteht. Mit der Methode `finditer` wird jetzt iterativ der Text durchsucht. Die eventuellen Ergebnisse stehen jetzt in dem Attribut `group` der einzelnen Objekte. Das Programm sieht so aus:

Listing 7.44: Doppelbuchstaben finden

```
#!/usr/bin/python
import re

text = '''In diesem Text sind mindestens zwee Wörter mit Doppelbuchstaben.
Findet er auch Meereswelle?'''
wmdb = re.compile(r"""\w*(\w)\1\w*""")
gefunden = wmdb.finditer(text)
for i in gefunden:
    print(i.group(0))

>>> zwee
Doppelbuchstaben
Meereswelle
```

7.5. Aufgaben zu Texten (Strings)

1. Lass Dir mit

   ```
   help(str)
   ```

   den Hilfetext zu Strings anzeigen; probiere einige der dort angegebenen Methoden von Strings aus. Notiere Dir Aktionen, die Dir interessant für Deine zukünftige Arbeit erscheinen, und speichere die dazugehörigen Programmschnipsel in Python-Dateien.

2. a) Benutze die Methode `translate`, um ein Programm zur Caesar-Verschlüsselung zu schreiben.

   b) Entsprechend natürlich ein Programm zur Entschlüsselung eines mittels der Caesar-Verschlüsselung verschlüsselten Textes.

3. a) Zerlege einen beliebigen Text in eine Liste von Wörtern.

   b) Sortiere diese Wörter alphabetisch.

   c) Nach dem Sortieren erhältst Du (mit dem obigen Satz) die Liste:

   ```
   ['Liste', 'Text', 'Wörtern', 'Zerlege', 'beliebigen', 'eine',
   'einen', 'in', 'von']
   ```

   Diese Sortierung gefällt Dir nicht? Mir auch nicht. Beschreibe Dein Missfallen, überlege Dir, wie Du diese Wörter gerne sortiert hättest und schreibe ein Programm, das es besser macht.

4. Es stehen am Fahrrad-Rikscha-Startplatz:

   a) 30 Fahrrad-Rikschas mit je 2 Sitzplätzen für Passagiere

   b) 18 gut durchtrainierte Fahrer

   c) 57 potentielle Fahrgäste

Ausgegeben werden soll etwa:

- Wir haben ... Rikschas, können also theoretisch ... Personen transportieren.
- Da im Moment nur ... Fahrer zur Verfügung stehen, können ... Fahrgäste mitfahren und ... Fahrgäste müssen in ca. 1 Stunde wiederkommen, wenn sie dann noch fahren wollen.

5. Ein Programm soll aus dem Buchstaben x ein Herz malen, etwa so:

Listing 7.45: Herz aus x

```
  xx  xx
 x   x   x
  x     x
   x  x
    x
```

Es ist sinnvoll, sich vorher ein Muster auf ein kariertes Papier zu malen!

Jetzt lass jeweils ein neues Programm die anderen Symbole der Spielkartenfarben Kreuz, Pik und Karo malen. (Es muss nicht jedesmal mit einem „x“ sein.)

8. Strukturierte Daten

8.1. Überblick

An strukturierten Daten unterscheidet man:

1. Listen
2. Dictionaries
3. Tupel
4. Mengen
5. Iteratoren

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.

8.2. Listen

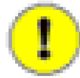
> Reg: [reading prepared statement] "We, the People's Front of Judea, brackets, official, end brackets, do hereby convey our sincere fraternal and sisterly greetings to you, Brian"
>
> *(Monty Python[1])*

8.2.1. 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. Die Zahl, die die „Hausnummern“ zählt, heißt Index

[1] in: Life of Brian

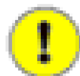
Wichtig!



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.

Wichtig!



Beachte dabei: Ein negativer Wert für den Index bedeutet, dass man von hinten anfängt zu zählen.

8.2.2. Erzeugung von Listen

8.2.2.1. Durch Angabe der Elemente

Die einfachste Art, eine Liste zu erzeugen, ist durch die Angabe der Elemente der Liste. Dabei weisen wir gleich einer Variablen diese Liste zu:

Listing 8.1: Eine Liste erzeugen

```
1 zahlenListe1 = [1,2,3,4,5,9,7]
```

8.2.2.2. Als Objekt der Klasse `list`

Das kann man auch erledigen, indem man die Liste als ein Objekt der Klasse „list“ erstellt und somit den Konstruktor der Klasse aufruft.[2]

Listing 8.2: Eine Liste als Objekt erzeugen

```
1 zahlenListe2 = list([1,2,3,4,5,9,7])
```

8.2.2.3. Mittels `range`

Auf Seite 216 (Einführung von Schleifen) lernst Du die Funktion `range` kennen. Diese Funktion liefert einen Bereich von Zahlen. Dadurch ist sie auch gut geeignet, um Zahlenlisten herzustellen:

[2] Der Satz ist im Moment noch schwer zu verstehen; wenn Du weiter hinten in die Objektorientierte Programmierung eingestiegen bist, wird Dir der Sinn dieses Satzes hoffentlich klar.

Listing 8.3: Eine Liste mit `range` erzeugen

```
1 zahlenListe3 = list(range(20))
```

Listing 8.4: Eine Liste von (überleg mal selber!!) Zahlen

```
1 zahlenListe3 = list(range(20,50,3))
2 zahlenListe3
3 >>> [20, 23, 26, 29, 32, 35, 38, 41, 44, 47]
```

Anmerkung

Hier ist ein Unterschied zwischen Python 2.x und Python 3.x! In Python 2.x hat der Befehl range(20,50,3) bereits die oben angezeigte Liste erzeugt. In Python 3.x wird nur ein Generator für diese Liste erzeugt, die einzelnen Listenelemente werden bei Bedarf erzeugt. Wenn man also in Python 3.x diese Liste erhalten will, funktioniert das wie oben beschrieben durch list(range(20,50,3)).

Anmerkung

Wenn man jetzt auf die Idee kommt, dass man für eine Tabelle der x-Werte einer Funktion in der Mathematik (gewünscht ist dies für $-5 <= x <= 5$ mit der Schrittweite 0,1) diese Tabelle durch x_Werte = range(-5,6,0.1) erstellen kann, wird eine Fehlermeldung bekommen: `range` erlaubt nur ganze Zahlen als Parameter. Aber das gewünschte Ergebnis erhält man, wenn man das nächste Kapitel durchgelesen hat; dann wird man die Lösung finden.

8.2.2.4. Mit Hilfe der `list comprehension`

(siehe auch weiter unten bei List Comprehensions) Bei einer list comprehension gibt man in den eckigen Klammern eine Vorschrift für die Erzeugung der Liste an. Dabei wird eine Schleife verwendet, also etwas, was erst weiter hinten im Kapitel 9.5 angesprochen wird.

Listing 8.5: Liste über `list comprehension`

```
1 >>> liste = [i*2 for i in range(3)]
2 >>> liste
3 [0, 2, 4]
4 >>>
```

Hier kommt ein Beispiel, wie man bestimmten Zahlen bestimmte Werte zuweisen kann. Gegeben sei also eine Liste von Zahlen, es interessiert aber nur, ob die Zahlen gerade oder ungerade sind.

Listing 8.6: Gerade oder ungerade?

```
>>> m = list(range(1,11))
>>> m
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> gu = ['gerade','ungerade']
>>> geradeUngerade = [gu[i%2] for i in m]
>>> geradeUngerade
['ungerade', 'gerade', 'ungerade', 'gerade', 'ungerade', 'gerade',
    'ungerade', 'gerade', 'ungerade', 'gerade']
```

Das kann man noch etwas verschönern:

Listing 8.7: Gerade oder ungerade? Schöner!

```
>>> zahlen = range(1,101)
>>> gu = ['gerade', 'ungerade']
>>> geradeUngerade = [(i,gu[i%2]) for i in zahlen]
>>> print(geradeUngerade)
[(1, 'ungerade'), (2, 'gerade'), (3, 'ungerade'), (4, 'gerade'),
(5, 'ungerade') ... usw.
```

Oder noch schöner:

Listing 8.8: Gerade oder ungerade? Viel schöner!

```
>>> zahlen = range(1,11)
>>> gu = ['gerade', 'ungerade']
>>> geradeUngerade = [(str(i)+ ' ist ' + str(gu[i%2])) for i in zahlen]
>>> print(geradeUngerade)
['1 ist ungerade', '2 ist gerade', '3 ist ungerade', '4 ist gerade', ...]
```

Anmerkung

Klar jetzt, wie man die Wertetabelle aus dem vorigen Kapitel erzeugt?
So: `defBereich =[x/10 for x in range(-50,61)]`

8.2.2.5. Mittels `input`

Das solltest Du mal ausprobieren. Gibt man einen Text ein, so wird der Text als eine Liste von Buchstaben (und anderen Zeichen) verstanden.

Listing 8.9: Liste über `input`(1. Version)

```
1 >>> l = list(input('Liste: '))
2 Liste: abcdef
3 >>> print(l)
4 ['a', 'b', 'c', 'd', 'e', 'f']
```

Aber gib mal etwas anderes ein!

Wenn Du herumprobiert hast, hast Du natürlich wenig glückliche Ergebnisse bekommen. Zum Beispiel so etwas:

Listing 8.10: Zahlen durch Leerzeichen getrennt ergeben keine Liste von Zahlen

```
1 >>> l = list(input('Liste: '))
2 Liste: 1 2 4 8 16 32
3 >>> print(l)
4 ['1', ' ', '2', ' ', '4', ' ', '8', ' ', '1', '6', ' ', '3', '2']
```

Listing 8.11: Liste über `input`(2. Version)

```
1 >>> l = list(input('Liste: ').split())
2 Liste: 1 2 4 8 16 32
3 >>> print(l)
4 ['1', '2', '4', '8', '16', '32']
```

Immerhin! Die Zahlen-Eingabe wird bei den Leerzeichen gesplittet, aber . . . es sind immer noch keine Zahlen, sondern Zeichenketten.

Listing 8.12: Liste über `input`(3. Version)

```
1 zListe = [int(z) for z in list(input('Liste: ').split())]
2 Liste: 1 2 4 8 16 32
3 >>> print(zListe)
4 [1, 2, 4, 8, 16, 32]
```

Ja, warum eigentlich nicht? In Mathematik haben wir gelernt, dass man Funktionen verketten kann, dann kann man das auch mal in Python versuchen. Also verketten wir einfach den bei Leerzeichen gesplitteten Zahlen-Text und auf das Ergebnis dieser Operation lassen wir die Funktion „Umwandlung in eine Ganzzahl“ , die Funktion `int`, los, und mit diesem Ergebnis gehen wir in eine `list comprehension`.

Hier nochmal schrittweise:

Schritt Nr.	Befehl	Ergebnis
1	`input('Liste :')`	'1 2 4 8 16 32'
2	`.split()`	'1' '2' '4' '8' '16' '32'
3	`...for ...`	list comprehension! mache aus den einzelnen Strings eine Liste von Strings
4	`int(z)`	aber während der list comprehension wandle jeden einzelnen String in eine Ganzzahl um

8.2.3. Anzeigen von Listenelementen

Das Anzeigen von Elementen einer Liste erfolgt fast immer über eine Schleife. Schleifen werden aber erst weiter hinten in diesem Skript bei angesprochen. Also gibt es 2 Möglichkeiten:

1. Große Teile dieses Unterkapitels überblättern und später hierher zurückkehren oder

2. sofort zu den Schleifen springen, dort die ersten Beispiele lesen, so dass man die Grundlagen von Schleifen verstanden hat, und hierher zurückkehren.

8.2.3.1. Beispiele von Listen

Hier folgen zwei Beispiele von Listen. Zuerst einmal eine typische Liste, nämlich eine Einkaufsliste: In diesem Beispiel werden die Elemente der Liste mit ihren Indizes aufgeschrieben, in der Zeile über den Listenelementen mit den Indizes („Hausnummern“), wenn man am Anfang der Liste anfängt, in der Zeile unter den Listenelementen, das selbe, wenn man am Ende der Liste anfängt.

Listing 8.13: Eine Einkaufsliste

```
lm = ['Senf', 'Milch', 'Brot', 'Salat', 'Salz']
```

Index vom Anfang her	0	1	2	3	4
Element	Senf	Milch	Brot	Salat	Salz
Index vom Ende her	-5	-4	-3	-2	-1

Tabelle 8.1.: Liste mit Hausnummern und Einträgen

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

Listing 8.14: Listenelemente

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

Und dann eine Liste mit ganz verschiedenartigen Elementen:

Listing 8.15: 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:

Listing 8.16: Listenelemente

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

Jetzt zeige ich eine ziemlich wilde Liste. Denn Elemente einer Liste können auch wieder eine Liste sein, und deren Elemente ... kurz: Listen können ziemlich tief geschachtelt sein.[3]

```
>>> dinger = ['Kamel', 'Elefant',
          ['Affe',['l. Hand','r. Hand'],'Nabel'],'Nilpferd']
```

Was ist denn `dinger[2][1][1][0]` ? Erst hier im Kopf lösen, dann in Idle eingeben!!

[3] Ich weiß nicht, wie tief!

8.2.4. Operationen auf Listen

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

Listing 8.17: 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:

Listing 8.18: Rechnen mit Listen

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

Man kann die Länge einer Liste feststellen:

Listing 8.19: 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**.

Listing 8.20: Position eines Listenelements

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

8.2.4.1. Wie funktioniert das „Enthaltensein“?

Wie also ist so eine Funktion programmiert? Das ist ein Standard-Problem der Programmierung, und die Lösung (eine der Lösungen) heißt „binäre Suche“. Binäre Suche funktioniert bei geordneten Dingen, also insbesondere bei geordneten Listen. Das kann eine Liste von Zahlen sein, die der Größe nach geordnet sind oder eine Liste von Begriffen, die ebenfalls der Größe nach geordnet sind.

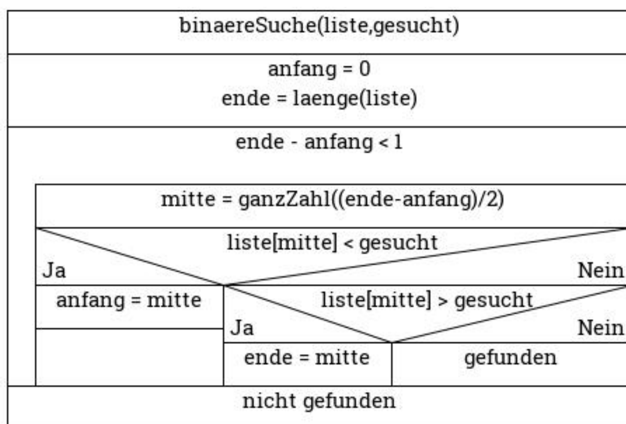


Abbildung 8.1.: Struktogramm binäre Suche

Wenn weiter hinten auf S. 220 bei der Einführung von Schleifen ein weiteres Element von Python zur Verfügung steht, kann man den Zusammenhang zwischen der Länge einer Liste, ihren Elementen und den dazugehörigen „Hausnummern“ besser verstehen.

8.2.4.2. Erzeugung einer Liste durch eine Filterfunktion

In Python gibt es eine eingebaute Funktion `filter`, die aus einer Liste mithilfe einer anderen Funktion, die jedes Element einer Liste bearbeitet, eine gefilterte Liste erzeugt. Als Beispiel sollen eine Liste der Quadratzahlen kleiner als 650 erzeugt werden. Da aber sowohl Schleifen als auch Funktionen erst weiter hinten in diesem Skript besprochen werden, wird hier nur auf die Stelle über Listen auf S. 249 verwiesen.

8.2.5. Veränderung von Listen

8.2.5.1. Ein Element anhängen

Etwas an eine bestehende Liste anzuhängen ist auch nicht schwer (aber es sieht prinzipiell anders aus als das Feststellen der Länge. Siehe dazu weiter vorne im Kapitel Objektorientierung)

Listing 8.21: Listen verlängern

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

Eine andere Möglichkeit, etwas an eine Liste anzuhängen, ist, zwei Listen zu einer zu vereinigen. Das sieht so aus:

Listing 8.22: Verlängerung einer Liste durch Zusammenkopieren

```
1 >>> bierListe = []
```

```
2 >>> bier = 'Weizen'
3 >>> bierListe += [bier]
4 >>> bier = 'Pils'
5 >>> bierListe += [bier]
6 >>> print(bierListe)
7 ['Weizen', 'Pils']
```

Man kann Listen auch verlängern, indem man einen Shortcut-Operator benutzt. Aber Vorsicht: das funktioniert nicht so, wie man es sich vielleicht naiv vorstellt.

Listing 8.23: Verlängerung einer Liste durch Shortcut (falsch)

```
1 >>> bierListe = ['Weizen','Pils']
2 >>> bier = 'Kölsch'
3 >>> bierListe += 'Kölsch'
4 >>> bierListe
5 ['Weizen', 'Pils', 'K', 'ö', 'l', 's', 'c', 'h']
```

Python fasst den Text `Kölsch` als eine Liste von Buchstaben auf, die einzeln angehängt werden. Richtig wird es so:

Listing 8.24: Verlängerung einer Liste durch Shortcut (richtig)

```
1 >>> bierListe = ['Weizen','Pils']
2 >>> bier = 'Kölsch'
3 >>> bierListe += ['Kölsch']
4 >>> bierListe
5 ['Weizen', 'Pils', 'Kölsch']
```

8.2.5.2. bisect oder „divide et impera“

Ein interessantes Modul ist `bisect`. Es erwartet eine bereits sortierte Liste, und kann dann sehr effektiv bestimmen, an welcher Stelle ein neues Objekt in diese sortierte Liste einsortiert werden soll; der Name `bisect` beschreibt gut, was passiert: die sortierte Liste wird zweigeteilt, der eine Teil enthält alle Elemente, die kleiner (in Bezug auf das Sortierkriterium), die andere alle Elemente, die größer als das neue Element sind. Außerdem kann es das auch gleich durchführen. Hier kommt ein dazu passendes Beispiel.

Listing 8.25: Finden und einfügen mit `bisect`

```
1  import bisect
2  >>> woerter = ['ich','du', 'er', 'sie', 'es', 'wir']
3  >>> woerter.sort()
4  >>> woerter
5  ['du', 'er', 'es', 'ich', 'sie', 'wir']
6  >>> bisect.bisect(sorted(woerter),'ihr')
7  4
8  >>> bisect.insort(woerter, 'ihr')
9  >>> woerter
10 ['du', 'er', 'es', 'ich', 'ihr', 'sie', 'wir']
```

Besonders schön ist es, wenn ich zwei sortierte Listen habe und die Elemente der einen Liste an der richtigen Stelle in die andere Liste einsortieren möchte. Im folgenden Beispiel enthält die eine Liste die Schulfächer Mathematik, Informatik und einige Naturwissenschaften, die andere Liste die sprachlichen Fächer.

Listing 8.26: Zwei Listen sortiert vereinigen

```
>>> mn = ['Physik','Biologie','Chemie','Mathematik','Informatik']
>>> spr = ['Deutsch','Französisch','Englisch','Spanisch',
'Chinesisch','Portugiesisch']
>>> mn.sort()
>>> spr.sort()
>>> mn
['Biologie', 'Chemie', 'Informatik', 'Mathematik', 'Physik']
>>> spr
['Chinesisch', 'Deutsch', 'Englisch', 'Französisch', 'Portugiesisch',
'Spanisch']
>>> for eineSpr in spr:
        bisect.insort(mn,eineSpr)

>>> mn
['Biologie', 'Chemie', 'Chinesisch', 'Deutsch', 'Englisch', 'Französisch',
'Informatik', 'Mathematik', 'Physik', 'Portugiesisch', 'Spanisch']
```

Für ein weiteres Beispiel zu `bisect` benötige ich noch Dictionaries. Deswegen ist es noch nicht hier zu finden, sondern auf Seite 167

8.2.5.3. Ein gern gemachter Fehler

Wenn `l` eine Liste ist und man eine Methode von Listen auf `l` anwendet, wird `l` verändert, die Methode gibt eine Erfolgs- oder Mißerfolgsmeldung zurück. Der Satz hört sich kompliziert an. Darum gibt es gleich ein Beispiel:

Listing 8.27: Fehler bei Methoden von Listen

```
>>> l = [1,2,3,4,5]
>>> print(l)
[1, 2, 3, 4, 5]
>>> l.append(6)
>>> print(l)
[1, 2, 3, 4, 5, 6]

>>> l2 = l
>>> print(l)
[1, 2, 3, 4, 5, 6]
>>> print(l2)
[1, 2, 3, 4, 5, 6]

>>> l2 = l.append(7)
>>> print(l)
```

```
16 [1, 2, 3, 4, 5, 6, 7]
17 >>> print(l2)
18 None
```

Hast Du gesehen, was passiert? Der Befehl in Zeile 14 hängt die Zahl 7 an die bestehende Liste `l` an. Die Erfolgsmeldung dieser Operation wird der Variablen `l2` zugewiesen. Was man ohne die Information aus dem Satz vor diesem Beispiel vielleicht erwartet, nämlich dass die verlängerte Liste in der Variablen `l2` steht ... Pustekuchen.

8.2.5.4. Mehrere Elemente anhängen

Der erste Versuch dazu: wie oben müssen die Biere wieder herhalten.

Listing 8.28: Erweiterung einer Liste mit `append` durch eine Liste

```
1 >>> bierListe = ['Weizen']
2 >>> zweiPils = ['Pilsener Urquell', 'Bitburger Pils']
3 >>> bierListe.append(zweiPils)
4 >>> print(bierListe)
5 ['Weizen', ['Pilsener Urquell', 'Bitburger Pils']]
```

Die Liste wird als Liste angehängt! Das ist nicht gewünscht.

Während man mit `append` nur ein einzelnes Element an eine Liste anhängen kann, ist es möglich, mit `extend` mehr anzuhängen. Genau gesagt: an eine Liste kann mit `extend` alles Mögliche angehängt werden, sofern es iterierbar ist. Dieser Satz ist aus 2 Gründen gemein:

1. erst im nächsten Kapitel9 , dem über Strukturierte Programmierung, taucht der Begriff „Iteration“ auf

2. Da taucht ganz heimlich eine Art Rekursion auf: iterierbar bedeutet, dass etwas strukturiert ist, und dass das etwas abzählbar ist, in der Form, dass man sagen kann, welches das erste Element ist, und was dann das darauffolgende usw.

   Also was? Na klar, eine Liste!!! Auf eine Liste trifft genau das zu, was im vorigen Satz steht. Also kann ich eine Liste mit `extend` um eine Liste erweitern.

Hier kommt das Beispiel dazu:

Listing 8.29: Verlängerung einer Liste um eine Liste

```
1 >>> liste1 = ['a','b','c']
2 >>> print(liste1)
3 ['a','b','c']
4 >>> liste1.extend(['x','y','z'])
5 >>> print(liste1)
6 ['a','b','c','x','y','z']
```

Hier kommt nochmal, was passiert, wenn ich hingegen mit `append` eine Liste anhänge: die Liste wird als ein Element genommen und angehängt.

Listing 8.30: Verlängerung einer Liste um eine Liste geht schief

```
1 >>> liste1 = ['a','b','c']
2 >>> print(liste1)
3 ['a','b','c']
4 >>> liste1.append(['x','y','z'])
5 >>> print(liste1)
6 ['a','b','c',['x','y','z']]
```

Und um nochmals vorzugreifen: das funktioniert auch mit einem Tupel.

Listing 8.31: Verlängerung einer Liste um ein Tupel

```
1 >>> liste1 = ['a','b','c']
2 >>> print(liste1)
3 ['a','b','c']
4 >>> liste1.extend(('x','y','z'))
5 >>> print(liste1)
6 ['a','b','c','x','y','z']
```

Es ist zwar nicht möglich, an eine Liste, die 4 Elemente enthält (also die Elemente 0 bis 3) ein weiteres mit `liste[4] ='bla blub'` anzuhängen. Aber es gibt eine Möglichkeit, die so ähnlich aussieht:

Listing 8.32: Elemente an eine Liste anhängen

```
1 >>> vn = ['Martin', 'Hannah', 'Theresa']
2 >>> vn += ['Heiko']
3 >>> vn
4 ['Martin', 'Hannah', 'Theresa', 'Heiko']
5 >>> vn[len(vn):] = ['Anne', 'Maria']
6 >>> vn
7 ['Martin', 'Hannah', 'Theresa', 'Heiko', 'Anne', 'Maria']
8 >>> vn.append('Lorenz')
9 >>> vn
10 ['Martin', 'Hannah', 'Theresa', 'Heiko', 'Anne', 'Maria', 'Lorenz']
```

Die Verlängerung in Zeile 6 ist gemeint: hier wird gesagt, dass die Listenelemente ab dem Element mit der Hausnummer „Länge der Liste“ die Namen „Anne“ und „Maria“ sein sollen.

Für das Erweitern einer Liste um eine Liste gibt es allerdings noch eine „Kurzschreibweise“ ; aber vielleicht wärst Du da selber drauf gekommen, jetzt, nachdem Du Dich schon ein bißchen mit Python beschäftigt hast:

Listing 8.33: Verlängerung einer Liste um eine Liste (Kurzform)

```
1 >>> liste1 = ['a','b','c']
2 >>> print(liste1)
3 ['a','b','c']
4 >>> liste1 += ['x','y','z']
5 >>> print(liste1)
6 ['a','b','c','x','y','z']
```

Python verhält sich so, wie man es von einer vernünftigen Programmiersprache erwartet.

8.2.5.5. Ein Element einfügen

Hier geht es darum, an einer bestimmten Stelle ein Element einzufügen, also nicht mehr nur einfach an das Ende der Liste etwas hinzuzufügen. Das Beispiel erklärt sich selbst (finde ich):

Listing 8.34: Einfügen eines Elements in eine Liste

```
>>> liste2 = [1,2,4]
>>> liste2
[1, 2, 4]
>>> liste2.insert(2,'DREI')
>>> liste2
[1, 2, 'DREI', 4]
```

Eines oder mehrere Elemente in eine Liste einfügen geht aber auch noch auf eine andere Art. Denn Listen sind veränderlich, und so kann man einfach Python sagen, dass das Listenelement oder die Listenelemente an einer bestimmten Stelle einen Wert haben sollen. Hier kommt das Beispiel:

Listing 8.35: Einfügen eines oder mehrerer Elemente in eine Liste

```
>>> m = list(range(10))
>>> m
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> m[2:2] = [1000]
>>> m
[0, 1, 1000, 2, 3, 4, 5, 6, 7, 8, 9]
>>> m[5:5] = [222,333,444]
>>> m
[0, 1, 1000, 2, 3, 222, 333, 444, 4, 5, 6, 7, 8, 9]
>>> m[5:6] = [99, 98, 97, 96]
>>> m
[0, 1, 1000, 2, 3, 99, 98, 97, 96, 333, 444, 4, 5, 6, 7, 8, 9]
```

Die letzte Veränderung ist einen Kommentar wert: hier wird gesagt, dass an der 5. bis 6. Stelle etwas eingefügt werden soll. Das ist gleichbedeutend damit, das Element an der 5. Stelle rauszuschmeissen und dort die angegebenen Werte einzufügen.

Dieses Vorgehen kann man auch benutzen, um Elemente einer Liste zu löschen.

Listing 8.36: Einfügen eines oder mehrerer Elemente in eine Liste

```
>>> m = list(range(10))
>>> m
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> m[3:6] = []
>>> m
[0, 1, 2, 6, 7, 8, 9]
```

Entscheide jeweils selbst, wie du solche oder ähnliche Probleme löst.

8.2.5.6. Ein bestimmtes Element entfernen (nicht so schön)

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

Listing 8.37: Löschen eines Elements aus einer Liste

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

8.2.5.7. Ein bestimmtes Element entfernen (schöner)

Anstatt ein Element aus einer Liste über die Angabe der „Hausnummer“ zu entfernen ist es oft angebracht, ein Element über seinen Wert zu entfernen. Das sieht so aus:

Listing 8.38: Löschen eines Elements aus einer Liste über den Namen

```
>>> namen = ['Andi', 'Ben', 'Carlo', 'Dora', 'Elli']
>>> namen.remove('Ben')
>>> print(namen)
['Andi', 'Carlo', 'Dora', 'Elli']
```

Aber auch hier ist Vorsicht geboten: wenn ein Element mehrmals in der Liste auftaucht, wird nur das erste Vorkommen gelöscht!

Listing 8.39: Nur ein Element aus einer Liste wird über den Namen gelöscht

```
>>> namen = ['Andi', 'Ben', 'Carlo', 'Ben', 'Ben', 'Dora', 'Ben', 'Elli']
>>> namen.remove('Ben')
>>> print(namen)
['Andi', 'Carlo', 'Ben', 'Ben', 'Dora', 'Ben', 'Elli']
```

8.2.5.8. Ein bestimmtes Element bearbeiten und entfernen

Im vorigen Abschnitt habe ich die Art des Entfernens eines Elementes aus einer Liste als nicht so schön bezeichnet. Das versteht man, wenn man weiter fortgeschritten ist und sich mit Objektorientierung befasst hat. Schöner ist es, wenn man eine Methode der Klasse „Listen“ benutzt. Die Methode `pop()` holt ein einzelnes Element aus der Liste und löscht es darin. Wenn in der Klammer kein Parameterwert angegeben ist, also mit `liste1.pop()` wird automatisch das letzte Element entfernt. Will man etwa das nullte Element (denke daran: Python fängt bei 0 an zu zählen) entfernen, geschieht das mit `liste1.pop(0)`.

8.2.5.9. Eine Teilmenge der Liste bearbeiten

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

Listing 8.40: Teile einer Liste

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

Und etwas, was oft sehr hilfreich ist:

Listing 8.41: Umkehrung einer Liste

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

Man kann eine Liste umkehren und anzeigen und dabei die Liste aber im Originalzustand lassen; man kann aber auch die Liste umkehren und umgekehrt speichern.

Listing 8.42: Umkehrung (2 Arten)

```
>>> l = [1,2,3,4,5]
# hier wird die Liste umgekehrt angezeigt, aber belassen
>>> l[::-1]
[5, 4, 3, 2, 1]
>>> print(l)
[1, 2, 3, 4, 5]
# hier wird die Liste umgekehrt und geändert
>>> l.reverse()
>>> print(l)
[5, 4, 3, 2, 1]
```

Von Texten her bekannt ist schon das Slicing. Das funktioniert auch bei Listen. Hier wird mit 3 Parametern gearbeitet, um aus der Liste der ersten 10 natürlichen Zahlen die geraden Zahlen herauszufiltern.

Listing 8.43: Eine Liste slicen

```
>>> zahlenListe = [0,1,2,3,4,5,6,7,8,9]
>>> print(zahlenListe[0:9:2])
[0, 2, 4, 6, 8]
```

8.2.5.10. Eine Liste sortieren

Das ist wahrscheinlich eine der häufigsten Aufgaben, die bei einer Liste entstehen: sie soll sortiert werden. Zum Glück hat Python für Listen die Methode `sort`. Sie sortiert die Elemente nach einer lexikalischen Ordnung:

Listing 8.44: Sortierung einer Liste

```
>>> liste2 = ['x','B','S','P','A','n','m']
>>> liste2.sort()
>>> liste2
['A', 'B', 'P', 'S', 'm', 'n', 'x']
```

Dabei heißt „lexikalisch“ , dass im Sinne der ASCII-Ordnung sortiert wird, und da kommen alle Großbuchstaben AB ... XYZ vor den Kleinbuchstaben ab ... xyz. Beachte dabei: die Liste wird an Ort und Stelle sortiert, das heißt, dass die Liste nach dem Sortieren nur noch in der sortierten Form existiert.

Da eine Liste beim Sortieren an Ort und Stelle verändert wird, ist es nicht ganz trivial zu überprüfen, ob eine Liste sortiert ist. Durch die Funktion `sorted` wird eine neue Liste erzeugt, in die die Elemente der gegebenen Liste in sortierter Reihenfolge eingefügt werden:

Listing 8.45: Ist eine Liste sortiert?

```
1 >>> liste12 = [3,5,7,6,4,2]
2 >>> print(liste12 == sorted(liste12))
3 False
```

Die Funktion `sorted` erlaubt es also auch, eine sortierte Kopie einer gegebenen Liste zu erzeugen und die ursprüngliche Liste unangetastet zu lassen:

Listing 8.46: Sortierte Kopie einer Liste

```
1 >>> l = [1,6,3,9,2,8]
2 >>> l2 = sorted(l)
3 >>> l2
4 [1, 2, 3, 6, 8, 9]
5 >>> l
6 [1, 6, 3, 9, 2, 8]
```

Es ist natürlich auch möglich, ohne Ansehen der Groß-/Kleinschreibung zu sortieren:

Listing 8.47: Sortierung einer Liste ohne Unterscheidung von Groß-/ Kleinschreibung

```
1 >>> liste2 = ['x','B','S','P','A','n','m']
2 >>> liste2.sort(key=str.lower)
3 >>> liste2
4 ['A', 'B', 'm', 'n', 'P', 'S', 'x']
```

Durch die Angabe des Sortierschlüssels `key=str.lower` werden alle Elemente aufgefasst, als wären sie klein geschrieben. Merke: sie werden so aufgefasst, aber keineswegs geändert!!

Das kann man auch noch in anderer Form anwenden. Nach dem Schlüsselwort `key` kann nicht nur ein Ausdruck stehen, sondern auch der Name einer Funktion, die etwas Sortierbares zurückgibt. Auch dieses Beispiel steht weiter hinten im Skript, wo Funktionen beschrieben werden.

8.2.6. Tricks mit Listen

8.2.6.1. Matrizen

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!

Listing 8.48: 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 (Wer spickeln möchte: S. 222).

Jetzt kann man natürlich auf die Idee kommen, zum Beispiel die 5x5-Nullmatrix über mit den oben genannten Mitteln zu erstellen.

Listing 8.49: Nullmatrix

```
>>> nullzeile =[0,0,0,0,0]
>>> nullmatrix = [[nullzeile]*5]
>>> nullmatrix
[[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]]
```

Das sieht doch ganz gut aus!

Es geht aber noch kürzer:

Listing 8.50: Nullmatrix (ganz kurz)

```
>>> nm = [[0 for i in range(5)]]*5
>>> nm
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
```

Und auch das scheint korrekt zu sein.

Jetzt spreche ich die einzelnen Elemente der Matrix zuerst lesend an

Listing 8.51: Nullmatrix, 3.Zeile, 1. Spalte

```
>>> nm[3][1]
0
```

Alles in Ordnung! Dann schreibe ich doch mal in die 3. Zeile 1. Spalte einen neuen Wert:

Listing 8.52: Nullmatrix, 3.Zeile, 1. Spalte ändern

```
>>> nm[3][1] = 99
>>> nm[3][1]
99
```

Das scheint auch gut funktioniert zu haben. Schaue ich mir also nochmal die gesamte Liste an:

Listing 8.53: geänderte Nullmatrix

```
>>> nm
[[0, 99, 0, 0, 0], [0, 99, 0, 0, 0], [0, 99, 0, 0, 0], [0, 99, 0, 0, 0],
[0, 99, 0, 0, 0]]
```

Es scheint nur gut funktioniert zu haben! Beim Erzeugen der Matrix habe ich nämlich nur eine 5-fache Kopie einer Nullzeile gemacht. Konkret: es existiert nur eine einzige

Zeile, die 5 mal kopiert die Matrix ergibt. Wenn ich jetzt also ein Element der Matrix ändere, ändere ich effektiv das angegebene Spaltenelement **jederZeile**

Da hilft wieder nur, eine Kopie einer Nullzeile (vom ersten bis zum letzten Element) an eine leere Liste anzuhängen:

Listing 8.54: Erzeugung und Änderung einer Matrix

```
>>> nullzeile = [0 for i in range(5)]
>>> nullzeile
[0, 0, 0, 0, 0]
>>> nullmatrix = []
>>> for i in range(5):
        nullmatrix.append(nullzeile[:])
>>> nullmatrix
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
>>> nullmatrix[1][1] = 77
>>> nullmatrix
[[0, 0, 0, 0, 0], [0, 77, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]
```

Nein, das „nicht nur“ aus dem vorigen Absatz gilt nicht. Es gibt eine elegante und pythonische Lösung, und zwar mit Hilfe einer Comprehension.(siehe auch bei List Comprehensions)

Listing 8.55: Erzeugung einer Matrix mit List Comprehension

```
>>> nullmatrix = [[0] * 4 for i in range(4)]
>>> nullmatrix
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> nullmatrix[1][1] = 77
>>> nullmatrix
[[0, 0, 0, 0], [0, 77, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Hier wird nicht mehr eine Zeile 3 mal kopiert, sondern wirklich 4 unterscheidbare Zeilen erzeugt. Die „77“ steht nach der Änderung wirklich nur in einer Zelle.

Außerdem kann man das Problem auch dadurch lösen, dass man die Matrix in einer for-Schleife erzeugt, in der jedesmal die Zeile explizit geschrieben ist. Unschön!

8.2.6.2. Listenelemente mit Namen ansprechen

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:

Listing 8.56: 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:

Listing 8.57: 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:

Listing 8.58: 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:

Listing 8.59: Ein Trick bei der Listen-Bearbeitung

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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

allePersonen = [ich, hannah, leni]

vorname, nachname, schuhgroesse = range(3)

print(allePersonen[0][vorname])

for einer in allePersonen:
  print(einer[vorname]+' '+einer[nachname])
```

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!

8.2.6.3. Bestimmte Elemente aus Liste herausziehen

s. /bin/python3/07ListenStrings/teileAusListe...

8.2.6.4. Richtige Datentypen aus einer Zeichenkette

Eigenartiger Titel, aber hier kommt die Erklärung. Angenommen, ich habe eine Liste, in der die Listenelemente Textzeilen sind, zum Beispiel eine csv-Liste (comma seperated values). Jede Zeile ist auf die selbe Art und Weise aufgebaut: der erste Eintrag ist eine Zeichenkette, der zweite eine Dezimalzahl und der dritte eine Ganzzahl. Es ist jetzt kein Kunststück, diese Einträge einer Zeile aufzusplitten:

Listing 8.60: Liste von Texten, diese sollen gesplittet werden

```
1 #!/usr/bin/python
2
3 zeilen = ["Martin, 1.91, 75",  "Klaus, 1.72, 55", "Hannes, 1.82, 66"]
4
5 for eineZeile in zeilen:
6     felderOF = eineZeile.split(',')
7     print(felderOF)
```

Die Ausgabe sieht halbwegs gut aus:

Listing 8.61: Ausgabe: halbwegs gutes Splitten

```
['Martin', ' 1.91', ' 75']
['Klaus', ' 1.72', ' 55']
['Hannes', ' 1.82', ' 66']
```

Das ist wirklich nur halbwegs gut, denn auch die Zahlen werden als Strings wiedergegeben.

Die Verbesserung ist jetzt, dass ich eine Musterzeile erstelle, diese mit der Datenzeile „zippe“ und erst dann ausgebe:

Listing 8.62: Liste von Texten, diese sollen gesplittet werden, jetzt richtig

```
>>> zeilen = ["Martin, 1.91, 75", "Klaus, 1.72, 55", "Hannes, 1.82, 66"]
>>> muster = [str, float, int]

>>> for eineZeile in zeilen:
        felderFormat = [typ(wert) for typ, wert in zip(muster, eineZeile.split('
        print('in 2. for-schleife', felderFormat)
        for feld in felderFormat:
            print('in innerer for-Schleife', feld, type(feld))
        print()

in for-schleife ['Martin', 1.91, 75]
in innerer for-Schleife Martin <class 'str'>
in innerer for-Schleife 1.91 <class 'float'>
in innerer for-Schleife 75 <class 'int'>

in for-schleife ['Klaus', 1.72, 55]
in innerer for-Schleife Klaus <class 'str'>
in innerer for-Schleife 1.72 <class 'float'>
in innerer for-Schleife 55 <class 'int'>

in for-schleife ['Hannes', 1.82, 66]
in innerer for-Schleife Hannes <class 'str'>
in innerer for-Schleife 1.82 <class 'float'>
in innerer for-Schleife 66 <class 'int'>

>>> for eineZeile in zeilen:
        print(eineZeile)

Martin, 1.91, 75
Klaus, 1.72, 55
Hannes, 1.82, 66
```

Gut so!

8.2.6.5. Mehrere Listen auf einmal bearbeiten

Das Problem soll gelöst werden, dass in mehreren Listen jeweils die ersten, zweiten ... Elemente zusammengefügt werden sollen. Dazu benutzt man die Funktion `zip`.

Listing 8.63: Listen vermischen

```
1 >>> festtage = ['Ostern','Nikolaus','Weihnachten']
2 >>> figuren = ['Osterhasi','Nikolausi', 'Christkind']
3 >>> geschenke = ['Nest mit Ostereiern', 'Stiefel mit Nüssen','viele Bücher']
4 >>> for tag,figur,geschenk in zip(festtage,figuren,geschenke):
5         print('An ',tag, 'bringt ', figur, 'als Geschenk ', geschenk )
6
7 An  Ostern bringt  Osterhasi als Geschenk  Nest mit Ostereiern
8 An  Nikolaus bringt  Nikolausi als Geschenk  Stiefel mit Nüssen
9 An  Weihnachten bringt  Christkind als Geschenk  viele Bücher
```

8.2.7. Kopie einer Liste

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

Listing 8.64: Kopie einer Liste (so klappt es nicht!)

```
1 >>> meineListe = ['Apfel','Birne', 'Kohl', 'Gurke']
2 >>> kopie = meineListe
3 >>> print('meine Liste = ', meineListe)
4 >>> print('Kopie = ', kopie)
```

Das ergibt

Listing 8.65: Kopie einer Liste

```
1 meine Liste = ['Apfel','Birne', 'Kohl', 'Gurke']
2 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:

Listing 8.66: Kopie einer Liste (erster Versuch)

```
1 >>> meineListe[2] = 'Sauerkraut'
2 >>> print('meine Liste = ', meineListe)
3 >>> print('Kopie = ', kopie)
```

Und man liest

Listing 8.67: Kopie einer Liste: so wollte ich es nicht

```
1 meine Liste =  ['Apfel', 'Birne', 'Sauerkraut', 'Gurke']
2 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“ .

Listing 8.68: 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.

Listing 8.69: 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']
```

8.2.8. Nicht mehr ganz so wilde Listen

Zu Beginn des Kapitels über Listen habe ich das als eine schöne Eigenschaft herausgestellt, dass Listen beliebige Elemente enthalten können. Manchmal ist das aber nicht wünschenswert. Um Listen zu erzeugen, die nur einen bestimmten Typ Daten enthalten dürfen, benutzen wir den Modul `array`.

Ein Array-Objekt erhält beim Erzeugen als ersten Parameter ein Kürzel für die Klasse von Objekten, die es enthalten darf.

Listing 8.70: Ein Zahlen-Array ... und was damit nicht geht

```
>>> import array
>>> meinArray = array.array('i',[i for i in range(20)])
>>> print(meinArray)
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
    16, 17, 18, 19])

>>> meinArray.append('a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: an integer is required (got type str)

>> meinArray.append(1.2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: integer argument expected, got float
```

Erzeugt wurde das Array mittels einer Comprehension. Und Zeichenketten oder Dezimalzahlen können nicht angehängt werden, weil der erste Parameter beim Erzeugen, das `'i'`, für integer steht, also für Ganzzahlen.

Die Hilfe zu `array.array` liefert unter anderem folgendes:

Listing 8.71: Hilfe zu array.array

```
>>> help(array.array)

Help on class array in module array:

class array(builtins.object)
 |  array(typecode [, initializer]) -> array
 |
 |  Return a new array whose items are restricted by typecode, and
 |  initialized from the optional initializer value, which must be a list,
 |  string or iterable over elements of the appropriate type.
 |
 |  Arrays represent basic values and behave very much like lists, except
 |  the type of objects stored in them is constrained. The type is specified
 |  at object creation time by using a type code, which is a single character.
 |  The following type codes are defined:
 |
 |      Type code   C Type             Minimum size in bytes
 |      'b'         signed integer     1
 |      'B'         unsigned integer   1
 |      'u'         Unicode character  2 (see note)
 |      'h'         signed integer     2
 |      'H'         unsigned integer   2
 |      'i'         signed integer     2
 |      'I'         unsigned integer   2
 |      'l'         signed integer     4
 |      'L'         unsigned integer   4
 |      'q'         signed integer     8 (see note)
 |      'Q'         unsigned integer   8 (see note)
 |      'f'         floating point     4
 |      'd'         floating point     8
```

8.2.9. Wie und wo gespeichert wird (jetzt bei Listen)

> Bin ich? Und wenn ja, immer noch genau so viele?
>
> *(noch freier nach R.D. Precht)*

Nein, das ist kein Fehler, dass der Titel und das Motto zum zweiten Mal auftaucht. Bereits weiter oben bei Zeichenketten wurde kurz angerissen, wie es mit der Speicherung von Objekten funktioniert. Bei Listen gibt es Kleinigkeiten, auf die man achten muss.

Listing 8.72: 2 Texte und 2 Listen

```
>>> text1 = 'Martin'
>>> text2 = 'Martin'
>>> id(text1)
140440488529344
>>> id(text2)
140440488529344
>>> liste1 = ['M', 'a', 'r', 't', 'i', 'n']
>>> liste2 = ['M', 'a', 'r', 't', 'i', 'n']
>>> liste3 = liste1
>>> liste4 = liste1[:]
>>> id(liste1)
140440488533064
>>> id(liste2)
140440488533128
>>> id(liste3)
140440488533064
>>> id(liste4)
140440488533576
>>> text1 is text2
True
>>> liste1 is liste2
False
>>> liste1 is liste3
True
>>> liste1 is liste4
False
```

Zusammengefasst: 2 Text-Variablen mit dem selben Inhalt weisen auf das selbe Objekt; 2 Listen-Variablen mit dem selben Inhalt weisen auf verschiedene Objekte. Das Problem der Kopie einer Liste (siehe weiter oben) ist damit auch geklärt: Die Kopie einer Liste mit `l2 = l1` erstellt nur eine Referenz auf den Speicherplatz des Objektes `l1`, die Kopie mit `l1[:]` erstellt ein neues Objekt.

8.3. Dictionaries

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

(Bob Dylan [4])

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.

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

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, manchmal auch beides:

```
1 >>> englDeutsch.keys()
2 ['sheep', 'dog', 'cow', 'cat']
3 >>> englDeutsch.values()
4 ['Schaf', 'Hund', 'Kuh', 'Katze']
5 >>> englDeutsch.items()
6 dict_items([('sheep', 'Schaf'), ('cat', 'Katze'),
7 ('dog', 'Hund'), ('cow', 'Kuh')])
```

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

8.3.0.1. Die Frage kommt immer wieder ...

Aber wie mache ich jetzt aus einem Englisch-Deutsch-Wörterbuch ein Deutsch-Englisch-Wörterbuch?

Die Funktion `zip` (wie Reißverschluss) hilft hier weiter. Sie nimmt 2 Listen und verknüpft die Elemente der beiden Listen paarweise. Diese Verknüpfung muss jetzt als Dictionary gespeichert werden.

Beispiel 8.1 aus Englisch-Deutsch mache Deutsch-Englisch

```
1 >>> englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh', 'sheep':'Schaf'}
2 >>> deutschEngl = dict(zip(englDeutsch.values(), englDeutsch.keys()))
3 >>> print(deutschEngl)
4 {'Hund': 'dog', 'Katze': 'cat', 'Kuh': 'cow', 'Schaf': 'sheep'}
```

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

8.3.1. Zugriff auf Dictionary-Elemente

8.3.1.1. Dictionary lesen

Beispiel 8.2 Zugriff auf das obige Wörterbuch

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

Der Versuch des Zugriffs auf ein Dictionary über einen Schlüssel, der nicht existiert, führt aber zu einem sehr unerwünschten Abbruch des Programms:

Beispiel 8.3 erfolgloser Zugriff auf ein Dictionary

```
>>>  englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh', 'sheep':'Schaf'}
>>> englDeutsch['bird']
Traceback (innermost last):
  File "<stdin>", line 1, in <module>
KeyError: 'bird'
```

Die Fehlermeldung ist hier deutlich: Zugriff mit einem Schlüssel, der nicht existiert.

Dictionaries haben aber die Zugriffsmethode `get`, der man einen Defaultwert mitgeben kann:

Beispiel 8.4 Abfangen eines erfolglosen Zugriffs auf ein Dictionary

```
>>> englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh', 'sheep':'Schaf'}
>>> englDeutsch.get('bird','bird ist nicht im Dictionary')
'bird ist nicht im Dictionary'
```

Eine weitere Methode, `setdefault`, leistet noch mehr. Während `get` nur einen Default-Wert ausgibt, kann man mit `setdefault` einen Neueintrag in das Dictionary machen:

Beispiel 8.5 Was fehlt, wird ergänzt

```
>>> englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh', 'sheep':'Schaf'}
>>> englDeutsch.get('bird','bird ist nicht im Dictionary')
'bird ist nicht im Dictionary'
>>> englDeutsch.setdefault('bird','Vogel')
'Vogel'
>>> englDeutsch.get('bird','bird ist nicht im Dictionary')
'Vogel'
```

Man sieht: beim ersten Aufruf von `get` wird nur das Fehlen angezeigt, beim zweiten Aufruf ist der Wert vorhanden. Man muss auch keine Sorge haben, dass ein bereits bestehender Wert überschrieben wird:

```
>>> englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh', 'sheep':'Schaf'}
>>> englDeutsch.get('bird','bird ist nicht im Dictionary')
'bird ist nicht im Dictionary'
>>> englDeutsch.setdefault('bird','Vogel')
'Vogel'
>>> englDeutsch.get('bird','bird ist nicht im Dictionary')
'Vogel'
>>> englDeutsch.setdefault('bird','Nashorn')
'Vogel'
>>> englDeutsch.get('bird','bird ist nicht im Dictionary')
'Vogel'
```

Das letzte `get` wäre sogar überflüssig gewesen: `setdefault` setzt einen neuen Wert, falls der Eintrag im Dictionary noch nicht existiert; falls er existiert, wird der Wert zurückgegeben.

Mit Hilfe dieser Methode ist es elegant möglich, etwa die Buchstaben häufigkeit eines Textes zu zählen. Man legt ein leeres Dictionary an und addiert 1 dazu bei jedem Auftreten eines Buchstaben. Falls in dem Dictionary noch kein Eintrag zu einem Buchstaben existiert, wird mit „get“ der Wert 0 eingetragen. Für die Ausgabe dieses und der beiden nächsten Schnipsel benötigen wir eine Schleife, die aber erst später im Text im Kapitel 9.5.1 kommt.

```
>>> bs = 'Dies ist ein wirklich doofer Satz mit vielen Buchstaben'
>>> anzB = {}
>>> for b in bs:
...             anzB[b] = anzB.get(b,0) + 1
...
>>> anzB
{'D': 1, 'i': 7, 'e': 6, 's': 3, ' ': 8, 't': 4, 'n': 3, 'w': 1,
'r': 2, 'k': 1, 'l': 2, 'c': 2, 'h': 2, 'd': 1, 'o': 2, 'f': 1,
'S': 1, 'a': 2, 'z': 1, 'm': 1, 'v': 1, 'B': 1, 'u': 1, 'b': 1}
```

Das ist nicht schlecht, aber bei der Häufigkeit von Buchstaben interessiert Groß-/Kleinschreibung nicht. Besser wäre also:

```
>>> bs = 'Dies ist ein wirklich doofer Satz mit vielen Buchstaben'
>>> anzB = {}
>>> for b in bs:
...     anzB[b.lower()] = anzB.get(b.lower(),0) + 1
...
>>> print(anzB)
{'d': 2, 'i': 7, 'e': 6, 's': 4, ' ': 8, 't': 4, 'n': 3, 'w': 1,
'r': 2, 'k': 1, 'l': 2, 'c': 2, 'h': 2, 'o': 2, 'f': 1, 'a': 2,
'z': 1, 'm': 1, 'v': 1, 'b': 2, 'u': 1}
```

Und jetzt folgt die Komfort-Version, nämlich die alphabetische Ausgabe der Häufigkeiten. Die Schlüssel werden dazu in einer Liste gespeichert, sortiert, und über diese sortierte Liste wird geschleift:

Beispiel 8.6 Buchstabenhäufigkeit bestimmen und alphabetisch ausgeben

```
>>> bs = 'Dies ist ein wirklich doofer Satz mit vielen Buchstaben'
>>> anzB = {}
>>> for b in bs:
...     anzB[b.lower()] = anzB.get(b.lower(),0) + 1
...
>>> schluessel = list(anzB.keys())
>>> schluessel.sort()
>>> for einSchl in schluessel:
...     print(einSchl,':',anzB[einSchl])
...
  : 8
a : 2
b : 2
c : 2
d : 2
e : 6
f : 1
etc.
```

(In der ersten Zeile der Ausgabe steht das Leerzeichen, das 8 mal auftaucht!!!)

Wie oben angedeutet gibt es noch eine (unvollständige) Lösung für die „Drei Chinesen mit dem Kontrabass“ . Fast, denn `translate` erlaubt als zu ersetzendes Objekt nur einen einzelnen Buchstaben. Die Diphtong-Problematik bleibt bestehen. Für `translate` muss man zuerst eine Übersetzungstabelle bauen. Für die Syntax der Übersetzungstabelle sollte man die Hilfe mit `help(str.maketrans)` nachschlagen. Ich benutze hier nur die erste Möglichkeit, bei der die Übersetzungstabelle ein Dictionary ist.

Beispiel 8.7 Drei Chinesen

Listing 8.73: Drei Chinesen mit `translate`

```
>>>          text = '''Drei Chinesen mit dem Kontrabass
... sassen auf der Straße und erzählten sich etwas
... da kam ein Polizist "Ei was ist denn das"
... Drei Chinesen mit dem Kontrabass'''
>>> nb = input('neuer Vokal: ')
neuer Vokal: i
>>> ueTab = ''.maketrans({'i':nb,'e':nb, 'a':nb,'o':nb,'u':nb, 'ä':nb})
>>> print(text.translate(ueTab))
Drii Chinisin mit dim Kintribiss
sissin iif dir Strißi ind irzihltin sich itwis
di kim iin Pilizist "Ei wis ist dinn dis"
Drii Chinisin mit dim Kintribiss
```

Mit `itertools` und Diphtong-Liste:

Beispiel 8.8 Drei Chinesen ... mit richtiger Ersetzung der Diphtonge

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import itertools

vokalNeu = input('Welcher Vokal soll es denn sein? ')
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" '''

vokale = 'aeiouäöü'
diphtonge = itertools.product(vokale,vokale)
diphtongeStrings = [str(x[0])+str(x[1]) for x in diphtonge]

for einDiph in diphtongeStrings:
    textNeu = text.replace(einDiph, vokalNeu)
    text = textNeu

for einVokal in vokale:
    textNeu = text.replace(einVokal, vokalNeu)
    text = textNeu

print(textNeu)
```

8.3.1.2. Dictionary schreiben

Dictionaries sind veränderbar. Den „bird“ kann man also locker einfügen durch

```
>>> englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh',
         'sheep':'Schaf'}
>>> englDeutsch.get('bird','bird ist nicht im Dictionary')
'bird ist nicht im Dictionary'
>>> englDeutsch['bird'] = 'Vogel'
>>> englDeutsch.get('bird','bird ist nicht im Dictionary')
'Vogel'
>>>
```

Der Modul `collections` enthält verschiedene Erweiterungen zu strukturierten Daten, so zum Beispiel das `defaultdict`. Das ist ein Dictonary, dem ein Defaultwert beim Anlegen zugeteilt werden kann. Das soll an einem einfachen Beispiel gezeigt werden, bei dem die Vornamen einer Person der Schlüssel, der Wohnort dieser Person der Wert sein soll. Leider wohnen die meisten betreffenden Personen nicht in München, Gilching oder Starnberg, sondern in Oberpfaffenhofen; aber diesen langen Ortsnamen will niemand öfter als nötig schreiben. Also schreibe ich eine Funktion `oph_zurueckgeben`, die diesen Namen zurückgibt.

Listing 8.74: defaultdict: langer Ortsname

```
>>> from collections import defaultdict
>>> def oph_zurueckgeben():
...     return 'Oberpfaffenhofen'
...
>>> wohnorte = defaultdict(oph_zurueckgeben)
>>> wohnorte['Paul']
'Oberpfaffenhofen'
>>> wohnorte['Jens'] = 'Gilching'
>>> wohnorte['Leni'] = 'Pasing'
>>> wohnorte['Lea']
'Oberpfaffenhofen'
>>> for eintrag in wohnorte:
...     print(eintrag,'wohnt in',wohnorte[eintrag])
...
Paul wohnt in Oberpfaffenhofen
Jens wohnt in Gilching
Leni wohnt in Pasing
Lea wohnt in Oberpfaffenhofen
```

So kann man sich schon viel Schreibarbeit sparen!

Hier kommt nochmals ein Beispiel, das (wieder einmal) ein Wörterbuch anlegt. Die Übersetzung wird innerhalb einer Funktion eingegeben.

Listing 8.75: defaultdict: Wörterbuch

```
>>> import collections
>>> def uebersetze():
        ue = input('Übersetzung: ')
        return ue

>>> d_Dt_Engl = collections.defaultdict(uebersetze)
>>> d_Dt_Engl['Apfel']
Übersetzung: apple
'apple'
>>> d_Dt_Engl['Kuh']
Übersetzung: cow
'cow'
>>> for x in d_Dt_Engl:
            print(x, '==>',d_Dt_Engl[x])
Apfel ==> apple
Kuh ==> cow
```

Das Beispiel sollte jetzt einmal benutzt werden, um ein „selbstlernendes“ Programm zu schreiben, selbstlernend in Anführungsstrichen, weil es für den Fall, dass etwas noch nicht im Dictionary enthalten ist, einen neuen Eintrag macht.

8.3.2. Dictionaries sortiert ausgeben

Listing 8.76: so geht es

```
>>> di2 = {'Orangensaft':1.30, 'Apfelsaft':0.80, 'Mangosaft':2.25,
           'Birnensaft':1.85, 'Kirschsaft':2.14}
>>> for k in sorted(di2.keys()):
        print(k, di2[k])

Apfelsaft 0.8
Birnensaft 1.85
Kirschsaft 2.14
Mangosaft 2.25
Orangensaft 1.3
```

8.3.3. Was ist denn das „irgendwas“ , das in einer Liste oder einem Dictionary stehen kann?

Alles!

Listing 8.77: Das alles (und noch viel mehr)

```
#!/usr/bin/python

import math

print('Arbeit mit der "wildenListe"')
wildeListe = [169, math.sqrt, math, math.pi]

print(wildeListe[1](wildeListe[0]))
print(wildeListe[2].sin(wildeListe[3]/2))

print('Arbeit mit dem "wildenDictionary"')
wildesDic = {'zahl1': 37, 'mm':math, 'wurzel':math.sqrt, 'pi':math.pi}

print(wildesDic['wurzel'](wildesDic['zahl1']))
print(wildesDic['mm'].sin(wildesDic['pi']/2))
```

Da steht also in der wilden Liste an Position 0 eine Ganzzahl, an Position 1 aus dem Modul `math` die Wurzel-Funktion `sqrt`, an Position 2 der gesamte Modul `math` und an Position 3 die aus dem Modul `math` importierte Konstante pi. Das wilde Dictionary ist sehr ähnlich aufgebaut, nur im Verhältnis zur Liste wurden der gesamte Modul und die Wurzel vertauscht.

Listing 8.78: Das alles funktioniert

```
Arbeit mit der "wildenListe"
13.0
1.0
Arbeit mit dem "wildenDictionary"
6.082762530298219
1.0
```

8.3.4. Tricks mit Dictionaries

8.3.4.1. Verschlüsselung durch Erzeugung eines Dictionaries mit zip

Ein anderes nettes Beispiel ist die Caesar-Verschlüsselung. Dazu baue ich ein Dictionary mittels **zip** auf, dessen Schlüssel die Kleinbuchstaben und dessen Werte die Zahlen 0 bis 25 sind. Dieses Dictionary drehe ich um, so dass Schlüssel zu Werten und Werte zu Schlüsseln werden, um den Geheimbuchstaben herauszufinden.

Beispiel 8.9 Caesar-Verschlüsselung

```
#!/usr/bin/python

import string

klartext = 'Caesar ist doof'

caesar = dict(zip(string.ascii_lowercase,range(len(string.ascii_lowercase))))
caesar_invers = dict(zip(caesar.values(),caesar.keys()))
verschiebung = 3

geheimtext = ''
for b in klartext:
    if b.lower() in caesar:
        index = caesar[b.lower()] + verschiebung
        geheimtext += caesar_invers[index % len(string.ascii_lowercase)]
    else:
        geheimtext += b

print()
print('Klartext: \t\t%s\nGeheimtext: %s' % (klartext, geheimtext))
```

Die Ausgabe ist:

Listing 8.79: Ausgabe der Caesar-Verschlüsselung

```
Klartext:   Caesar ist doof
Geheimtext: fdhvdu lvw grri
```

8.3.4.2. Dictionary Comprehension

Wie bei Listen können auch Dictionaries über eine Comprehension erzeugt werden. Hier wird das demonstriert für ein Dictionary, das als Schlüssel die Zahl, als Wert ihre Quadratzahl enthält.

Listing 8.80: Erzeugung eines Dictionaries Zahl / Quadratzahl

```
qDic  = dict((x, x*x) for x in range(1,21))
for z in qDic:
    print(z,  qDic[z])
```

Das kann auch dadurch geschehen, dass man für die Berechnung des Wertes eine Funktion aufruft:

Listing 8.81: Erzeugung eines Dictionaries Zahl / Quadratzahl über eine Funktion

```
def qZahlBerechnen(x):
    return x*x

qDic  = dict((x, qZahlBerechnen(x)) for x in range(1,21))
for z in qDic:
    print(z,  qDic[z])
```

Listing 8.82: Erzeugung eines Dictionaries über Funktion und `zip`

```
def qZahlBerechnen(x):
    return x*x

r = range(1, 21)
qDic = dict(zip(r, map(qZahlBerechnen,  r)))

for z in qDic:
    print(z, qDic[z])
```

Eine schöne Anwendung einer dictionary comprehension ist die Umkehrung eines Dictionaries, speziell eines Wörterbuchs. Umkehrung ist hier so gemeint, dass aus einem Englisch-Deutsch-Dictionary ein Deutsch-Englisch-Dictionary wird.

Listing 8.83: Erzeugung eines umgekehrten Dictionaries

```
englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh', 'sheep':'Schaf'}
deutschEngl = {deutsch:englisch for englisch, deutsch in englDeutsch.items()}
```

So kurz und knapp! Die zweite Zeile dieses Beispiels ist die Umwandlung: ein Item in dem ursprünglichen Dictionary besteht aus einem Paar englischesWort : deutschesWort; die beiden Wörter werden als ein umgekehrtes Paar in das neue Dictionary geschrieben.

8.3.4.3. Dictionaries und Listen und `bisect`

Hier kommt noch ein anderes Beispiel aus der Schule: die Bewertung von Klassenarbeiten. Ein Schüler bekommt für die Aufgaben eine Summe von Punkten. Name des Schülers und Anzahl der Punkte stehen in einem Dictionary. Ferner gibt es noch einen Iterator für die Noten und eine Liste von Grenzwerten, ab denen es die nächstbessere Note gibt. Das Programm sieht so aus:

Listing 8.84: `bisect` für Notenfindung

```
#!/usr/bin/python3

import bisect
noten = range(6, 0, -1)

ergebnisse = {'Karl':32, 'Suse':25, 'Carola':12, 'Emil':35, 'Franz':6,
    'Ilse':38, 'Ina':21}
grenzenFuerNoten = [8, 16, 22, 28, 36]

def notenBerechnen(einErg, gr, n):
    eineN = n[bisect.bisect(gr, einErg)]
    return einErg, eineN

for einErg in ergebnisse:
    erg, note = notenBerechnen(ergebnisse[einErg], grenzenFuerNoten, noten)
    print(einErg+': '+str(erg) + ' Punkte ergibt die Note '+ str(note))
```

Und so sieht die Ausgabe aus:

Listing 8.85: Ausgabe der Noten

```
Karl: 32 Punkte ergibt die Note 2
Suse: 25 Punkte ergibt die Note 3
Carola: 12 Punkte ergibt die Note 5
Emil: 35 Punkte ergibt die Note 2
Franz: 6 Punkte ergibt die Note 6
Ilse: 38 Punkte ergibt die Note 1
Ina: 21 Punkte ergibt die Note 4
```

8.4. Tupel

8.4.1. Allgemeines zu Tupeln

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:

```
1 >>>  ding1 , ding2 = ding2 , ding1
```

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

8.4.1.1. Erzeugen von Tupeln

Tupel kann man auf verschiedene Art erzeugen. Am einfachsten ist es, die Elemente in runden Klammern aufzulisten.

```
1 >>> tupel1 = (1,2,3)
```

Zu beachten dabei ist aber, dass für den Fall, dass das Tupel nur ein Element enthalten soll, hinter diesem einzigen Element ein Komma steht. Hier kommt das Gegenbeispiel:

```
1 >>> tupel4 = (4)
2 >>> type(t4)
3 <class 'int'>
```

Ein einziges Element, eine Zahl, ein Komma vergessen: das ist kein Tupel, sondern eine Integer-Zahl. Richtig ist es so:

```
1 >>> tupel5 = (5,)
2 >>> type(t5)
3 <class 'tuple'>
```

Wenn man ein Tupel mit mehr als einem Element erzeugt, kann man hinter das letzte Element ein Komma setzen . . . oder es sein lassen. Wenn es gesetzt wird, wird es großzügig ignoriert. Aus diesem Grund ist es eine sinnvolle Entscheidung, hinter das letzte Element ein Komma zu setzen: bei der Veränderung des Quellcodes vermeidet man unter Umständen dadurch Fehler.

Es geht noch ein bißchen einfacher ... dafür muss man etwas besser aufpassen: man kann bei der Deklaration eines Tupels auch die Klammern weglassen. Das haben wir schon ein paar Zeilen weiter oben gesehen bei der Vertauschung der Werte von 2 Variablen. Dieser Trick funktioniert auch beim Umsortieren von beliebig vielen Elementen, zum Beispiel nach dem Motto „die Ersten werden die Letzten sein“.

Listing 8.86: Belgischer Kreisel

```
1 >>> t6 = 6,
2 >>> t7 = 7,
3 >>> t8 = 8,
4 >>> t9 = 9,
5 >>> t6, t7, t8, t9 = t7, t8, t9, t6
6 >>> print (t6, t7, t8, t9)
7 (7,) (8,) (9,) (6,)
```

Radrennfahrer kennen so etwas: Belgischer Kreisel

Merke: nicht die runden Klammern, sondern die Kommata sind das Merkmal eines Tupels.

Listing 8.87: Tupel oder kein Tupel?

```
1 >>> tupel1 = (1,2,3)
2 >>> type(tupel1)
3 <class 'tuple'>
4 >>> tupel11 = 1,2,3
5 >>> type(tupel11)
6 <class 'tuple'>
7 >>> tupel12 = 1,2,3,
8 >>> type(tupel12)
9 <class 'tuple'>
10 >>> tupel2 = (222)
11 >>> type(tupel2)
12 <class 'int'>
13 >>> tupel21 = ('222')
14 >>> type(tupel21)
15 <class 'str'>
```

Eine weitere Möglichkeit, Tupel zu erzeugen, besteht darin den Tupel-Konstruktor aufzurufen (schon wieder ein Vorgriff auf Objektorientierung). Hier wird dadurch aus einer Liste ein Tupel gemacht:

```
1 >>> schulfaecher = tuple(['Mathematik', 'Physik', 'Informatik'])
2 >>> print(schulfaecher)
3 ('Mathematik', 'Physik', 'Informatik')
4 >>> type(schulfaecher)
5 <class 'tuple'>
```

Was passiert, wenn ich dem Tupel-Konstruktor einen Text mitgebe? Das:

```
1 >>> schoenstesFach = tuple('Mathematik')
```

```
2 >>> type(schoenstesFach)
3 <class 'tuple'>
4 >>> print(schoenstesFach)
5 ('M', 'a', 't', 'h', 'e', 'm', 'a', 't', 'i', 'k')
```

Hoffentlich klar: der Text „Mathematik“ ist eine Liste von Buchstaben, und s.o. . . .

8.4.2. Benannte Tupel

Eine interessante Anwendung für Tupel sind die benannten Tupel. Mit ihnen kann man komplexe Datenstrukturen abbilden. Okay, das macht man in einer objektorientierten Sprache wie Python über Klassen, aber manchmal ist ein benanntes Tupel auch ganz effektiv, etwa wenn man weiß, dass das ähnliche Objekte ähnliche Eigenschaften haben, aber das Verhalten dieser Objekte uninteressant ist.

Als Beispiel dienen hier Fahrzeuge. Die `namedtuple` gehören nicht zum Python-Kern, müssen also importiert werden. Ein benanntes Tupel wird angelegt, indem man ihm einen Namen und eine Liste von Eigenschaften gibt. Dann legt man ein Exemplar an, indem man den Eigenschaften einen Wert zuweist. Jetzt kann man die Eigenschaften über ihren Namen abrufen:

Listing 8.88: Ein benanntes Tupel namens Auto

```
1 >>> auto = namedtuple('Auto', 'modell leistung farbe')
2 >>> meinSchrotti = auto('Peugeot', '60 PS', 'blau')
3 >>> meinSchrotti.modell
4 'Peugeot'
5 >>> meinSchrotti.leistung
6 '60 PS'
7 >>> meinSchrotti.farbe
8 'blau'
9 >>>
```

8.4.3. . . . so ähnlich wie Splitten

Auch Listen und Tupel kann ich in Einzelteile zerlegen oder diese zusammensetzen. Hier wird das als packen bzw. entpacken bezeichnet. Das Beispiel hier unten erklärt sich (hoffentlich!) von selbst.

Listing 8.89: Packen und entpacken

```
1 >>> text1 = 'abcdefgh'
2 >>> liste1 = list(text1)
3 >>> print(liste1)
4 ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
5
6 >>> tupel1 = tuple(liste1)
7 >>> print(tupel1)
8 ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h')
9
```

```
10 # jetzt wird die Liste entpackt:
11 >>> anfang1, *mitte1, ende1 = liste1
12 >>> print(anfang1)
13 a
14 >>> print(anfang1, mitte1, ende1)
15 a ['b', 'c', 'd', 'e', 'f', 'g'] h
16
17 # jetzt das Tupel:
18 >>> anfang1, *mitte1, ende1 = tupel1
19 >>> print(anfang1, mitte1, ende1)
20 a ['b', 'c', 'd', 'e', 'f', 'g'] h
21
22 # und jetzt der Text:
23 >>> anfang1, *mitte1, ende1 = text1
24 >>> print(anfang1, mitte1, ende1)
25 a ['b', 'c', 'd', 'e', 'f', 'g'] h
```

Eine nette Anwendung dazu: gegeben ist ein Text, aus dem der kleinste (der, der am weitesten im Alfabet vorne stehende) und der größte Buchstabe ausgegeben werden soll.

```
1 >>> text2 = 'buchstaben'
2 >>> liste2 = list(text2)
3 >>> print(liste2)
4 ['b', 'u', 'c', 'h', 's', 't', 'a', 'b', 'e', 'n']
5
6 >>> kleinster, *mittlere, groesster = sorted(liste2)
7 >>> print('Kleinster B.:', kleinster, '\tgroesster B.:', groesster)
8 Kleinster B.: a      groesster B.: u
```

8.4.4. Tupel mit Komfort

Und wenn man beides will: Immutabilität und verständlichen Zugriff? Dafür gibt es im Modul `collections` die Funktion `namedtuple`: ein Tupel, dessen Elemente Namen haben. Stellen

Listing 8.90: namedtuple

```
1 >>> import collections
2 >>> personen = []
3 >>> eineP = collections.namedtuple('person', ('name','beruf','schuhgr'))
4 >>> klaus = eineP(name='Klaus', beruf='Buchhändler', schuhgr='43')
5 >>> klaus
6 person(name='Klaus', beruf='Buchhändler', schuhgr='43')
7 >>> personen.append(klaus)
8 >>> personen
9 [person(name='Klaus', beruf='Buchhändler', schuhgr='43')]
10 >>> leni = eineP(name='Leni', beruf='Bluesgitarristin', schuhgr='40')
11 >>> personen.append(leni)
12 >>> personen
13 [person(name='Klaus', beruf='Buchhändler', schuhgr='43'),
```

```
14  person(name='Leni', beruf='Bluesgitarristin', schuhgr='40')]
15
16
17 >>> for eineP in personen:
18         if eineP.name == 'Klaus':
19             eineP.name = 'Karl'
20
21
22  Traceback (most recent call last):
23    File "<pyshell#26>", line 3, in <module>
24      eineP.name = 'Karl'
25  AttributeError: can't set attribute
```

Das ist nicht die übliche Fehlermeldung, wenn man versucht ein Element eines Tupels zu verändern; die lautet ja `TypeError: 'tuple' object does not support item assignment`

8.5. Zusammenfassung

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

Name	andere Bez.	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

Tabelle 8.2.: Überblick Strukturierte Daten

8.6. Was bei (fast) allen strukturierten Daten funktioniert

Das Wort „fast“ steht in der Überschrift, weil es Dich dazu ermutigen soll, dass Du das unten geschriebene bei verschiedenen strukturierten Daten ausprobieren sollst.

8.6.1. Die Funktion `enumerate`

Das Problem stellt sich so dar: ich habe z.B. eine Liste, die ich ausgeben möchte, aber so, dass jedem Element die laufende Nummer in der Liste vorangestellt wird. Inzwischen sollte jede und jeder das mit den bisher bekannten Bordmitteln lösen können. Die Funktion `enumerate` vereinfacht die Lösung des Problems.

Listing 8.91: Die Funktion `enumerate` bei der Arbeit

```
1 >>> teilnehmer = [['Martin', 'Schimmels'], ['Karl A.', 'Schwein'],
2                  ['Rudi', 'Ratlos'], ['Susi', 'Sorglos']]
3
4 >>> for index, (vn, nn) in enumerate(teilnehmer, start = 1):
5         print(str(index)+'. '+nn+', '+vn)
6 1. Schimmels, Martin
7 2. Schwein, Karl A.
8 3. Ratlos, Rudi
9 4. Sorglos, Susi
```

8.6.2. Die Funktion `map`

Mappe 2 Listen miteinander mit Hilfe der Funktion `v`, die nur die jeweils beiden Elemente ausgibt. Noch ziemliche Handarbeit.

Listing 8.92: Listen mappen

```
1  >>> l1 = [1,2,3,4,5]
2  >>> l2 = ['Apfel','Birne','Feige','Kirsche','Kiwi']
3  >>> def v(x,y):
4          return str(x)+' '+str(y)
5
6  >>> erg = map(v,l1,l2)
7  >>> for el in erg:
8          print(el)
9
10 1 Apfel
11 2 Birne
12 3 Feige
13 4 Kirsche
14 5 Kiwi
```

Nicht mehr so viel Handarbeit: erzeuge ein Tupel der Länge der Liste `l2`, das die Ordnungszahlen von 1 bis Länge der Liste `l2` enthält. Mappe das Tupel mit der Liste `l2`.

Listing 8.93: Listen mit Tupel mappen

```
1  >>> t1 = (i for i in range(1,len(l2)+1))
2  >>> erg = map(v,t1,l2)
3  >>> for el in erg:
4      print(el)
5
6  1 Apfel
7  2 Birne
8  3 Feige
9  4 Kirsche
10 5 Kiwi
```

8.7. Iteratoren

Hier taucht ein (sprachliches?) Problem auf.

1. Listen, Dictionaries, Mengen, ... sind iterierbar, das heißt, dass sie in einer Zählschleife abgearbeitet werden können.
2. aus einem iterierbaren Objekt kann man einen Iterator bilden; das ist ein Objekt, das u.a. die Methode `__next__` beherrscht.

Im Folgenden wird das an einer Liste und an einem Dictionary demonstriert:

Listing 8.94: Ein Liste wird zum Iterator ... und dann abgearbeitet

```
1  >>> liste1 = [10,8,1,13,5,11,4,9]
2  >>> l1 = iter(liste1)
3  >>> try:
4      while True:
```

```
5           nZahl = next(l1)
6           print(nZahl,end = ' -> ')
7   except StopIteration:
8       pass
9
10  10 -> 8 -> 1 -> 13 -> 5 -> 11 -> 4 -> 9 ->
```

Listing 8.95: Ein Dictionary wird zum Iterator ... und dann abgearbeitet

```
1  >>> dicto1 = {1:'eins',2:'zwei',3:'drei',4:'vier'}
2  >>> d1 = iter(dicto1)
3  >>> try:
4      while True:
5          nZahl = next(d1)
6          print(nZahl,end = ' -> ')
7  except StopIteration:
8      pass
9
10 1 -> 2 -> 3 -> 4 ->
```

8.8. Mengen (sets)

Mengen (englisch: sets) sind wie Dictionaries ohne Wert, also ein Dictionary mit ausschließlich Schlüsseln. Aus der Mathematik wissen wir: Mengen sind Sammlungen von wohlunterschiedenen Elementen; einfacher gesagt: die Elemente sind alle voneinander verschieden. In den beiden folgenden Listings wird jeweils das 2. Beispiel mit mehreren „3“ geschrieben. Die Umwandlung in eine Menge schmeißt die mehrfach genannten Einträge raus. Sie sehen in Python auch so aus:

```
1  >>> zahlenMenge = {1, 5, 3, 2, 8}
2  >>> print(zahlenMenge)
3  {8, 1, 2, 3, 5}
4  >>> zahlenMenge = {1, 5, 3, 2, 8, 3, 3}
5  >>> print(zahlenMenge)
6  {8, 1, 2, 3, 5}
```

Mengen können aber auch explizit durch `set` aus einer iterativen Struktur erzeugt werden:

```
1  >>> zahlenMenge = set([1, 5, 3, 2, 8])
2  >>> print(zahlenMenge)
3  {8, 1, 2, 3, 5}
4  >>> zahlenMenge = set([1, 5, 3, 2, 8, 3, 3])
5  >>> print(zahlenMenge)
6  {1, 2, 3, 5, 8}
```

Man sieht aber auch hier zwei Dinge:

1. Python speichert eine Menge nicht unbedingt in der Reihenfolge, in der sie eingegeben wurde, genau wie bei Dictionaries.

2. Die Mathematiker wissen es: in einer Menge ist ein bestimmtes Element höchstens einmal enthalten. Deswegen ist in der gespeicherten Menge nur eine „3“ .

Ganz wichtig im Umgang mit Mengen ist die leere Menge; sie enthält kein Element.

```
>>> leereMenge = set()
>>> type(leereMenge)
<class 'set'>
>>> print(leereMenge)
set()
```

8.8.1. Mengenoperationen mit Rechenzeichen

Mengen haben aber besondere Eigenschaften. Man kann den Durchschnitt und die Vereinigungsmenge von zwei Mengen bilden. Der Operator für den Durchschnitt ist das `&`, denn die Durchschnittsmenge ist die Menge aller Elemente, die in der einen und auch in der anderen Menge enthalten sind. Der Operator für die Vereinigung ist das `|` (der senkrechte Strich), denn die Vereinigungsmenge ist die Menge aller Elemente, die in der einen oder in der anderen Menge enthalten sind (oder in beiden; der Mathematiker nennt das ein nicht ausschließliches „ODER“ im Gegensatz zum ausschließlichen „ODER“ , das in der deutschen Sprache mit „ENTWEDER ... ODER“ gesprochen wird).

Listing 8.96: Durchschnitt und Vereinigung

```
>>> zahlenMenge = {1, 5, 3, 2, 8}
>>> zahlenMenge2 = {2, 5, 8, 11}
>>> zahlenMenge & zahlenMenge2
{8, 2, 5}
>>> zahlenMenge | zahlenMenge2
{1, 2, 3, 5, 8, 11}
```

Auch die Differnzmenge kann erzeugt werden, selbstverständlich durch das Minuszeichen. Aber Vorsicht: die Differenz ist nicht symmetrisch:

Listing 8.97: Differenzmenge

```
>>> zahlenMenge = {1, 5, 3, 2, 8}
>>> zahlenMenge2 = {2, 5, 8, 11}
>>> zahlenMenge - zahlenMenge2
{1, 3}
>>> zahlenMenge2 - zahlenMenge
{11}
```

In der Differenz „menge1 - menge2“ sind alle Elemente von Menge1, die nicht in Menge2 sind. Die symmetrische Differenz, also die Elemente von Menge1, die nicht in Menge2 sind zusammen mit den Elementen von Menge2, die nicht in Menge1 sind, wird durch den Operator ^ gebildet:

Listing 8.98: Symmetrische Differenz

```
1 >>> zahlenMenge2 ^ zahlenMenge
2 {1, 3, 11}
3 >>> zahlenMenge ^ zahlenMenge2
4 {1, 3, 11}
```

Teilmenge ist eine Menge, die einen Teil der Elemente der ursprünglichen Menge enthält. Der Operator ist das <=-Zeichen, für die echte Teilmenge das <-Zeichen

Listing 8.99: Teilmenge

```
1 >>> menge = {1,4,7,11,8}
2 >>> unechte = {1,8,4,7,11}
3 >>> teilMenge = {1,11,4}
4 >>> keineTeilmenge = {1,2,3}
5 >>> teilMenge <= menge
6 True
7 >>> keineTeilmenge <= menge
8 False
9 >>> unechte <= menge
10 True
```

Entsprechend werden für die Obermenge die Operatoren > bzw. >= verwendet.

8.8.2. Mengenoperationen als Methoden der Klasse Menge

Eine Menge ist ein Objekt der Klasse `set`. [5] Das bedeutet insbesondere, dass man eine Menge auch über den folgenden Befehl erzeugen kann:

Listing 8.100: Menge als Objekt der Klasse `set`

```
1 >>> ungerade = set([1,3,5,7,9])
2 >>> ungerade
3 {9, 1, 3, 5, 7}
4 >>> type(ungerade)
5 <class 'set'>
```

Das ist also tatsächlich ein Objekt der Klasse `set`.

Damit können die Mengenoperationen auch als Methoden der Klasse beschrieben werden.

Listing 8.101: Mengenoperationen als Methoden

```
1 >>> menge = set([1,4,7,8,11])
2 >>> unechte = set([1,8,4,7,11])
3 >>> teilMenge = set([1,11,4])
4 >>> keineTeilmenge = set([1,2,3])
5 >>> teilMenge.issubset(menge)
6 True
7 >>> menge.issuperset(teilMenge)
8 True
```

[5] Objekte und Klassen werden weiter hinten besprochen.

```
9  >>> keineTeilmenge.intersection(menge)
10 {1}
11 >>> keineTeilmenge.union(menge)
12 {1, 2, 3, 4, 7, 8, 11}
```

Für Mathematiker ist das klar, aber auch für Andere ist das verständlich.

Hier kommen noch einige Mengenoperationen, realisiert durch Methoden der Klasse `set`:

Listing 8.102: Mengenoperationen als Methoden (Forts.)

```
>>> a = {1,2,3,4}
>>> b = {2,4,6,8}
>>> sm = a.intersection(b)
>>> sm
{2, 4}
>>> vm = a.union(b)
>>> vm
{1, 2, 3, 4, 6, 8}
>>> dab = a.difference(b)
>>> dab
{1, 3}
>>> dba = b.difference(a)
>>> dba
{8, 6}
>>> dsym = a.symmetric_difference(b)
>>> dsym
{1, 3, 6, 8}
>>> dsym2 = b.symmetric_difference(a)
>>> dsym2
{1, 3, 6, 8}
```

8.8.3. Mengen aus anderen Daten erzeugen

So wie für Listen und Dictionaries gibt es auch für Mengen eine `set comprehension`. Das ist ganz geschickt, um aus einer Liste mehrfach auftretenden Elemente auf ein Element reduzieren. Die Syntax einer `set comprehension` ist ähnlich der der anderen comprehensions: in eine geschweifte Klammer kommt der Ausdruck, der die Menge erzeugen will. Zum Beispiel so:

Listing 8.103: set comprehension

```
>>> l2 = ['Karl', 'Paul', 'Timo', 'Karl', 'Tom', 'Paul', 'Sepp']
>>> s2 = {name for name in l2}
>>> s2
{'Paul', 'Timo', 'Tom', 'Sepp', 'Karl'}
>>> type(s2)
<class 'set'>
```

Das selbe Problem könnte man noch anders lösen:

Listing 8.104: Liste in Menge umwandeln

```
>>> l2 = ['Karl', 'Paul', 'Timo', 'Karl', 'Tom', 'Paul', 'Sepp']
>>> s3 = set(l2)
>>> s3
```

```
4 {'Paul', 'Timo', 'Tom', 'Sepp', 'Karl'}
5 >>> type(s3)
6 <class 'set'>
```

Vorteil der ersten Formatierung ist, dass der Ausdruck zur Erzeugung der Menge noch Bedingungen enthalten kann (oder sogar einen Unterausdruck).

8.9. Formatierung der Ausgabe (Fortsetzung)

Nehmen wir uns eine Liste von (rudimentären) Adressen vor, deren einzelne Adresseinträge wieder Listen mit den Feldern Vorname, Nachname, Wohnort sind.

Listing 8.105: Formatierung von Listenelementen

```
1  #!/usr/bin/python
2
3  adrListe = [ ['Karl', 'Maier', 'Stuttgart'],
4               ['Josef', 'Schulz',  'Berlin' ],
5               ['Thomas',  'Müller', 'München']
6             ]
7
8  for name in adrListe:
9       print("{0[0]} {0[1]} aus {0[2]}".format(name))
10
11 Karl Maier aus Stuttgart
12 Josef Schulz aus Berlin
13 Thomas Müller aus München
```

Die 0 bezieht sich auf die einzelne Adresse, darin werden nacheinander das jeweils 0., 1. und 2. Element ausgegeben. Wenn das jetzt in der für Adressangaben üblichen Reihenfolge Nachname, Vorname, Wohnort geschehen soll, geschieht das durch eine Vertauschung der entsprechenden „Hausnummern“ .

Listing 8.106: Formatierung von Listenelementen (Nachname zuerst)

```
1  #!/usr/bin/python
2
3  adrListe = [ ['Karl', 'Maier', 'Stuttgart'],
4               ['Josef', 'Schulz',  'Berlin' ],
5               ['Thomas',  'Müller', 'München']
6             ]
7
8  for name in adrListe:
9       print("{0[1]}, {0[0]} aus {0[2]}".format(name))
10
11 Maier, Karl aus Stuttgart
12 Schulz, Josef aus Berlin
13 Müller, Thomas aus München
```

Auch Dictionaries können mit Vorteil für die Formatierung benutzt werden.

Listing 8.107: Formatierung von Dictionaries

```
#!/usr/bin/python

adrListe = [ {'vn':'Karl', 'nn':'Maier', 'wo':'Stuttgart'},
             {'vn':'Josef', 'nn':'Schulz',  'wo':'Berlin' },
             {'vn':'Thomas',  'nn':'Müller', 'wo':'München'}
           ]

for name in adrListe:
    print("{0[vn]} {0[nn]} aus {0[wo]}".format(name))
print('------------------------------')

for name in adrListe:
    print("{0[nn]}, {0[vn]} aus {0[wo]}".format(name))
```

Auch hier bezieht sich die `0` auf das in Arbeit befindliche Objekt, die einzelnen Teile werden jetzt aber jeweils mit ihrem Schlüssel angezeigt.

Versuche einfach, irgendwelche Strukturen, die Du in den folgenden Aufgaben bearbeitest, schön formatiert auszugeben. Übung macht den Meister.

8.9.1. Typen oder Objekte einer Klasse

Listing 8.108: So findet man den Typ

```
>>> print(type('0b0101'))
<class 'str'>

>>> print(type(eval('0b0101')))
<class 'int'>

>>> print(eval('0b0101'))
5

>>> print(type('#'))
<class 'str'>

>>> print(type(3.1))
<class 'float'>

>>> print(3.1)
3.1

>>> print(eval('3.123'))
3.123

>>> print(eval('3.1.23'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    3.1.23
         ^
SyntaxError: unexpected EOF while parsing
```

8.10. Aufgaben zu Listen, Dictionaries ...

1. Schreibe ein Programm, das alle Schaltjahre von einem eingegebenen Beginn und einem eingegebenen Ende in eine Liste schreibt. Beschränke dich nicht auf eine Lösung. (list comprehension!)
2. 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.
3. 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?

4. 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.

8.11. Veränderbarkeit von Daten

Die Frage, ob Daten eines bestimmten Typs (genauer: Daten, die Objekte einer bestimmten Klasse sind) verändert werden können, ist sowohl im Kapitel über Zeichenketten als auch im Kapitel über strukturierte Daten aufgetaucht. Hier soll nochmals kurz zusammengefasst werden, was bisher bekannt ist.

Zeichenketten können nicht verändert werden. Es ist zwar möglich, ein bestimmtes Element einer Zeichenkette (also genau ein Zeichen) auszulesen, aber es kann nicht verändert werden. Schau:

```
>>> name = 'Sandra'
>>> name[5]
'a'
>>> name[5] = 'o'

Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    name[5] = 'o'
TypeError: 'str' object does not support item assignment
```

Wie gesagt: es ist nicht so einfach, aus Sandra einen Sandro zu machen; die Fehlermeldung sagt uns genau das, dass nämlich ein Element eines `str`-Objects keine Zuweisung erlaubt.

Bei einer Liste sieht das ganz anders aus.

Listing 8.109: Versuch, ein Element einer Liste zu ändern

```
>>> bListe = ['S','a','n','d','r','a']
>>> bListe[5] = 'o'
>>> bListe
['S', 'a', 'n', 'd', 'r', 'o']
```

Das klappt.

Die Unveränderlichkeit von Zeichenketten hat auch eine Bedeutung für die Speicherung von Variablen. Schauen wir, was passiert, wenn man zwei verschiedenen Variablen die selbe Zeichenkette zuweist. Nach diesen zwei Zuweisungen schauen wir uns mit Hilfe von `id` an, an welchem Speicherplatz der Text gespeichert ist.

Listing 8.110: 2 Variable mit dem selben (Text-)Inhalt

```
1 >>> a = 'Martin'
2 >>> b = 'Martin'
3 >>> id(a)
4 3055883200L
5 >>> id(b)
6 3055883200L
```

Beide Variable haben die selbe Adresse!! Um Speicherplatz zu sparen, wird der Text nur einmal gespeichert, da Python ja weiß, dass dieser unveränderlich ist. Also werden die Variable `a` und die Variable `b` ihren (gemeinsamen) Wert immer an der richtigen Stelle finden. Der Speicherplatz, an dem die Information „Martin" steht, hat zuerst den Namen `a` bekommen, später noch ein Pseudonym, einen Alias-Namen, nämlich `b`.

Wenn man die selbe Aktion mit zwei Listen durchführt, geschieht etwas völlig anderes.

Listing 8.111: 2 Listen mit dem selben Inhalt

```
1 >>> liste1 = ['M','a','r','t','i','n']
2 >>> liste2 = ['M','a','r','t','i','n']
3 >>> id(liste1)
4 3055883276L
5 >>> id(liste2)
6 3055840108L
```

Tatsächlich: zwei verschiedene Adressen. Das muss so sein, denn jede Liste könnte verändert werden, also müssen die beiden Variablen auf verschiedene Speicherplätze zugreifen.

Weiter unten im Kapitel 9.6 werden wir sehen, dass das unter Umständen Probleme bereiten kann.

8.12. Generatoren

TO DO!!!

```
1 >>> qZahlen = tuple(x**2 for x in range(1,11))
2 >>> print(qZahlen)
3 (1, 4, 9, 16, 25, 36, 49, 64, 81, 100)
4 >>> type(qZahlen)
```

Teil V.

Strukturen

9. Programmstrukturen

Don't think twice, it's alright

(Bob Dylan[1])

9.1. 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.

Anders ausgedrückt: das DRY[2]-Prinzip sollte man immer einhalten.

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!

9.1.1. Kontrollfluß

Darum haben wir uns bis jetzt noch nicht gekümmert! Bis jetzt wurde ein einzelner Befehl ausgeführt, bestenfalls ein paar Befehle hintereinander; und es gab keine Möglichkeit, von dieser Reihenfolge abzuweichen. Der Fluß des Programms war linear und von uns eindeutig vorgegeben.

Das soll sich jetzt ändern, wir werden in den Kontrollfluß eingreifen. Das geht grundsätzlich auf zwei Arten: durch Wiederholungsanweisungen und durch das Treffen von Entscheidungen. Also stürzen wir uns rein ins Vergnügen! Aber damit alles seine Ordnung hat, wird das Thema des linearen Programms gleich als erstes aufgegriffen.

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

[2] Don't Repeat Yourself

9.2. 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.

Wenn jetzt die Programmstrukturen eingeführt werden, ist es Zeit, wieder einmal den Begriff des Algorithmus hervorzuholen. Ein Algorithmus ist, einfach gesagt, nichts anderes als ein Kochrezept. Einen solchen Algorithmus beschreibt man oft in einem Mittelding zwischen natürlicher Sprache, für uns also Deutsch, und Programmiersprache. Die natürliche Sprache wird dazu leicht formalisiert, um Strukturen hervorzuheben. Das klassische Mittel dafür ist das Einrücken von zusammengehörigen Anweisungen, die von einer Bedingung abhängig sind.

Bei Sequenzen ist ein Algorithmus natürlich einfach: hier ist nichts abhängig von etwas anderem, und alles wird der Reihe nach aufgeschrieben:

Listing 9.1: Sequenz

```
1    Kerngehäuse aus Paprika entfernen
2    Paprika mit Tomaten, Zwiebeln und Knoblauch zu einem Mus mixen
3    saure Sahne schaumig rühren
4    Mus mit der Sahne vermischen
5    mit Salz und Paprikapulver abschmecken
6    ziehen lassen
```

Das ist nachzuvollziehen. Und lecker!!!

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

9.2.1. 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.
7. Aus dem Stundenlohn und der Anzahl der gearbeiteten Stunden soll der Verdienst berechnet werden.
8. In ein Programm soll der Radius eines Kreises eingegeben werden. Das Programm soll den Umfang und die Fläche des Kreises ausgeben.
9. Länge und Breite eines Rechtecks sollen eingegeben werden. Das Programm soll den Umfang, die Fläche und die Länge der Diagonalen des Rechtecks ausgeben.
10. Kosten einer Klassenfahrt:
    - Eingabe:
      - Kosten für den Bus
      - Übernachtungskosten pro Person und Nacht
      - Anzahl Nächte
      - Anzahl Schüler
      - Anzahl Lehrer
      - Gesamtkosten für Denkmäler, Museeen, Konzerte etc.
    - Ausgabe:
      - Gesamtkosten für die Fahrt
      - Kosten je Person
11. Quader:
    - Eingabe:
      - Länge des Quaders
      - Breite des Quaders
      - Höhe des Quaders
    - Ausgabe:
      - Volumen des Quaders
      - Oberfläche des Quaders
      - Benötigte Menge Draht für ein Kantenmodell des Quaders

9.3. Wenn ... dann: Die Alternative

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

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

9.3.1. Wahrheit

Hier muss ein kleines Kapitel eingeschoben werden, das sich mit Wahrheit befasst. Dazu gilt es, ein paar Fragen zu stellen und zu beantworten.

1. Welche Wahrheitswerte gibt es?
2. Welche Arten von Termen gibt es?
3. Bei welchen dieser Terme kann man sinnvoll die Frage nach einem Wahrheitswert dieses Terms stellen?
4. Wenn ich mehrere solcher Terme habe:
   a) Wie kann ich sie sinnvoll verknüpfen?
   b) Wie sieht es bei einer solchen Verknüfpung mit der Wahrheit aus?

Die erste Frage ist einfach zu beantworten: eine Aussage kann wahr oder falsch sein. Da Programmiersprachen sich an der englischen Sprache orientieren: sie kann „true“ oder „false“ sein. In Python entspricht das den Werten `True` und `False`. Diese Werte werden „Boole'sche Werte“ genannt, eine Variable, die nur diese Werte annehmen kann, heißt entsprechend „Boole'sche Variable“ [3]

Hier folgen verschiedene Arten von Termen: Ziffern, Zahlen, Buchstaben, Texte, Gleichungen, Bedingungen . . .

Zur dritten Frage: jetzt sollten wir die Arten von Termen auf einen möglichen Wahrheitswert untersuchen. Python hilft uns dabei, denn Python enthält die Funktion `bool`, die den Wahrheitswert eines Termes zurückgibt.

1. Kann eine Ziffer wahr oder falsch sein?

Listing 9.2: Wahrheit von Zahlen

```
1  >> bool(9)
2  True
3  >>> bool(5)
4  True
5  >>> bool(1)
6  True
7  >>> bool(0)
8  False
9  >>> bool(-1)
10 True
11 >>> bool(-9)
12 True
```

Fazit: Jede Ziffer ausser der Null ist wahr.

[3] nach dem englischen Mathematiker George Boole. Siehe hierzu: http://de.wikipedia.org/wiki/George_Boole

2. Kann ein Buchstabe oder ein Text wahr oder falsch sein?

Listing 9.3: Wahrheit von Texten

```
>>> bool('a')
True
>>> bool('Z')
True
>>> bool('ä')
True
>>> bool('Python')
True
>>> bool('Python ist klasse!')
True
>>> bool(' ')
True
>>> bool('')
False
```

   Fazit: Jeder Text der Länge 1 bis unendlich ist wahr. Nur der Text, der aus nichts besteht, ist falsch.

3. So, das reicht! Selbst probieren macht schlau!

Bedingungen sind die interessanteste Art von Termen. Eine Bedingung ist normalerweise der Vergleich von zwei Dingen, meistens einem konstanten Wert und dem Wert einer Variablen. Beispiele (in natürlicher Sprache), wobei die Variablen, mit denen verglichen werden soll, explizit in Klammern angesprochen werden, sind:

1. (Wert der Variablen) n größer als 17
2. (Wert der Variablen) nachname gleich „Meier“
3. „Pfeffer“ in (der Variablen, die die) Liste der Gewürze (enthält)
4. (Lesezeiger hat das) Ende der Datei (erreicht)

Hier ist klar: alle diese Bedingungen können, als Frage gestellt, mit „Ja“ oder mit „Nein“ beantwortet werden, in der Sprache der Informatik mit „True“ oder mit „False“ .

Zwischen den Werten, die in einer Bedingung miteinander verglichen werden, steht ein Vergleichsoperator. Zu den „elementaren“ Vergleichsoperatoren siehe weiter vorne im Buch.

Die ersten drei dieser Beispiele sollen jetzt in Python formuliert werden.

1. Ist n größer als 17?

```
>>> n = 18
>>> n > 17
True
>>> n = 5
False
>>> n > 17
```

2. Ist der Nachame „Meier“ ?

```
>>> nachname = 'Schimmels'
>>> nachname == 'Meier'
False
>>> nachname = 'Meier'
>>> nachname == 'Meier'
True
```

Beachte hierbei: der Vergleich auf Gleichheit geschieht durch zwei aufeinanderfolgende Gleichheitszeichen. Siehe oben bei Vergleichsoperatoren

3. Ist „Pfeffer“ ein Gewürz?

```
>>> listeGewuerze = ['Salz','Curry','Paprika']
>>> 'Pfeffer' in listeGewuerze
False
>>> listeGewuerze = ['Salz','Curry','Pfeffer','Paprika']
>>> 'Pfeffer' in listeGewuerze
True
```

Eine Bedingung kann immer den Wahrheitswert „wahr“ oder den Wahrheitswert „falsch“ haben.

Listing 9.4: True und False

```
>>> False
False
>>> True
True
>>> not False
True
>>> not True
False
```

Dieses Beispiel zeigt eigentlich nur, dass „True“ das Gegenteil von „False“ ist (und umgekehrt). Aber Vorsicht: das sind Variable, und deswegen dürfen diese Wörter keineswegs in Anführungsstrichen geschrieben werden.

Listing 9.5: True und False (ganz falsch)

```
>>> not "True"
False
>>> not "False"
False
```

Oft sind Bedingungen zusammengesetzt, und aus diesem Grunde muss man sich Gedanken darüber machen, was eine Aussage wie „WAHR und FALSCH“ oder wie „WAHR oder FALSCH“ bedeutet. Das macht man meistens in sogenannten „Wahrheitstafeln“ . Zum Glück ist das, was man in der Programmierung benötigt, überschaubar. Jetzt soll die Frage der Verknüpfung von Bool'schen Termen angegangen werden. Es gibt 3 elementare Verknüpfungen:

1. mit „und“
2. mit „oder“
3. mit „nicht“

Die reservierten Wörter in Python dazu sind **and**, **or** und **not**

Die Verknüpfungstafeln lasse ich durch ein Python-Programm schreiben (das wieder einmal vorgreift auf Abschnitt 9.5). Das sieht so aus:

Listing 9.6: Logische Verknüpfungen

```
#!/usr/bin/python

print('UND-Verknüpfung')

formatString = "{:<8} {:<8} {:<6} "
print(formatString.format('1. Wert',  '2.Wert', 'Ergebnis'))
for wert1 in (True, False):
    for wert2 in (True, False):
        print(formatString.format(str(wert1),  str(wert2), str(wert1 and wert2)))

print('\n\nODER-Verknüpfung')

formatString = "{:<8} {:<8} {:<6} "
print(formatString.format('1. Wert',  '2.Wert', 'Ergebnis'))
for wert1 in (True, False):
    for wert2 in (True, False):
        print(formatString.format(str(wert1),  str(wert2), str(wert1 or wert2)))

print('\n\nNICHT-Verknüpfung')

formatString = " {:<8} {:<6} "
print(formatString.format('1. Wert', 'Ergebnis'))
for wert1 in (True, False):
    print(formatString.format(str(wert1),  str(not wert1)))
```

Die Ausgabe des Programms sind die 3 Verknüpfungstafeln, und das sieht so aus:

1. Wert	2. Wert	Ergebnis
True	True	True
True	False	False
False	True	False
False	False	False

Tabelle 9.1.: UND-Verknüpfung

1. Wert	2.Wert	Ergebnis
True	True	True
True	False	True
False	True	True
False	False	False

Tabelle 9.2.: ODER-Verknüpfung

1. Wert	Ergebnis
True	False
False	True

Tabelle 9.3.: NICHT-Verknüpfung

Die logischen Verknüpfungen „und“ und „oder“ werden im Kurzschluß-Verfahren ausgewertet. Was das bedeutet, sieht man mit einem Zwischenblick auf die obigen Verknüpfungstafeln. Eine „und“ -Verknüpfung hat den Wert „falsch“ , wenn die erste Aussage falsch ist; eine „oder“ -Verknüpfung hat den Wert „wahr“ , wenn die erste Aussage wahr ist. In diesen beiden Fällen wird die Auswertung abgebrochen, weil das Ergebnis jetzt schon fest steht.

Aber Vorsicht: wenn man zwei Aussagen mit „oder“ verknüpft und die erste Aussage ist falsch, wird die zweite Aussage zurückgegeben. Das liefert unter Umständen nicht das gewünschte Ergebnis:

Listing 9.7: Kurzschluß beim logischen ODER

```
1  >>> 3+5 == 8  or "Heute ist der 31. Mai"
2  True
3  >>> 3+5 == 7  or "Heute ist der 31. Mai"
4  'Heute ist der 31. Mai'
```

Ich glaube, Du willst jetzt mal testen, ob Du das alles verstanden hast. Also kommt hier eine Entscheidungstabelle, in die Du in die letzte Spalte den korrekten Wahrheitswert eintragen sollst.

x	y	z	Verknüpfung(en)	Wahrheitswert
5	12		x < y	
15	12		x < y	
3	8		x > y	
3	3		x <= y	
17	18		x >= y	
3	5	7	y > x and z > y	
3	5	7	y > x and y >= z	
3	5	7	y > x or y <= z	
„sonnig“	„heiss“	„bewölkt“	x == „sonnig“ and z == „heiß“	
„sonnig“	„heiß“	„bewölkt“	x == „sonnig“ and y == „heiß“	
„sonnig“	„heiß“	„bewölkt“	x == „sonnig“ or z == „bewölkt“	
„sonnig“	„heiß“	„bewölkt“	x == „sonnig“ or not (y == „heiß“)	
„sonnig“	„heiß“	„bewölkt“	x == „sonnig“ and y == not „heiß“	
„sonnig“	„heiß“	„bewölkt“	x == „sonnig“ or not (z == „bewölkt“)	
„sonnig“	„heiß“	„bewölkt“	x == „sonnig“ or y == „heiß“	

Tabelle 9.4.: Übung zu Wahrheitswerten

Wenn Du merkst, dass das noch nicht so ganz locker klappt, denk Dir noch mehr solche Übungen aus!

9.3.1.1. Sprachliche Ungenauigkeiten und Fallen

Als erstes kommt hier ein Beispiel, das ich letzthin bei einem Programmieranfänger gesehen habe. Die Aufgabe war, ein Programm zu schreiben, das bei einer Schulnote von 5 oder 6 den Text „nicht versetzt“ ausgibt. In Pseudocode (s. Glossar) geschrieben:

Listing 9.8: Entscheidung mit einem „oder“ in Pseudocode

```
1 WENN die Note gleich 5 ODER 6 ist:
2     gib aus NICHT VERSETZT
3
```

Also kommt man in Versuchung, in Python zu schreiben:

Listing 9.9: Das selbe in Python, aber mit einem Denkfehler

```
1 note = 4
2 if note == 5 or 6:
3     print('NICHT VERSETZT')
```

Die Umgangssprache hat uns hier auflaufen lassen! Schauen wir den Term hinter dem `if` an:

1. Der Inhalt der Variablen `note` soll auf Gleichheit mittels `==` verglichen werden
2. mit dem, was hinter dem Vergleichsoperator steht,
3. also mit dem Term `5 or 6`
4. Jetzt schauen wir weiter oben noch mal nach:
   a) 5 ist eine Zahl ungleich 0, hat also den Wahrheitswert `True`
   b) 6 ist eine Zahl ungleich 0, hat also den Wahrheitswert `True`
   c) `True` mit `or` verknüpft mit `True` ist `True`
   d) Ergebnis: Wenn die Note von `0` verschieden ist, ist der Vergleich immer `True`, also wird außer bei der Note `0` immer NICHT VERSETZT ausgegeben.

Korrekt muss das Programm so aussehen:

Listing 9.10: Richtiges `oder`

```
1 note = 4
2 if note == 5 or note == 6:
3     print('NICHT VERSETZT')
```

Hier werden tatsächlich 2 Bedingungen mit einem „oder“ verknüpft.

Die umgangssprachliche Formulierung „Wenn die Note 5 oder 6 ist, ...“ muss korrekt heißen „Wenn die Note 5 ist oder die Note 6 ist, ...“

9.4. Wahrheit angewandt: die Alternative

Alternative bedeutet, dass hier in eine Sequenz eingegriffen wird und von zwei Möglichkeiten eine ausgewählt wird.[4]

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:

```
1 WENN die Person mit dem Namen Müller weibich ist:
2     begrüße sie mit Frau Müller
3 SONST:
4     begrüße sie mit Guten Tag
```

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

Die Bedingung gefolgt von Programmtext, der von der Bedingung abhängt, ist das erste Beispiel einer Kontrollstruktur. Die Syntax von Kontrollstrukturen ist immer gleich. Deshalb wird hier jetzt eine Tabelle aufgestellt, in die jetzt die erste Kontrollstruktur eingetragen wird; später folgen dann die anderen Kontrollstrukturen.

[4] 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.

Name	Schlüssel-wort	Ein-rückung	Anwei-sung(en)	Ende der Kontrollstr.
Alternative	if	keine		:
		4 Leerz.	Befehl 1	
		4 Leerz.	Befehl 2	
	elif	keine		:
		4 Leerz.	Befehl 3	
		4 Leerz.	Befehl 4	
	else	keine		:
		4 Leerz.	Befehl 5	
		4 Leerz.	Befehl 6	

Tabelle 9.5.: Kontrollstruktur Teil 1

Üblicherweise wird eine Bedingung formuliert, indem

- zwei Werte miteinander verglichen werden
- der Wert einer Variablen mit einem Wert verglichen wird
- der Wert einer Variablen mit dem Wert einer anderen verglichen wird

Es gibt dabei verschiedene Möglichkeiten, einen Vergleich zu formulieren:

- es wird auf Gleichheit untersucht
- es wird auf Ungleichheit untersucht
- es wird auf Größe mit Hilfe von „größer" , „kleiner" „größer oder gleich" , „kleiner oder gleich" untersucht

Die Zeile, die mit dem Schlüsselwort „WENN" beginnt, heißt Kopf der Alternative, das was gemacht werden soll, wenn die Bedingung erfüllt wird (oder auch nicht), ist der Körper. Die Begriffe Kopf und Körper werden auch später bei der Iteration und bei Unterprogrammen wieder auftauchen.

Eine solche Schreibweise, bei der Wörter der Alltagssprache benutzt werden, aber eine Syntax, die sich an die Logik wird „Pseudocode" (s. Glossar) genannt. Und zum Verhältnis von Pseudocode einer Programmiersprache anlehnt, zu Programmen in der Programmiersprache Python sagt Lutz [5]: „Because Python's syntax resembles *executable pseudocode*, it yields programs that are easy to understand, change, and use long after they have been written". Es gibt also wenig Gründe, in einem Skript über Python Pseudocode zu verwenden, denn Python-Code sieht wie Pseudocode aus. Wenn ich es mache, dann meistens nur, um etwas auf deutsch (und nicht auf englisch) zu schreiben.

Der Code in Python sieht dann so aus:

[5] [31],Seite 5

Listing 9.11: Entscheidung (`if - then`) (Python)

```
if geschlecht == 'w':
    print('Guten Tag, Frau Müller')
else:
    print('Guten Tag')
```

Man sieht sofort die Parallelen zwischen Pseudocode und Python-Code!

9.4.1. Blöcke und Einrückungen

Wenn mehrere Anweisungen auf der selben logischen Stufe stehen, spricht man von einem Block, oft genau von einem Anweisungsblock. Eine Sequenz, so wie sie in den bisherigen Beispielen geschrieben wurde, ist ein Block auf der höchsten logischen Ebene. Im Folgenden werden Blöcke auf einer tieferen Ebene behandelt, nämlich Blöcke, die von einer Bedingung abhängen.

Für alle strukturierenden Elemente der Sprache Python, also für alles, was in diesem Kapitel 9 behandelt wird, gilt die gleiche Syntax. Alle diese Elemente bestehen aus einem Block-Kopf und einem Block-Körper. Der Block-Kopf beginnt mit einem Schlüsselwort, das das Element beschreibt, enthält einen Namen oder eine Bedingung und endet mit einem Doppelpunkt. Der Block-Körper wird immer eingerückt, und zwar immer auf die selbe Art und mit der selben Weite. **Standard bei Python ist eine Einrückung um 4 Leerzeichen.** Der Block endet dadurch, dass wieder ausgerückt wird. Formal sieht das so aus:

Listing 9.12: Blockstruktur

```
SCHLUESSELWORT Name oder Bedingung :
    Anweisung des Blockkörpers
    weitere Anw. des Blockkörpers
    weitere Anw. des Blockkörpers
    weitere Anw. des Blockkörpers
nächste Anw. nach Beendigung des Blocks
```

Blöcke können geschachtelt werden. Die Regeln des vorigen Absatzes gelten aber weiter, also sieht eine formale Beschreibung eines geschachtelten Blockes so aus:

Listing 9.13: geschachtelte Blöcke

```
SCHLUESSELWORT Name oder Bedingung des äußeren Blocks :
    Anweisung des äußeren Blockkörpers
    weitere Anw. des äußeren Blockkörpers
    SCHLUESSELWORT Name oder Bedingung des inneren Blocks :
        Anweisung des inneren Blockkörpers
        weitere Anw. des inneren Blockkörpers
    weitere Anw. des äußeren Blockkörpers
    weitere Anw. des äußeren Blockkörpers
nächste Anw. nach Beendigung des äußeren Blocks
```

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 weiblich 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“

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):

Listing 9.14: Verschachtelte Entscheidungen (Pseudocode)

```
1  WENN die Person weiblich ist:
2      WENN es vor 12 Uhr ist:
3          begrüße sie mit 'Guten Morgen, Frau Müller'
4      SONST:
5          begrüße sie mit 'Guten Tag, Frau Müller'
6  SONST:
7      WENN es vor 12 Uhr ist:
8          begrüße sie mit 'Guten Morgen'
9      SONST:
10         begrüße sie mit 'Guten Tag'
```

In Python sieht das so aus:

Listing 9.15: Verschachtelte Entscheidungen (Python-Code

```
1  if geschlecht == 'w':
2      if std < 12:
3          print('Guten Morgen, Frau Müller')
4      else:
5          print('Guten Tag, Frau Müller')
6  else:
7      if std < 12:
8          print('Guten Morgen')
9      else:
10         print('Guten Tag')
```

Und auch hier sollte man noch mal einen genießerischen Blick auf die Ähnlichkeit des Pseudocodes und des Python-Codes werfen!

Auch wenn es einigen Lesern überflüssig erscheint, muss hier kurz auf den Begriff „Bedingung“ eingegangen werden. Eine Bedingung ist eine Aussageform, die entweder den Wert „WAHR“ oder den Wert „FALSCH“ annehmen kann. In Python wird durch `True` der Wert „WAHR“ , durch `False` der Wert „FALSCH“ angegeben. Beachte hierbei die Groß-/Kleinschreibung der Wahrheitswerte in Python!

9.4.2. Beispiele zu Bedingungen

Trotzdem kommen hier noch einige Code-Beispiele zu Bedingungen. Dabei formuliere ich immer die Bedingung, gebe in Abhängigkeit von dem Wahrheitswert der Bedingung einen beschreibenden Satz aus und zusätzlich noch den Wahrheitswert.

Listing 9.16: Beispiele für Vergleiche auf Gleichheit

```
>>> z1 = 3
>>> z2 = 5
>>> z3 = 7 - 4
>>> if z1 == z2:
...     print(z1, ' = ', z2)
...     print(z1 == z2)
... else:
...     print(z1, ' != ', z2)
...     print(z1 != z2)
...
3  !=  5
True

>>> if z1 == z3:
...     print(z1, ' = ', z3)
...     print(z1 == z3)
... else:
...     print(z1, ' != ', z3)
...     print(z1 != z3)
...
3  =  3
True

>>> t1 = 'gut'
>>> t2 = 'schoen'
>>> if t1 == t2:
...     print(t1, ' = ', t2)
...     print(t1 == t2)
... else:
...     print(t1, ' != ', t2)
...     print(t1 != t2)
...
gut  !=  schoen
False
```

Das ist vermutlich leicht nachzuvollziehen. Also kommen hier entsprechende Vergleiche auf „größer“ und „kleiner “

Listing 9.17: Beispiele für Vergleiche auf größer und kleine

```
>>> z1 = 3
>>> z2 = 5
>>> if z1 > z2:
...     print(z1, ' größer als ', z2)
...     print(z1 == z2)
... else:
...     print(z1, ' nicht größer als ', z2)
...     print(z1 > z3)
...
3  nicht größer als  3
False
>>> if z1 < z2:
...     print(z1, ' < ', z2)
...     z1 < z2
... else:
...     print(z1, ' nicht kleiner als ', z2)
...     print(z1 < z2)
...
3  <  5
True
```

Nachdem man das gelesen und verstanden hat, sollte man auf der Python-Shell oder in Idle ähnliche Dinge probieren.

Nur eine kleine Selbstverständlichkeit sollte hier am Rande erwähnt werden: das Gegenteil von „größer“ ist NICHT „kleiner“ !! Das Gegenteil von „größer“ ist „nicht größer“ , oder anders gesagt: das Gegenteil von „größer“ ist „kleiner oder gleich“ !!

9.4.2.1. Kurzschlüsse

Kurzschlüsse in der Elektrik sind schlecht. In der Programmierung sind Kurzschlüsse oft erlaubt, manchmal sinnvoll, sie machen Programmcode kürzer ... aber sie machen Programmcode nicht immer verständlicher. Auch bei der Alternative gibt es die Möglichkeit des Kurzschlusses. Dazu kommt hier zuerst das ausführliche Beispiel. Inzwischen verstehst Du dieses Beispiel ohne weitere Kommentare:

Listing 9.18: Vergleich und Bewertung ausführlich

```
1  >>> schuhgroesse = 48
2  >>> if schuhgroesse > 45:
3  ...     print('Das sind keine Schuhe, sondern kleine Boote')
4  ... else:
5  ...     print('Das sind keine Boote!!')
6  ...
7  Das sind keine Schuhe, sondern kleine Boote
8  >>> schuhgroesse = 43
9  >>> if schuhgroesse > 45:
10 ...     print('Das sind keine Schuhe, sondern kleine Boote')
11 ... else:
12 ...     print('Das sind keine Boote!!')
13 ...
14 Das sind keine Boote!!
```

Verständlich, ordentlich, aber ein bißchen viel Schreibarbeit. Kurzschlüsse machen es kürzer!

Listing 9.19: Vergleich und Bewertung mit Kurzschluss

```
1  >>> sg = 48
2  >>> print('Das sind keine ',(sg > 45 and 'Schuhe, sondern Boote')
3          or ' Boote!!')
4  Das sind keine Boote!!
5  >>> sg = 45
6  >>> print('Das sind keine ',(sg > 45 and 'Schuhe, sondern Boote')
7          or ' Boote!!')
8  Das sind keine  Schuhe, sondern Boote
```

Die beiden Texte 'Schuhe, sondern kleine Boote' und 'Boote!!' haben jeweils den boole'schen Wert True. Der Ausdruck in der Klammer ist also dann wahr, wenn sg größer als 45 ist, und dann wird der erste Text (der vor dem or) ausgegeben, falls sg nicht größer als 45 ist, ist der Ausdruck in der Klammer falsch und der zweite Text (der nach dem or) wird ausgegeben.

9.4.3. 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.

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

Listing 9.20: 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? ')
```

Wenn man eine solche Mehrfachentscheidung treffen muss, gibt es eine Faustregel: arbeite entweder von „groß“ nach „klein“ oder von „klein“ nach „groß“ , aber vermische nie die Vorgehensweisen. Das führt zu logischen Fehlern, die man im Moment des Schreibens (vor allem, wenn man noch nicht so viel Erfahrung mit dem Schreiben solcher Anweisungen hat) aber gar nicht merkt. Als Gegenbeispiel mische ich das obige Beispiel nur ein wenig durch:

Listing 9.21: Mehrere Möglichkeiten, die schief gehen

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

Gehen wir das also mal in Handarbeit durch, für den Fall, dass die Variable `stunde` den Wert 19 hat:

1. beim ersten `if` stellt Python fest, dass der Wert größer als 10 ist, also wird nicht in das erste print (den „Guten Morgen“) verzweigt, es wird also nichts ausgegeben

2. beim zweiten `if` stellt Python fest, dass der Wert größer als 18 ist, also wird nicht in das zweite print (den „Guten Tag“) verzweigt, es wird also nichts ausgegeben
3. beim dritten `if` stellt Python fest, dass der Wert kleiner als 24 ist, also wird in das dritte print (die „Gute Nacht“) verzweigt, es wird also „Gute Nacht“ ausgegeben, obwohl hier der „Guten Abend“ angebracht wäre

   Außer bei der Programmierung gibt das vor allen Dingen bei unseren Kindern richtig Ärger: „Es ist doch noch gar nicht so spät, und ich will doch noch den Tatort im Fernsehen schauen und überhaupt ist das unfair!“
4. das vierte `if` wird nämlich gar nicht mehr überprüft, weil das Programm ja schon eine korrekte Entscheidung getroffen hat. Dann hört es auf zu vergleichen!

Du siehst: durch ein Nichteinhalten einer natürlichen Reihenfolge wird hier eine Bedingung bejaht, die eigentlich verneint werden müsste.

9.4.4. So sehen Mehrfachentscheidungen schöner aus!

Das oben stehende Beispiel sieht aber nicht schön aus[6].

Die in zwei aufeinanderfolgenden Zeilen stehenden

```
1 else:
2     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:

Listing 9.22: Verschachtelte Entscheidungen mit `elif`

```
1  if stunde < 10:
2      print('Guten Morgen')
3  elif stunde < 18:
4      print('Guten Tag')
5  elif stunde < 21:
6      print('Guten Abend')
7  elif stunde <24:
8      print()'Gute Nacht ')
9  else:
10     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

[6] 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.

ausdrücklich aufgelisteten Fälle behandelt werden. Meistens ist der Programmierer nämlich froh, wenn das Programm mit den Daten, die er sich vorstellt, korrekt funktioniert, und er denkt nicht an alle „unmöglichen“ Möglichkeiten, auf die ein Anwender kommt.

9.4.5. 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!

Listing 9.23: Verschachtelte Entscheidungen getrickst mit Dictionary

```
1  preise = {'Milch':1.20, 'Apfelsaft':1.50, 'Orangensaft':1.60, 'Cola':2.20}
2
3  getraenk = input('Bitte Getränk eingeben (Milch/Apfelsaft/
4                   Orangensaft/Cola)'
5  print(preise[getraenk])
```

ist gleichwertig zu

Listing 9.24: Verschachtelte Entscheidungen zu Fuß

```
1  getraenk = input('Bitte Getränk eingeben (Milch/Apfelsaft/
2                   Orangensaft/Cola)'
3  if getraenk == 'Milch':
4      print(1.20)
5  elif getraenk == 'Apfelsaft':
6      print(1.50)
7  elif getraenk == 'Orangensaft':
8      print(1.60)
9  elif getraenk == 'Cola':
10     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:

Listing 9.25: Verschachtelte Entscheidungen zu Fuß mit Fehlerbehandlung

```
1  getraenk = input('Bitte Getränk eingeben (Milch/Apfelsaft/
2                                           Orangensaft/Cola)'
3  if getraenk == 'Milch':
4      print(1.20)
5  elif getraenk == 'Apfelsaft':
6      print(1.50)
7  elif getraenk == 'Orangensaft':
8      print(1.60)
9  elif getraenk == 'Cola':
10     print(2.20)
11 else:
12     print('fehlerhafte Eingabe')
```

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

Listing 9.26: das selbe mit Dictionary

```
1 preise = {'Milch':1.20, 'Apfelsaft':1.50, 'Orangensaft':1.60, 'Cola':2.20}
2
3 getraenk = input('Bitte Getränk eingeben (Milch/Apfelsaft/
4                    Orangensaft/Cola)'
5 print(preise.get(getraenk, 'falsche Eingabe'))
```

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

9.4.6. Aufgaben zu Auswahl

1. In ein Programm soll der Umsatz eines Mitarbeiters eingegeben werden. Wenn der Umsatz über 30 000 € liegt, wird dem Mitarbeiter eine Prämie von 1 Prozent des Umsatzes gewährt. Diese Prämie soll ausgegeben werden.
2. Der Eintrittspreis für die Minigolf-Anlage beträgt
   - 2,50 € für Kinder und Jugendliche unter 18 Jahren
   - 4,00 € für Erwachsene

   Das Alter eines Spielers soll eingegeben werden, und der zu zahlende Preis soll ausgegeben werden.
3. Die vorige Aufgabe wird schwerer: Der Eintrittspreis für die Minigolf-Anlage beträgt
   - 2,50 € für Kinder und Jugendliche unter 18 Jahren
   - 4,00 € für Erwachsene
   - 3,00 € für Senioren über 65 Jahre

   Das Alter eines Spielers soll eingegeben werden, und der zu zahlende Preis soll ausgegeben werden.
4. **Punktprobe:** In ein 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.
5. **Zwei-Punkte-Form der Geradengleichung:** In ein 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.
6. 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!
7. 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.
8. **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. (Hallo!! Dazu muss man die Quadratwurzel aus einem Term berechnen. Das kann ich noch gar nicht. Aber … vielleicht erinnerst Du Dich, in welchem Zusammenhang Quadatwurzeln damals in der Schule behandelt wurden, und wie man Quadratwurzeln mit den bisher bekannten Fertigkeiten berechnen kann.)
9. 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.

10. 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.
11. Für die Berechnung des Wochentages hat Chr. Zeller (in Erweiterung einer Formel von 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): $w = (d + \lfloor 2,6 \cdot m - 0,2 \rfloor + y + \lfloor \frac{y}{4} \rfloor + \lfloor \frac{c}{4} \rfloor - 2c) mod7$

    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

0	1	2	3	4	5	6
So.	Mo.	Di.	Mi.	Do.	Fr.	Sa.

9.4.7. Verknüpfte Logik-Operatoren und Reihenfolge

Fast selbstverständlich ist, dass verknüpfte logische Operationen von links nach rechts abgearbeitet werden. Wenn ein Ausdruck wie **x > 3 or x < -2** auftaucht, wird zuerst geprüft ob **x > 3** ist.

Jetzt aber!!! Jetzt verhält sich Python ausgesprochen vernünftig. Weiter oben bei der oder-Verknüpfung haben wir gesehen, dass eine „ver-oderte“ Verknüpfung wahr ist, wenn ein Teil wahr ist. Wenn x also den Wert 7 hat, dann ist der erste Teil, nämlich **x > 3** wahr. Also ist es völlig egal, ob der zweite Teil, nämlich **x < -2** wahr oder falsch ist, der gesamte Ausdruck ist wahr.

Dann braucht man also nicht weiterzulesen, was mit dem zweiten Teil passiert, denn der gesamte Ausdruck wird wahr sein. Also liest Python nicht weiter.

Das ist gut so, denn das Programm spart dadurch Zeit. Es läuft schneller! Na gut, bei einem einzelnen Ausdruck ist die Zeitersprarnis nicht der Rede wert, aber wenn ein Programm Hunderte solcher Vergleiche enthält, dann merkt man das vielleicht doch. Viel wichtiger ist, dass man damit aber eine elegante Methode hat, mögliche Fehler abzufangen.

Nehmen wir uns als Anwendung dafür die Mitternachtsformel vor. Dazu benötigt man die (Quadrat-)Wurzel-Funktion, auf englisch die **square root**, als Funktion **sqrt**. Die steckt im Modul für Mathematik, der import werden muss mit **import math**. Das Programm könnte in seiner schlichtesten Form so aussehen:

Listing 9.27: Mitternachtsformel

```
import math

print("\t\tMitternachtsformel\n\t(Lösung von quadratischen Gleichungen)")

a = float(input("Koeffizient a des quadratischen Gliedes eingeben: \t"))
b = float(input("Koeffizient b des linearen Gliedes eingeben: \t"))
c = float(input("Koeffizient c des absoluten Gliedes eingeben: \t"))

diskr = b*b - 4*a*c
print('Die quadratische Gleichung lautet', a, 'x^2 +', b, 'x +',c, '= 0')
print('Die Diskriminante der quadratischen Gleichung: D =', diskr)
if diskr > 0:
    print("\t2 Lösungen")
    print("x1 = ",)
    print( (-b + math.sqrt(diskr))/(2*a) )
    print("x2 = ",)
    print( (-b - math.sqrt(diskr))/(2*a) )
elif diskr == 0:
    print("1 Lösung")
    print("x1/2 = ",)
    print( (-b)/(2*a) )
else:
    print("keine Lösung")
```

Das scheint zu funktionieren:

Listing 9.28: Test 1 zu Mitternachtsformel

```
        Mitternachtsformel
    (Lösung von quadratischen Gleichungen)
Koeffizient a des quadratischen Gliedes eingeben:    1
Koeffizient b des linearen Gliedes eingeben:    -5
Koeffizient c des absoluten Gliedes eingeben:    6
Die quadratische Gleichung lautet 1 x^2 + -5 x + 6 = 0
Die Diskriminante der quadratischen Gleichung: D = 1
    2 Lösungen
x1 =  3.0
x2 =  2.0
```

Noch ein Test:

Listing 9.29: Test 2 zu Mitternachtsformel

```
        Mitternachtsformel
    (Lösung von quadratischen Gleichungen)
Koeffizient a des quadratischen Gliedes eingeben:   1
Koeffizient b des linearen Gliedes eingeben:   1
Koeffizient c des absoluten Gliedes eingeben:   1
Die quadratische Gleichung lautet 1 x^2 + 1 x + 1 = 0
Die Diskriminante der quadratischen Gleichung: D = -3
keine Lösung
```

Es scheint alles in Ordnung zu sein!

Listing 9.30: Test 3 zu Mitternachtsformel

```
        Mitternachtsformel
    (Lösung von quadratischen Gleichungen)
Koeffizient a des quadratischen Gliedes eingeben:   0
Koeffizient b des linearen Gliedes eingeben:   2
Koeffizient c des absoluten Gliedes eingeben:   5
Die quadratische Gleichung lautet 0 x^2 + 2 x + 5 = 0
Die Diskriminante der quadratischen Gleichung: D = 4
    2 Lösungen
x1 =
Traceback (most recent call last):
  File "./mnf.py", line 18, in <module>
    print( (-b + math.sqrt(diskr))/(2*a) )
ZeroDivisionError: float division by zero
```

Da wollte jemand also den absoluten Härtetest machen, und ausprobieren, ob unser tolles Programm auch die Nullstelle einer linearen Funktion berechnen kann, also hat er einfach für den Koeffizienten `a` eine 0 eingegeben. Wenn man sich den Quelltext weiter oben nochmal anschaut (oder noch besser: wenn man weiß, dass in der Mitternachtsformel durch `2 * a` geteilt werden muss), dann ist klar, dass ein Fehler auftaucht.

Es ist aber relativ einfach, diesen Fehler abzufangen, d.h. zu verhindern, dass eine fehlerhafte Eingabe wie oben das Programm abbrechen lässt, wenn man das oben gesagte über vernüpfte Logik-Operatoren nochmals liest und sich dann erinnert, dass ein mit „und“ verknüpfter logischer Ausdruck nur dann wahr ist, wenn beide Teile wahr sind. Also fragt man einfach an den kritischen Stellen nicht nur den Wert der Diskriminante ab, sondern zuerst den Wert des Parameters `a`. Ich nenne das Vorgehen „Aufpasser“ , weil diese zusätzliche Abfrage aufpasst, dass die kritische Variable einen sinnvollen Wert enthält. Also sieht das so aus:

Listing 9.31: Mitternachtsformel mit „Aufpasser“

```
import math

print("\t\tMitternachtsformel\n\t(Lösung von quadratischen Gleichungen)")

a = float(input("Koeffizient a des quadratischen Gliedes eingeben: \t"))
b = float(input("Koeffizient b des linearen Gliedes eingeben: \t"))
c = float(input("Koeffizient c des absoluten Gliedes eingeben: \t"))

diskr = b*b - 4*a*c
print('Die quadratische Gleichung lautet', a, 'x^2 +', b, 'x +',c, '= 0')
print('Die Diskriminante der quadratischen Gleichung: D =', diskr)
if a != 0 and diskr > 0:
    print("\t2 Lösungen")
    print("x1 = ",)
    print( (-b + math.sqrt(diskr))/(2*a) )
    print("x2 = ",)
    print( (-b - math.sqrt(diskr))/(2*a) )
elif a != 0 and diskr == 0:
    print("1 Lösung")
    print("x1/2 = ",)
    print( (-b)/(2*a) )
else:
    print("keine Lösung")
```

In den beiden Zeile mit den Bedingungen, also den Zeilen die mit `if` bzw. mit `elif` beginnen, steht jetzt also vorne die Abfrage, ob `a` ungleich 0 ist. Und jetzt passiert nichts schlimmes mehr:

Listing 9.32: Test 3 zu Mitternachtsformel

```
>>>            Mitternachtsformel
    (Lösung von quadratischen Gleichungen)
Koeffizient a des quadratischen Gliedes eingeben:    0
Koeffizient b des linearen Gliedes eingeben:    1
Koeffizient c des absoluten Gliedes eingeben:    4
Die quadratische Gleichung lautet 0 x^2 + 1 x + 4 = 0
Die Diskriminante der quadratischen Gleichung: D = 1
keine Lösung
```

Klar? Wenn `a` den Wert 0 hat, ist der erste Teil des mit „und“ verknüpften Ausdrucks falsch, damit ist der ganze Ausdruck falsch, und damit wird dieser Ausdruck nicht weiter untersucht. Also wird auch gar nicht versucht, durch 0 zu dividieren, und alles ist in bester Ordnung. Zugegeben, das ist noch nicht die absolut anwenderfreundliche Kommunikation des Programms, aber schon ein ganzes Stück besser, als mit einer Fehlermeldung abzubrechen.

9.5. Schleifen

God knows when
But you're doin' it again

(Bob Dylan[7])

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

```
1   programmieranweisung_fuer_den_wert_1
2   programmieranweisung_fuer_den_wert_2
3   programmieranweisung_fuer_den_wert_3
```

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! Ganz allgemein kann man das Problem auch so umschreiben: hier ist eine Menge von Dingen; mach für jedes von ihnen das selbe (oder etwas sehr ähnliches).

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.

Diese beiden Arten sollen noch an einem Beispiel verdeutlicht werden: es sollen alle Zahlen von 1 bis 100 ausgedruckt werden. Hier folgt die Realisierung, so wie wir sie mit den bisherigen Mitteln erreichen könnten:

Listing 9.33: Die (umgangssprachlich) ersten 100 natürlichen Zahlen ausgeben

```
1 print(1)
2 print(2)
3 print(3)
4 ...
5 print(99)
6 print(100)
```

(Im Titel des Beispiels steht „umgangssprachlich“ , weil für den Mathematiker die ersten 100 natürlichen die Zahlen von 0 bis 99 sind.) Das funktioniert. Aber das ist nicht schön, kein Mensch wollte so etwas schreiben, es ist fehleranfällig, kurz: so nicht! Die erste oben beschriebene Art, dieses Problem zu lösen, nämlich in einer Zählschleife, sieht in Pseudocode so aus:

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

Listing 9.34: Zählschleife im Pseudocode

```
für jede Zahl i aus der Menge (1 ... 100)
    drucke i
```

Nicht schlecht: statt 100 Zeilen Programmcode sind das 2 Zeilen Pseudocode. Und wir werden sehen: daraus werden nach der Übersetzung einiger Schlüsselwörter auch genau 2 Zeilen Python-Code.

Listing 9.35: Die selbe Zählschleife in Python

```
for i in range(1,101):
   print(i)
```

Die zweite oben beschriebene Art, dieses Problem zu lösen, nämlich in einer Schleife mit Abbruchbedingung, sieht in Pseudocode so aus:

Listing 9.36: Schleife mit Abbruchbedingung im Pseudocode

```
i = 1
solange zahl <= 100
    drucke i
    i = i + 1
```

Auch nicht schlecht: die 100 Zeilen Programmcode von oben sind zu 4 Zeilen Pseudocode geschrumpft. Das wird auch wieder zu 4 Zeilen Python-Code, und zwar genauso einfach wie bei der Zählschleife.

Allerdings kann man hier schon eine Faustregel festhalten, wenn man die Faust aufmacht und die Finger nachzählt: 4 ist doppelt so viel wie 2! Die Faustregel lautet: wenn Du etwas in einer Zählschleife oder in einer Schleife mit Abbruchbedingung erledigen kannst, dann nimm die Zählschleife. Die ist kürzer und besser verständlich!

9.5.1. Zähl-Schleifen

> 5,4,3,2,1
> Cassius Clay you'd better run
> 99, 100, 101, 102
> Your mother won't even
> recognize you
>
> *(Bob Dylan[8])*

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,

[8] I shall be free No. 10 *auf:* Another Side of

1. was der Anfangswert der Zählung sein soll,
2. was der Endwert sein soll und
3. mit welcher Schrittweite gezählt werden soll.

Eine Möglichkeit, über eine Menge von Zahlen zu schleifen, ist mittels des `range`-Objekts.[9] Ein `range`-Objekt ist eine Konstruktionsvorschrift, die beim Aufruf jeweils das nächste Element erzeugt. Das `range`-Objekt wird erzeugt, indem ihm

1. entweder der Endwert als Parameter
2. oder der Anfangswert und der Endwert
3. oder der Anfangswert, der Endwert und die Schrittweite

übergeben werden.

Das englische Wort „range“ bedeutet Bereich. Wenn nur ein Wert angegeben ist, wird als Standard-Startwert die „0“ angenommen. Das freut den Mathematiker, denn die natürlichen Zahlen fangen bei „0“ an. Ebenso wird angenommen, dass der dritte Wert, die Schrittweite, wenn er nicht angegeben ist, „1“ ist. Außerdem sieht man im unten stehenden Beispiel, dass der `bereichx` (mit $x \in \{1; 2; 3\}$) ein Objekt ist, das eine Konstruktionsbeschreibung für eine Liste enthält. Mit `print` wird diese Konstruktionsbeschreibung ausgeben, mit `list` wird die Liste erzeugt.

Listing 9.37: `range` (Bereich)

```
1  >>> bereich1 = range(16)
2  >>> print(bereich1)
3  range(0, 16)
4  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
5  >>> list(bereich1)
6  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
7  >>> bereich2 = range(3,12)
8  >>> list(bereich2)
9  [3, 4, 5, 6, 7, 8, 9, 10, 11]
10 >>> bereich3 = range(1,25,2)
11 >>> print(bereich3)
12 range(1, 25, 2)
13 >>> list(bereich3)
14 [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
```

Der einfachste Fall in Python ist

Listing 9.38: Zählschleife

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

[9] In Python 2.x war `range` eine Funktion und erzeugte eine Liste. Das war ungünstig, wenn `range` einen großen Bereich umfasste: die Liste wurde sehr lang und belegte folglich viel Speicherplatz.

In der folgenden Tabelle wird gegenübergestellt, wie sich ein einfacher Algorithmus, der in der linken Spalte in „Pseudocode“ beschrieben ist, fast eins zu eins in Python übertragen lässt. Das ist das Beispiel vom Anfang des Kapitels. Auch hier gilt wieder: wer das Problem richtig durchdenkt und klar formuliert, der schreibt das dazugehörige Python-Programm sehr schnell.

Pseudocode	Python-Code

für jede Zahl i aus der Menge (1 ... 100)
    drucke i
	for i in range(1,101):
	    print(i)

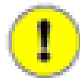
Tabelle 9.6.: Pseudocode und Python-Code

Hier finden wir wieder die Einrückung, die wir aus dem Kapitel über die Alternative kennen. Der Aufruf der print-Funktion 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.

Listing 9.39: 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.

Listing 9.40: 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

Listing 9.41: 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:

Listing 9.42: 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:

Listing 9.43: Schleifen über Liste

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

Mit den beiden Sprachelementen aus den letzten beiden Absätzen kann man sich nochmal veranschaulichen, wie die Hausnummern mit den Elementen zusammenhängen. Im folgenden Code-Segment wird die Hausnummer und das Element jeweils ausgegeben:

Listing 9.44: Länge einer Liste

```
>>> zliste = [1,2,3,4,5,6]
>>> for index in range(len(zliste)):
    print(index, zliste[index])


0 1
1 2
2 3
3 4
4 5
5 6
```

Alles klar? Die Länge der Liste ist 6, die `range`, geht also genau von 0 bis 5, und das sind die Hausnummern der Elemente der Liste. Jetzt hat wahrscheinlich jeder verstanden, warum es

1. sinnvoll ist, von 0 ab zu zählen
2. dass das Endelement einer `range` nicht mehr mit bearbeitet wird

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

Listing 9.45: 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 der den print-Funktion als zusätzlichen Parameter `end=' '` mitgibt.

Listing 9.46: Schleifen über Zeichenketten

```
>>> name = "Martin"
>>> for buchstabe in name:
            print(buchstabe, end='')
 M a r t i n
```

Und hier noch ein Beispiel von der Ostalb:

Listing 9.47: Schleife mit 2 Variablen

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

  Abele
  Bbele
  Cbele
  Dbele
```

Als letztes Beispiel soll hier, wieder in einer Gegenüberstellung von Pseudocode und Python-Code, der (Noten-)Durchschnitt einer Klasse berechnet werden. Der Lehrer weiß zwar, dass in der Klasse 25 Schüler sind, aber aus allgemein bekannten Gründen sind bei einer Klassenarbeit nicht zwingend alle Schüler anwesend. Trotzdem ist das kein Problem. Seien also ganz einfach alle Noten dieser Klassenarbeit in einer Liste gespeichert `notenL = [2.5, 3.6, 1.7, 4.2, 5.5, 5.2, 1.6]` (ok, da fehlen aber heute viele!!! Egal, ich bin zu faul, da mehr reinzuschreiben, und das Programm soll auch funktionieren, gleich wieviele Elemente in der Liste sind).

Pseudocode	Python-Code
`setze Anzahl der Schueler 0`	`anzSchueler = 0`
`setze Summe der Noten 0`	`notenSumme = 0`
`Noten sind in Listen-Variable notenL`	`notenL = [2.5,3.6,1.7,4.2,5.5,5.2,1.6]`
`für jede Note aus der Notenliste`	`for note in notenL:`
`   zähle Anzahl Schüler hoch`	`   anzSchueler += 1`
`   addiere Note zur Notensumme`	`   notenSumme += note`
`Durchschn. = Noten-Summe / Anz. Schüler`	`durchschn = notenSumme / anzSchueler`
`drucke Durchschnitt aus`	`print('Durchschnitt = ', durchschn)`

Tabelle 9.7.: Notendurchschnitt: Pseudocode und Python-Code

Das Programm ist gut! Mit einer überschaubaren (sogar endlichen) Anzahl von Programmzeilen kann ich eine nicht mehr so gut überschaubare Anzahl von Daten bearbeiten. Umgangssprachlich würde man wohl behaupten, dass die Notenliste unendlich viele Noten enthalten könnte, als Mathematiker muss man natürlich genau sein und sagen, dass die Notenliste beliebig lang sein kann. Ich habe sogar mehr erreicht, als ich ursprünglich vorhatte. Die erste Idee war, dass in der Notenliste immer die Noten aller Schüler stehen, dass die Länge der Liste also fest ist. Python ist aber in der Lage, mit einfachen Mitteln eine Liste beliebiger Länge abzuarbeiten.

Weil das Mitzählen, oben durch den Zähler `anzSchueler` erledigt, einfach ist, aber immer wieder benötigt wird, ist so etwas natürlich in Python integriert. Man könnte das Programm vereinfachen, indem man die Methode `len(liste)` aufruft. Diese gibt die

Anzahl der Elemente der Liste, hier also die Anzahl der Schüler, die die Klassenarbeit mitgeschrieben haben, zurück.

Es fehlt aus dem Kapitel über strukturierte Daten 8.2.6 noch das Beispiel, wie man eine Matrix schön darstellt.

Listing 9.48: 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.

Auch Dictionaries können in einer Zähl-Schleife abgearbeitet werden, allerdings ist hierbei nicht klar, in welcher Reihenfolge. Das kann aber manchmal ganz geschickt sein, zum Beispiel, wenn man ein Ratespiel programmiert. Dieses Spiel hier hilft vielleicht beim Vokabellernen. Es gibt den englischen Begriff aus und fragt die deutsche Übersetzung ab:

Listing 9.49: Ein kleines Quiz

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
englDeutsch = {'dog':'Hund', 'cat':'Katze', 'cow':'Kuh', 'sheep':'Schaf'}
punkte = 0
for tier in englDeutsch:
  eingAufforderung = "Wie heisst ", tier, " auf deutsch?"
  dTier = input(eingAufforderung)
  if englDeutsch[tier] == dTier:
    punkte = punkte + 1
    print("Gut!!")
  else:
    print("so nicht")

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

Das Spiel könnte noch dadurch erweitert werden, dass es hinzulernt, etwa dadurch, dass ein Begriff, der noch nicht im Dictionary enthalten ist, weil eine falsche Antwort gegeben wurde, eingefügt wird. Eine nette Übungsaufgabe!

9.5.1.1. Mitzählen in einer Zählschleife

Manchmal ist es wünschenswert, dass Python sein Wissen, bei dem wievielten Element innerhalb einer Zählschleife es sich gerade befindet, nicht für sich behält, sondern ausgibt. Dazu dient die Funktion `enumerate`. Als Beispiel wird hier eine Liste von Namen genommen, bei der durch eine Zählschleife nicht nur der Name, sondern auch die Position des Namens in der Liste ausgegeben werden soll.

Listing 9.50: Zählschleife mit Zähler

```
#!/usr/bin/python

namensListe = ['Eva', 'Klaus','Bernd', 'Mia', 'Anna', 'Paulo', 'Pia']
for zaehler, name in enumerate(namensListe):
    print('%d. Person: %s' % (zaehler+1, name))
>>> 1. Person: Eva
2. Person: Klaus
3. Person: Bernd
4. Person: Mia
5. Person: Anna
6. Person: Paulo
7. Person: Pia
```

Ein ähnliches Problem ist, dass ich in einer Schleife zwei Sachen machen will: rechnen und zählen. Ein Beispiel: ich habe eine Liste mit Zahlen und ich will zum einen wissen, wie lang die Liste ist, und zum anderen, wie groß die Summe der Zahlen in der Liste ist.

Listing 9.51: Zählen und rechnen

```
>>> liste = [1,2,3,4]
>>> anz = 0
>>> sum = 0
>>> for i in liste:
...     (anz,sum) = (anz+1, sum+i)
...
>>> print(anz, sum)
4 10
```

Hier benutze ich, dass Python in einer Anweisung mehrere Wertzuweisungen machen kann.

9.5.2. While-Schleifen

Oft soll aber nicht „gezählt geschleift“ werden, sondern man soll einen Block von Anweisungen solange wiederholen, bis eine bestimmte Bedingung erreicht ist. Um wieder den Vergleich eines Algorithmus mit einem Kochrezept hervorzuholen: da gibt es so etwas sehr häufig:

Listing 9.52: Solange-Schleife im Kochrezept

```
so lange die Fruchtmasse noch nicht sprudelnd kocht
        weiter köcheln
        weiter umrühren
```

Der Programm-Code für solche Schleifen sieht wie folgt aus:

Listing 9.53: While-Schleife

```
>>> while (eingabe != 'Ende'):
         machWas
```

Auch hier wird in einer Tabelle gegenübergestellt, wie das einfache Beispiel vom Anfang des Kapitels in Pseudocode und in Python ausgedrückt wird. Der Algorithmus, der in der linken Spalte in „Pseudocode“ beschrieben ist, kann fast eins zu eins nach Python übertragen werden.

Pseudocode	Python-Code

i = 1
solange zahl <= 100
   drucke i
    zähle i um eins hoch
	i = 1
	while i <= 100:
	   print(i)
	   i = i + 1

Tabelle 9.8.: Pseudocode und Python-Code

Für den Programmieranfänger ist eine `while`-Schleife nicht immer auf den ersten Blick verständlich. An einem ganz einfachen Beispiel will ich zeigen, wie man sich selbst das Funktionieren einer `while`-Schleife (aber auch anderer Programmstrukturen) verständlich macht. Hier kommt das Beispiel:

Listing 9.54: Was macht diese `while`-Schleife

```
>>> startwert = 1
>>> schrittweite = 2
>>> summe = 0
>>> while summe < 101:
         summe += startwert
         startwert = startwert + schrittweite
```

Nun: Du siehst wahrscheinlich sofort, welche Variable das Objekt der Begierde des Programmiers ist. Siehst Du auch sofort, welche Werte diese Variable annimmt?

Als noch nicht so erfahrener Programmierer erstelle ich mir eine Tabelle, in die ich die Variablen und Ihre Werte eintrage:

summe	schrittweite	startwert	Standpunkt
0	2	1	vor der Schleife
1	2	3	nach 1. Durchlauf des Schleifenkörpers
4	2	5	nach 2. Durchlauf des Schleifenkörpers
9	2	7	nach 3. Durchlauf des Schleifenkörpers
16	2	9	nach 4. Durchlauf des Schleifenkörpers
25	2	11	nach 5. Durchlauf des Schleifenkörpers

Tabelle 9.9.: `while`-Schleife?

Aha. Die Variable `summe` enthält die Quadratzahlen. Die erhält man rekursiv, indem man zur ersten Zahl immer die nächste ungerade Zahl addiert. (Die Veranschaulichung findest Du in der Wikipedia unter http://de.wikipedia.org/wiki/Quadratzahl)

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

Listing 9.55: Menu-Auswahl

```
1  while eingabe != 'e':
2      eingabe = input('Was willst Du? m für mehr, w für weniger, e für Ende')
3      if eingabe == 'm':
4          machmehr
5      elif eingabe == 'w':
6          machweniger
7      elif eingabe == 'e':
8          pass # für e muss man gar nichts machen; wenn das Programm zum
9               # nächsten Mal zur Überprüfung der While-Bedingung kommt,
10              # wird die Schleife einfach nicht mehr durchlaufen
11     else:
12         print('fehlerhafte Eingabe; nur m, w, e sind erlaubt')
```

`pass` ist eine Anweisung, die Python auffordert, nichts zu machen. Das ist manchmal sehr sinnvoll, denn wie in dem obigen Beispiel will (oder soll) man alle möglichen Eingaben in ein Programm berücksichtigen, aber in diesem einen Fall einer korrekten Eingabe nichts machen. Eine andere Anwendung für die `pass`-Anweisung ist, dass man bei einem längeren Projekt einen möglichen Programmzweig noch nicht programmiert hat. Die Alternative zu der Anweisung `pass` ist die Anweisung `...`. Eine gute Vereinbarung mit sich selbst könnte lauten:

- `pass` verwende ich, wenn hier wirklich nichts passieren soll. Nie!!!
- `...` verwende ich, wenn ich im Laufe der Entwicklung hier noch etwas schreiben werde

Man zeigt an, dass hier etwas passieren wird, indem man die Möglichkeit öffnet, aber man schreibt die leere Anweisung hin, weil man noch nicht weiß, was passieren wird.

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
```

9.5.3. 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 Definition von Listen und Listenelemente, manchmal als ein Dictionary gegeben.

9.5.3.1. Eine Liste von Listen abarbeiten

Auch wenn in der Überschrift zweimal „Liste“ steht: das hier gilt auch für Tupel, konkret für Listen von Tupeln oder Tupel von Listen oder Tupel von Tupeln.

Als Beispiel nehme ich die Erstellung einer Volleyball-Tabelle, und zwar speziell die Satzdifferenz. Aus meiner Datenquelle erhalte ich unter anderem die Anzahl der gewonnenen Sätze und die der verlorenen. Da bei Punktgleichheit der Wert der Differenz für die Plazierung ausschlaggebend ist, muss diese Differenz ausgerechnet werden. Seien also die gewonnenen und verlorenen Sätze in einer Liste von Paaren gegeben, jedes Paar wieder als Liste. Mit den bisher bekannten Sprachmitteln könnte das so aussehen:

Listing 9.56: Liste von Listen abarbeiten (standardmäßig)

```
saetze = [[24,5],[19, 8],[17,6],[11,18],[8,11],[5,21],[3, 18]]
>>> diffs = []
>>> for einVerh in saetze:
        eineDiff = einVerh[0] - einVerh[1]
        diffs.append(eineDiff)

>>> print(diffs)
[19, 11, 11, -7, -3, -16, -15]
```

Das ist verständlich, aber nicht elegant. Denn hier werden die gewonnenen bzw. verlorenen Sätze über ihren Index in der jeweiligen inneren Liste angesprochen. Mit einer Mehrfachzusweiung wie im Kapitel ?? Variablen geht das übersichtlicher und kürzer.

Listing 9.57: Liste von Listen abarbeiten (schöner)

```
1 saetze = [[24,5],[19, 8],[17,6],[11,18],[8,11],[5,21],[3, 18]]
2 >>> diffs = []
3 >>> for gew, verl in saetze:
4         diffs.append(gew - verl)
5 >>> print(diffs)
6 [19, 11, 11, -7, -3, -16, -15]
```

Durch die for-Schleife wird jeweils das nächste Element aus der äußeren Liste genommen. Da dieses Element wieder eine Liste aus 2 Elementen ist, werden im Schleifenkopf in einer Doppel-Zuweisung die beiden Listen-Elemente zwei Variablen zugewiesen. Im Schleifenkörper wird die Differenz dieser beiden Variablen an die Liste `diffs` angehängt.

9.5.3.2. Listen oder Dictionaries durch eine Art Liste erzeugen (list comprehensions)

List comprehensions sind Ausdrücke, die aus einer Kombination von Listen und Schleifen bestehen, um daraus eine neue Liste zu machen. Als einfaches Beispiel soll eine Liste aller Quadratzahlen zwischen 0 und 100 erstellt werden, das Ergebnis unserer Arbeit soll also `q = [0,1,4,9,16,25,36,49,64,81]` sein. Natürlich könnte man für eine solch einfache Liste seinen Editor nehmen und sie von Hand erstellen. Aber das ist nicht schön!

Eleganter ist es, eine Liste zu schreiben, die einen Ausdruck enthält. Und diese Eleganz ist in Python unübertroffen:

Listing 9.58: Liste von Quadratzahlen (list comprehension)

```
1 >>> q = [x**2 for x in range(10)]
2 >>> q
3 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`q` ist unsere neue Liste, die dadurch entsteht, dass das Quadrat jeder Zahl zwischen 0 und 10 gebildet wird. Man kann sich die obige erste Programmzeile auch laut vorlesen: q ist die Liste aller Zahlen, die dadurch gebildet werden, dass man nacheinander die Quadrate aller Zahlen zwischen 0 und 10 berechnet. Python ist wirklich eine schöne weil einfache Sprache.

Man kann zusätzlich zu der `for`-Schleife in eine list comprehension noch eine Bedingung einbauen. Das Problem, alle ungeraden Quadratzahlen von Zahlen zwischen 0 und 21 könnte man also folgendermaßen lösen:

Listing 9.59: Ungerade Quadratzahlen mit list comprehension

```
1 >>> uqz = [x*x for x in range(1,21) if x%2 == 1]
2 >>> uqz
3 [1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

Überlege Dir, wie das anders gelöst werden könnte!

Ein nächstes Beispiel: die Wertetabelle der natürlichen Exponentialfunktion für den Bereich von -2 bis 1:

Listing 9.60: Wertetabelle der natürlichen Exponentialfunktion

```
>>> from math import exp
>>> wtab = [(x/10, exp(x/10)) for x in range(-20,11)]
>>> for eintrag in wtab:
...     print(eintrag[0],'\t', eintrag[1])
...
-2.0      0.1353352832366127
-1.9      0.14956861922263506
-1.8      0.16529888822158653
-1.7      0.18268352405273466
-1.6      0.20189651799465538
-1.5      0.22313016014842982
-1.4      0.2465969639416065
-1.3      0.2725317930340126
-1.2      0.30119421191220214
-1.1      0.33287108369807955
-1.0      0.36787944117144233
-0.9      0.4065696597405991
-0.8      0.44932896411722156
-0.7      0.4965853037914095
-0.6      0.5488116360940264
-0.5      0.6065306597126334
-0.4      0.6703200460356393
-0.3      0.7408182206817179
-0.2      0.8187307530779818
-0.1      0.9048374180359595
0.0       1.0
0.1       1.1051709180756477
0.2       1.2214027581601699
0.3       1.3498588075760032
0.4       1.4918246976412703
0.5       1.6487212707001282
0.6       1.8221188003905089
0.7       2.0137527074704766
0.8       2.225540928492468
0.9       2.45960311115695
1.0       2.718281828459045
```

Das geht noch umfangreicher! Ich brauch jetzt eine Liste der Punkte im cartesischen Koordinatensystem, deren Koordinaten ganze Zahlen sind. Also muss ich Paare erstellen, auch über die y-Koordinate geschleift werden muss (der Mathematiker würde das kurz als cartesisches Produkt bezeichnen, wobei sowohl über die x-Koordinate als auch über die y-Koordinate geschleift wird).

Listing 9.61: Liste von Punkten mit ganzzahligen Koordinaten

```
>>> xyKoord = [(x,y) for x in range(3) for y in range(3)]
>>> xyKoord
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

Wenn ich eine Matrix, also eine Liste von Listen, erzeugen will, ist das auch über eine list comprehension in einer Zeile zu erreichen:

Listing 9.62: Matrix durch list comprehension erzeugen

```
qm = [[[zeile,spalte] for spalte in range(3)] for zeile in range(3)]
>>> qm
[[[0, 0], [0, 1], [0, 2]], [[1, 0], [1, 1], [1, 2]], [[2, 0],
   [2, 1], [2, 2]]]
```

Das Problem stellt sich auch für Eheanbahnungsinstitute:

Listing 9.63: Paare bilden

```
>>> m = ['Max', 'Tim', 'Kai']
>>> w = ['Isa', 'Lea', 'Jana']
>>> paare = [(junge, maedchen) for junge in m for maedchen in w]
>>> paare
[('Max', 'Isa'), ('Max', 'Lea'), ('Max', 'Jana'), ('Tim', 'Isa'),
   ('Tim', 'Lea'), ('Tim', 'Jana'), ('Kai', 'Isa'),
   ('Kai', 'Lea'), ('Kai', 'Jana')]
```

Da es tatsächlich funktioniert, Listen durch list comprehensions, die eine geschachtelte Schleife enthalten, zu erzeugen, sollte es auch möglich sein, ein Dictionary auf diese Art zu erstellen. Die Buchstaben eines Textes auf ihre Häufigkeit zu untersuchen ist eine Anwendung dieses Problems. Der Programmcode sollte selbstredend sein.

Listing 9.64: Häufigkeitsverteilung der Buchstaben eines Textes

```
>>> satz = 'Wie sieht es aus mit der Häufigkeit der Buchstaben
      in diesem Satz?'
>>> haeufigkeitsDic = {buchst: satz.count(buchst) for buchst in satz}
>>> haeufigkeitsDic
{'m': 2, 'n': 2, 'd': 3, 'e': 9, 'u': 3, 't': 5,
' ': 11, 'r': 2, 'W': 1, 'g': 1, 'B': 1, 's': 5,
'c': 1, 'H': 1, 'f': 1, 'z': 1, 'a': 3, 'ä': 1,
'h': 2, 'S': 1, 'k': 1, 'b': 1, '?': 1, 'i': 7}
```

Und jetzt soll noch aus einem Text ein Dictionary erzeugt werden. Ein Datum ist als Zeichenkette gegeben (in der üblichen deutschen Schreibweise), daraus soll ein Dictionary gebaut werden.

Listing 9.65: Text zu Dictionary machen

```
>>> datum = '15.04.2019'
>>> datVar = 'Tag Monat Jahr'

```

```
4 >>> datDic = {datVar.split()[i]:datum.split('.')[i] for i in range(3)}
5 >>> datDic
6 {'Tag': '15', 'Monat': '04', 'Jahr': '2019'}
```

Alles klar?

9.5.3.3. 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 Programmchen 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.

Listing 9.66: Länge einer Liste – zu Fuß berechnet

```
1  #!/usr/bin/python
2
3  liste = [7,12,5,23,1,14,18,33,4]
4  print(len(liste))
5
6  zaehler = 0
7  for i in liste:
8    zaehler += 1
9
10 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.

9.5.3.4. 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.

Listing 9.67: Anzahl der durch 3 teilbaren Zahlen in der Liste

```
1  #!/usr/bin/python
2
3  liste = [7,12,5,23,1,14,18,33,4]
4
5  zaehler = 0
6  for i in liste:
7    if i % 3 == 0:
8      zaehler += 1
9
10 print(zaehler)
```

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

9.5.3.5. . . . 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.

Listing 9.68: 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)
```

9.5.3.6. 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.

Listing 9.69: 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.

9.5.3.7. 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.

Listing 9.70: 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.

9.5.4. Ein bißchen Klassik

Sortieren sollte man können! Vor langer Zeit, ungefähr in der zweiten Hälfte des 20. Jahrhunderts, gab es wohl kaum ein Buch zur Informatik, in dem nicht auf die verschiedenen Möglichkeiten des Sortierens eingegangen wurde. Das will ich hier nicht, aber wenigstens die Problematik anreissen. Nehmen wir an, ich habe eine endliche Menge von Dingen, die nach einem bestimmten Kriterium sortiert werden sollen: Bücher nach dem Nachnamen, dann dem Vornamen des Autors, und falls der mehrere Bücher geschrieben hat, auch noch nach dem Erscheinungsjahr; Rock-Platten nach der Band (für schwäbische Rock-CD's: der Kapelle[10]), dann nach dem Erscheinungsjahr. Alle diese Probleme benötigen natürlich eine Methode, um zwei Exemplare nach dem jeweiligen Vergleichskriterium zu vergleichen.

Da ich nicht auf diese Vergleichskriterien eingehen will, vereinfache ich das Problem dadurch, dass ich als Dinge ganze Zahlen wähle, und dass diese nach der Größe sortiert werden sollen. Da bietet es sich natürlich an, diese Dinge in eine Liste zu schreiben, womit die Aufgabe verkürzt so formuliert wird: sortiere doch mal bitte diese Liste aufsteigend: **liste1 = [13,5,8,51,23,41,1,55,36,3,64,7,52,37,32,6,42,38,53,2,12,9,47,39,24,43,21,17]**

Eines der historisch ersten Verfahren ist unter dem Namen „Bubblesort“ bekannt. Der Name kommt daher, dass bei diesem Verfahren (je nach Vorgehen) die größten oder kleinsten Elemente wie Luftblasen aufsteigen. Und es funktioniert so:

[10] Gruß an Alex und Georg

Listing 9.71: Bubblesort im Pseudocode

```
solange noch nicht vollständig sortiert ist
    arbeite die Liste ein Element nach dem anderen von links nach rechts durch
        vergleiche das aktuelle Element mit seinem Nachfolger
        falls das aktuelle Element größer ist als der Nachfolger
            vertausche die beiden Elemente
```

Das soll einmal an einer kürzeren Liste von Hand begonnen werden.

Listing 9.72: Bubblesort einer kurzen Liste von Han

```
liste1 = [13,7,8,2]
noch nicht vollständig sortiert, also
    erstes Element: 13
        Vergleich mit dem Nachfolger 7
        13 > 7, also vertauschen
        Ergebnis:   liste1 = [7,13,8,2]
    zweites Element: 13
        Vergleich mit dem Nachfolger 8
        13 > 8, also vertauschen
        Ergebnis:   liste1 = [7,8,13,2]
    drittes Element: 13
        Vergleich mit dem Nachfolger 8
        13 > 2, also vertauschen
        Ergebnis:   liste1 = [7,8,2,13]
    (jetzt bin ich am Ende der Liste angelangt
    es ist noch nicht vollständig sortiert,
    aber immerhin ist das größte Element ans Ende gelangt,
    also weiter, genauer wieder von vorne)
    erstes Element: 7
        Vergleich mit dem Nachfolger 8
        7 < 8, also nicht vertauschen
        Ergebnis:   liste1 = [7,13,8,2]
    .... usw
```

Das Programm folgt:

Listing 9.73: Bubblesort

```
#!/usr/bin/python

l2 = [13,5,8,51,23,41,1,55,36,3,64,7,52,37,32,6,42,38,
      53,2,12,9,47,39,24,43,21,17]
sortiert = False

while sortiert == False:
  sortiert = True
  for z in range(len(l2)-1):
    if l2[z] > l2[z+1]:
      [l2[z], l2[z+1]] = [l2[z+1], l2[z]]
    else:
      sortiert = sortiert and True
    if z >= 1:
      if l2[z] < l2[z-1]:
        sortiert = sortiert and False
print(l2)
```

9.5.5. Eigentlich keine Schleife

Die eingebaute Funktion `reduce` verhält sich so ähnlich wie eine Schleife, ohne dass man diese Schleife ausdrücklich programmieren muss. Diese Funktion nimmt ein abzählbares Objekt (einen Text, der aus n Buchstaben besteht; eine Liste, die n Elemente enthält) und führt nacheinander

- eine gegebene andere Funktion mit den ersten beiden Elementen des abzählbaren Objekts aus und gibt das Ergebnis dieser Funktion zurück,
- nimmt dann dieses Ergebnis und das dritte Element, führt wieder die Funktion aus und gibt das neue Ergebnis zurück
- nimmt dieses neue Ergebnis und das vierte Element ... usw.

Ein Beispiel veranschaulicht das sofort (auch wenn Funktionen in Python noch nicht besprochen wurden; aber das folgt innerhalb der nächsten Seiten):

Listing 9.74: Hier wird eine Funktion definiert

```
def summe(x,y):
    print(x,'+',y)
    return x+y
```

Das macht die Funktion, wenn man ihr zwei Dinge übergibt:

Listing 9.75: So funktioniert diese Funktion mit 2 Zahlen

```
summe(3,5)
3 + 5
8
```

Listing 9.76: Und so mit 2 Buchstaben

```
summe('a','b')
a + b
'ab'
```

Klar, oder? Jetzt kommt `reduce` ins Spiel.

Listing 9.77: `reduce` mit dieser Funktion und einer Liste von Zahlen

```
reduce(summe, [1,2,3,4,5])
1 + 2
3 + 3
6 + 4
10 + 5
15
```

Nett! Hier werden also auch die Zwischenergebnisse sichtbar.

Listing 9.78: `reduce` mit dieser Funktion und einer Zeichenkette

```
reduce(summe, 'Martin')
M + a
Ma + r
Mar + t
Mart + i
Marti + n
'Martin'
```

Auch schön! Jetzt hast Du wohl verstanden, was `reduce` macht.

9.5.6. Eingriffe in das Durchlaufen von Schleifen

Bei diesen Eingriffen in den Schleifendurchlauf ist es egal, ob es sich um eine Zählschleife oder eine Schleife mit Abbruchbedingung handelt. In beiden Fällen kann es passieren, dass man mit manchen Elementen, die beim Durchlauf auftauchen, nichts anfangen kann, dass bei anderen Elementen genug geschleift wurde.

Das nächste Beispiel ist (zugegeben!) arg konstruiert, aber es macht klar, welcher Art Eingriffe in Schleifen vorkommen können und wie sie in Python programmiert werden. Stellen wir uns eine ziemlich große Liste von ganzen Zahlen vor, ziemlich unsortiert, und als Aufgabe, die Summe der ersten 5 geraden Zahlen zu bestimmen. Das soll hier in einer Zählschleife realisiert werden. Hier kommt der Python-Code zu diesem Problem:

Listing 9.79: Eingriff in den Schleifenablauf

```
zahlenListe = [1,5,523,274,143,76,15,72,2456,1,13,752,3416,6134,
    3475,374,347,4612,57834,437]
summe = 0
gefundenZaehler = 0
for zahl in zahlenListe:
    if zahl % 2 == 1:
        continue
    else:
        if gefundenZaehler < 5:
            summe += zahl
            gefundenZaehler += 1
        else:
            break

print(summe)
>>> 3630
print(gefundenZaehler)
>>> 5
```

In Zeile 6 wird gefragt, ob der Rest bei der Division einer Zahl durch 2 gleich 1 ist, also ob die Zahl ungerade ist. Wenn ja, dann wird durch den Befehl `continue` der Durchlauf der Schleife abgebrochen, zum Ende des Schleifenkörpers gesprungen und mit der nächsten Zahl weitergemacht. Wenn das nicht der Fall ist, wird gefragt, ob der Zähler für die gefundenen geraden Zahlen noch kleiner als 5 ist. Wenn ja, wird die aktuelle Zahl auf die Summe aufsummiert und der Zähler für die gefundenen Zahlen um 1 erhöht. Wenn der Zähler nicht mehr kleiner als 5 ist, wird durch den Befehl `break` die gesamte Schleife sofort beendet.

Auch ungewöhnlich für jemanden, der schon andere Programmiersprachen kennt: Python kennt in Schleifen ein `else`. Der Code, der von dem `else` abhängt, wird dann ausgeführt, wenn die Schleifenbedinung nicht mehr erfüllt ist, in einer Zählschleife also wenn das Ende für die Schleifenvariable erreicht ist.

Sei also eine Liste mit Zahlen gegeben. Das Problem, das es zu lösen gilt: ist in der Liste eine Quadratzahl? Wenn ja, an welcher Stelle. Der Test, ob eine Zahl eine Quadratzahl ist, wird hier ohne die Funktion `sqrt` aus dem Modul `math` gemacht; hier wird benutzt, dass $\sqrt{x} = x^{\frac{1}{2}}$ und dass eine Zahl eine Quadratzahl ist, wenn der ganzzahlige Teil der obigen Berechnung gleich der Zahl selbst ist.

Listing 9.80: Ist eine Quadratzahl in der Liste?

```
>>> zli = [13, 14, 15, 16, 17, 18, 19]
>>> for i, n in enumerate(zli):
        if (int(n**(1/2)) == n**(1/2)):
                print('Quadratzahl', n, 'an der Stelle', i)

Quadratzahl 16 an der Stelle 3
```

So weit, so gut. Aber wenn keine Quadratzahl gefunden wird, gibt dieses Programm nichts aus. Eine Meldung für den Fall des Scheiterns wäre hilfreich, und jetzt kommt das Schleifen-`else` ins Spiel:

Listing 9.81: Ist eine Quadratzahl in der Liste?

```
>>> zli = [13, 14, 15, 21, 17, 18, 19]
>>> for i, n in enumerate(zli):
        if (int(n**(1/2)) == n**(1/2)):
                print('Quadratzahl', n, 'an der Stelle', i)
        else:
                print('Keine Quadratzahl in der Liste')
Keine Quadratzahl in der Liste
```

9.5.7. Was schief gehen kann

Vor allem bei Schleifen mit Abbruchbedingung macht man als Anfänger oder als unaufmerksamer Programmierer immer wieder 2 klassische Fehler. Diese beiden Fehler sollen hier an Beispielen gezeigt werden.

Listing 9.82: Initialisierung der Schleifenvariable?

```
while x < 5:
    print(x)
```

Das ist der weniger schlimme Fehler: zu Beginn der Schleife ist die Variable `x` nicht bekannt, weil ihr bisher kein Wert zugewiesen wurde. Weniger schlimm ist das deshalb, weil das Programm an dieser Stelle einfach mit dem Fehler `NameError: name 'x' is not defined` abbricht. Damit ist der Fehler schnell korrigiert.

Listing 9.83: Verschlimmbesserung

```
x = 7
while x < 5:
    print(x)
```

Jetzt hat die Variable `x` vor Beginn der Schleife einen Wert, und der ist so gewählt, dass die Bedingung nach dem `while` erfüllt ist: die Schleife startet und gibt den Wert 7 aus. Danach geht das Programm wieder zum Schleifenbeginn zurück, der Wert der Variablen `x` ist 7, die Schleife wird ausgeführt usw. und so fort, und wenn sie nicht gestorben sind, schleifen sie noch immer. So etwas wird „Endlosschleife“ genannt. Jetzt gibt es nur 2 Möglichkeiten: wenn man Glück hat, kann man noch in irgendeiner Form das Programm abbrechen, wenn nicht, kann man nur noch den Rechner ausschalten.

Merke also: Bei Schleifen mit Abbruchbedingung muss die Variable, die die Schleife steuert, vor dem Beginn der Schleife initialisiert sein; innerhalb der Schleife muss der Wert dieser Variablen verändert werden (und zwar so, dass irgendwann die Bedingung nicht mehr erfüllt ist).

9.5.8. Aufgaben zu Schleifen

1. Schreibe eine Liste, die die Namen der Mitglieder der Gruppe enthält. Schreibe ein Programm, das ausgibt `Die Mitglieder ...von li. nach re. sind ...`
2. Überlege Dir eine Datenstruktur, die die Rollen enthält, die Eric Idle und John Cleese in den diversen Monty-Python-Filmen spielen. Schreibe ein Programm, das diese Datenstruktur ausliest und schön ausgibt.
3. Ein Python-Programm soll eine Sternchen-Pyramide malen; das Ergebnis soll so aussehen:

```
1                   *
2                  ***
3                 *****
4                *******
5               *********
6              ***********
7             *************
8            ***************
9           *****************
10         *******************
11        *********************
12       ***********************
13      *************************
14     ***************************
15    *****************************
16   *******************************
17  *********************************
18 ***********************************
19*************************************
```

4. Ein Programm soll die Quadratzahlen bis zu einer bestimmten Grenze ausgeben.
5. 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)

6. Quersumme einer Zahl ausrechnen; (beachte dabei: Typ-Konvertierung String-Element -> Zahl)
7. Primzahlen mit dem Sieb des Eratosthenes berechnen
8. Fakultät einer Zahl berechnen
9. ggT und kgV ausrechnen mit Euklids Algorithmus
10. Newtonsches Näherungsverfahren zur Berechnung von Nullstellen
11. Eine Zahl soll im Dezimalsystem eingegeben und in ein beliebiges anderes Stellenwertsystem umgerechnet werden.
12. Gegeben ist der Satz „Dieser Satz hat x Buchstaben sowie einen Punkt.“ Dabei soll für x das Zahlwort einer Zahl eingesetzt werden, zum Beispiel so: „Dieser Satz hat zwei Buchstaben sowie einen Punkt.“ Dieser Satz ist sicher falsch (denn der Satz „Dieser Satz hat zwei Buchstaben sowie einen Punkt.“ hat nicht nur 2 Buchstaben)! Schreibe ein Programm, das überprüft, ob es eine Zahl `n` gibt, so dass der Satz richtig ist und gib das Ergebnis aus.

    Schreibe ein anderes Programm, das für den Satz „Dieser Satz besteht aus n Buchstaben, zehn Leerzeichen sowie einem Komma und einem Punkt.“ die selbe Prüfung mit der selben Ausgabe macht.
13. 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.
14. 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].
15. 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.
16. Das Programm aus dem Kapitel über „Strukturierte Daten“ , das mit Daten von Aufgaben zu Listen, Dictionaries ... 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.
17. 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
18. 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.

19. Das Programm aus dem Kapitel über „Strukturierte Daten“ , das Aufgaben zu Listen, Dictionaries … behandelt, soll aufgebohrt werden, so dass ein schönes Adressbuch ausgegeben wird.
20. 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 €-Scheine, zwei 2 €-Münzen etc. herausbekommt.
21. Für eine gegebene Funktion (hier: eine ganzrationale Funktion) soll eine Wertetabelle in einem Intervall mit einer einzugebenden Schrittweite ausgegeben werden.
22. 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.
23. 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. Leerzeichen bleiben Leerzeichen, Satzzeichen werden erhalten.

    a) Die einfache Variante arbeitet mit einer Zeichenkette, die nach Buchstaben durchsucht wird. Diese Zeichenkette muss einige Buchstaben doppelt enthalten.
    b) Die nächstschwierige Variante arbeitet modulo der Länge des Alfabets.
    c) Die komfortable Lösung erlaubt auch Kleinbuchstaben und Sonderzeichen. Für diese Lösung 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!
24. 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:

Schlüssel	B	A	D
Wert	2	1	4

Tabelle 9.10.: Schlüsselwort

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.

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

Tabelle 9.11.: Verschlüsselung

Es lohnt sich, genau hinzuschauen! Der 0. und der 4. Buchstabe im Originaltext sind jeweils ein „E“ , aber das „E“ an Position 0 wird zu einem „G“ , das an Position 4 zu einem „F“ . Der Geheimtext an Stelle 4 ist ein „F“ , ebenso der an Stelle 11. Der Originaltext ist aber unterschiedlich („E“ bzw. „B“). Es gibt also keine eindeutige Zuordnung mehr!

25. Nun kommt noch eine Übungsaufgabe zur `while`-Schleife: ein bißchen Klassik. Blättere zurück zu den Datenstrukturen; dort findet sich ein Struktogramm für die Wie funktioniert das „Enthaltensein“?. Die soll jetzt programmiert werden.

26. 10-stellige ISBN:
    - Eingabe:
        - 9 Stellen der ISBN

- Ausgabe:
  - Prüfziffer
  - gesamte ISBN
- Tips:
  - Google: ISBN
  - Rest bei Division

9.6. Funktionen

9.6.1. 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“ [11], 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.

[11] So einfallsreich sind nun mal Mathematiker. Funktionen heißen fast immer „f“ , und wenn ein Mathematiker mehr als eine Funktion benötigt, besteht seine Kreativität darin, die Funktionen dann f_1 und f_2 zu nennen.

9.6.2. Eingebaute Funktionen

Python hat zum Glück schon viele eingebaute Funktionen, und die haben wir auch zum Teil schon genutzt.

- `print()` druckt die Werte aus, die in der Klammer stehen
- `len()` berechnet die Länge von Dingen, die eine Länge haben, z.B. von Texten, von Listen
- `int()` und `float()` wandeln eine Eingabe, die mit `input()` erfolgt ist, in eine Ganzzahl oder eine Fließkommazahl um.

Andere Funktionen sind in Moduln enthalten, die schon verwendet wurden.

- `sqrt()` aus dem Modul `math` berechnet die (Quadrat-)Wurzel einer Zahl
- `sin()` aus dem Modul `math` berechnet den Sinus einer Zahl.

9.6.3. Funktionen selbstgebaut

9.6.3.1. Definition und Aufruf von Funktionen

Einfache Funktionen Eine Funktion hat

- einen Funktionsnamen
- einen Funktionskopf
- einen Funktionskörper
- und manchmal einen Rückgabewert (den Funktionswert)
- oder manchmal auch mehrere Rückgabewerte

Für den Funktionsnamen gelten alle Regeln, die für Variablennamen (siehe Kapitel 5.3.2) 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: in der 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. Da es (vor allem zu Beginn der Entwicklung eines Programms) vorkommt, dass man weiß, dass ein Teil der Problemlösung durch eine Funktion erledigt werden soll, man aber sich noch nicht mit deren Details auseinandersetzen will, ist es praktisch, eine Funktion zu schreiben, die nichts tut. Dies geschieht durch die Anweisung `pass`:

Listing 9.84: Funktions-Definition einer Funktion, die nichts tut

```
1 >>> def tunix():
2         pass
```

Aber hier folgt gleich ein einfaches Beispiel, das etwas tut:

Listing 9.85: Funktions-Definition

```
1 >>> def gruessen():
2         print("Hallo und Guten Tag")
```

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

Listing 9.86: Funktionsaufruf

```
1 >>> gruessen()
2         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. Ein solcher Wert wird Parameter oder Argument der Funktion genannt. Also bohren wir das obige Beispiel auf, so dass der Gruss etwas persönlicher wird:

Listing 9.87: Funktion mit Parameter

```
1 >>> def gruessenMitName(name):
2         print("Hallo ", name, " und Guten Tag")
```

Dies wird jetzt aufgerufen mit

Listing 9.88: Funktionsaufruf mit Parameter

```
1 >>> gruessenMitName("Martin")
2         Hallo Martin und Guten Tag
```

Wie weiter oben angesprochen kommt hier eine Funktion, die ausgibt, ob eine Zahl zwischen zwei anderen Zahlen liegt.

Listing 9.89: Funktion mit 3 Parametern: zwischen

```
def liegtZwischen(z,unten,oben):
    return unten < z < oben

erg =   liegtZwischen(3, 0, 10)
print(erg)
>>> True
erg =   liegtZwischen(8, 0, 10)
print(erg)
>>> False
erg2 = liegtZwischen('e', 'a', 'h')
print(erg2)
>>> True
```

Und die letzten 3 Zeilen wundern auch nicht mehr: Python kann auch zum Beispiel überprüfen, ob ein Buchstabe zwischen zwei anderen liegt. Hier sollte man vielleicht einmal probieren, ob diese Funktion auch mit anderen Daten als Zahlen und Buchstaben klar kommt.

9.6.3.2. Rückgabewerte und Parameterlisten

Bis jetzt hat eine Funktion etwas gemacht, nicht mehr und nicht weniger. In mancher anderen Programmiersprache [12] wird so etwas „Prozedur“ genannt. Funktionen unterscheiden sich in diesen Sprachen von Prozeduren, dass sie etwas zurückgeben. Das soll an einem einfachen Beispiel vorgeführt werden.

Listing 9.90: Funktion ohne Rückgabewert

```
def summeAusdrucken(a, b):
    print(a + b)
```

Dazu muss man nichts sagen! Beim Aufruf dieser Funktion wird einfach die Summe der beiden Parameter ausgegeben.

Listing 9.91: Aufruf der Funktion ohne Rückgabewert

```
summeAusdrucken(5,7)
>>> 12
```

Jetzt soll eine Funktion diese Summe zurückgeben. Das reservierte Wort, das einen Rückgabewert zurückgibt, heißt „return“ . So sieht das aus:

Listing 9.92: Funktion mit Rückgabewert

```
def summeZurueckgeben(a, b):
    return a + b
```

Diese Funktion wird anders aufgerufen.

[12] z. B. in den Wirth-Sprachen

Listing 9.93: Aufruf der Funktion mit Rückgabewert

```
ergebnis = summeZurueckgeben(5,7)
print(ergebnis)
>>> 12
```

Es folgt ein weiteres Beispiel dazu, und zwar sowohl die Definition als auch die Ausführung:

Listing 9.94: Funktion mit Rückgabewert (Programm und Ausgabe)

```
 >>> 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:

Listing 9.95: nochmal: Funktion mit Rückgabewert

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

Es fehlt noch eine Funktion mit mehr als einem Rückgabewert. Dazu schauen wir uns an, was die Funktion sqrt aus dem math-Modul macht.

```
>>> from math import sqrt
>>> w = sqrt(4)
>>> print(w)
2.0
```

So weit nicht schlecht. Aber in Mathematik haben wir gelernt, dass es aus einer Zahl 2 Quadratwurzeln gibt, im obigen Fall ausser der 2 noch die -2. Also müssen wir die Funktion umschreiben:

```
def wurzeln(zahl):
    from math import sqrt
    w1 = sqrt(zahl)
    w2 = -w1
    return w1, w2

quadratwurzeln = wurzeln(4)
print(quadratwurzeln)
(2.0, -2.0)
```

Das, was hinter dem „return“ steht, das, was zurückgegeben wird, ist ein Tupel (ohne runde Klammern). Deswegen steht in der Variablen „quadratwurzeln“ ein Tupel (in runden Klammern).

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

Listing 9.96: Funktion mit mehreren Parametern

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

Listing 9.97: Parameterübergabe mit Parameternamen

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

Listing 9.98: 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:

Listing 9.99: 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:

Listing 9.100: 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:

Listing 9.101: Nochmal: Unbekannte Anzahl Parameter

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

Wenn einer Funktion Schlüsselparameter übergeben werden, also Parameter mit Parameternamen, dann soll daraus ein Dictionary gemacht werden. Das geschieht mit den „**“ . Die Erklärung ergibt sich aus dem folgenden Beispiel:

Listing 9.102: Ein Dictionary als Ausgabe

```
>>> def dicParameterAusgeben(**woerterbuch):
      print(woerterbuch)
```

```
3 >>> dicParameterAusgeben(Kuh='cow',Hund='dog',Vogel='bird')
4   {'Hund': 'dog', 'Vogel': 'bird', 'Kuh': 'cow'}
```

9.6.3.3. Nachtrag zu Listen

Weiter oben im Skript, bei den Listen, ging es um die Aufgabe, aus einer Liste mithilfe einer anderen Funktion, die jedes Element einer Liste bearbeitet, eine gefilterte Liste zu erzeugen. Als Beispiel sollen eine Liste der Quadratzahlen kleiner als 650 erzeugt werden. Dazu wird eine Funktion geschrieben, die auf „Quadratzahl-sein“ überprüft.

Listing 9.103: Filtern von Listen

```
1  from math import sqrt
2  def qZahl(z):
3      if sqrt(z) == int(sqrt(z)):
4          return z
5      else:
6          return ''
7  qs = filter(qZahl,range(650))
8  for z in qs:
9      print(z,end=', ')
10 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289,
11 324, 361, 400, 441, 484, 529, 576, 625,
```

Weiter oben bei Listen wurde bereits erwähnt, dass eine Sortierung einer Liste nach beliebigen Kriterien geschehen kann, wenn man das Kriterium in einer Funktion beschreibt.

Listing 9.104: Sortierung einer Liste nach eigenen Kriterien (Programm)

```
1  def sortiere_nach_wortlaenge(wort):
2      return len(wort)
3
4  woerter = ['Martin', 'ist', 'doof']
5
6  woerter.sort(key=sortiere_nach_wortlaenge)
7
8  print(woerter)
9
10 def sortiere_nach_zweitem_Buchstaben(wort):
11     return wort[1]
12
13 woerter.sort(key=sortiere_nach_zweitem_Buchstaben)
14 print(woerter)
15
16 liste = ['Andreas Mueller-Morgenschoen','Brigitte Maier','Carlo Carlowitsch',
17 'Daniela Deppele']
18 def nachnameExtrahieren(name):
19     lz = name.index(' ')
20     return name[lz+1:]
21
```

```
22 liste.sort(key = nachnameExtrahieren)
```

Das Programm liefert die folgenden Ausgaben:

Listing 9.105: Ergebnis der Sortierung einer Liste nach eigenartigen Kriterien

```
1 ['ist', 'doof', 'Martin'] # Sortierung nach Wortlänge
2 ['Martin', 'doof', 'ist'] # Sortierung nach dem 2. Buchstaben
3 ['Carlo Carlowitsch', 'Daniela Deppele', 'Brigitte Maier',
4 'Andreas Mueller-Morgenschoen'] # Sortierung nach Nachname
```

Eine Klasse, die die Arbeit mit Listen erleichtern kann, ist die Klasse `enumerate`. Es hört sich vielleicht seltsam an, dass um eine Liste, die ja abzählbar ist, ein „Umschlag“ herumgemacht wird, der ein abzählbares Objekt ist. Ein Beispiel veranschaulicht das aber: ich will bei einer Liste nicht nur die einzelnen Listenelemente ausgeben, sondern auch die „Hausnummer“ . Mit den bisher bekannten Sprachelementen ist das möglich (man könnte das auch gut als Übung machen, bevor man weiterliest), aber mit der Erzeugung eines `enumerate`-Objektes sieht das einfach aus:

Listing 9.106: Listenausgabe mit enumerate

```
>>> zw = ['null', 'eins', 'zwei', 'drei', 'vier']
>>> for zaehler, element in enumerate(zw):
        print( 'zw an der Stelle', zaehler, 'ist',element)

zw an der Stelle 0 ist null
zw an der Stelle 1 ist eins
zw an der Stelle 2 ist zwei
zw an der Stelle 3 ist drei
zw an der Stelle 4 ist vier
```

9.6.3.4. Verkettung von Funktionen

In der Mathematik wird die Verkettung von Funktionen spätestens in der Oberstufe des Gymnasiums relevant: da taucht dann vielleicht etwas in der Art von $f(x) = \sin(e^x)$ auf. Und die Schüler mögen es gar nicht!

Dabei ist das Prinzip ganz einfach: man nehme ein x, gebe es in eine erste Funktion hinein, und diese erste Funktion liefert ein Ergebnis zurück; dieses Ergebnis gebe man in eine zweite Funktion hinein, und diese liefert das Endergebnis zurück. Bei dem Beispiel aus dem vorigen Absatz wird das x zuerst in die natürliche Exponentialfunktion reingeschmissen, und als Ergebnis kommt dort e^x heraus. Von diesem Ergebnis wird jetzt der Sinus gebildet. Fertig.

In der Programmierung gibt es das natürlich auch. Hier folgt ein Beispiel, bei dem eine Zahl zuerst durch eine erste Funktion verdoppelt wird. Das Ergebnis wird dann durch eine zweite Funktion quadriert.

Listing 9.107: Verkettung von Funktionen

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

def quadrieren(z):
    return z*z

zahl = 3
zwischenergebnis = verdoppeln(zahl)
endergebnis = quadrieren(zwischenergebnis)
print(endergebnis)
>>> 36
```

Das funktioniert! Aber es sieht nicht schön aus. Die beiden Funktionen sind kurz, der Name sagt aus, was sie jeweils machen, also bleiben die beiden Funktionen so. Der Aufruf hingegen kann verbessert werden. Also vereinfachen (und verschönern) wir das:

Listing 9.108: Verkettung von Funktionen – einfacher

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

def quadrieren(z):
    return z*z

zahl = 3
endergebnis = quadrieren(verdoppeln(zahl))
print(endergebnis)
>>> 36
```

Den Mathematiker freut es, denn so schreibt er auch die Verkettung von Funktionen. Wenn zuerst die Funktion f_1 mit dem Argument x und dann die Funktion f_2 mit dem Ergebnis der ersten Funktion als Argument aufgerufen werden soll, dann schreibt der Mathematiker $f_2(f_1(x))$

9.6.3.5. 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:

Listing 9.109: 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:

Listing 9.110: Ausgabe: 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.

Die Variable, die innerhalb der Funktion definiert wird, gehört zum Gültigkeitsbereich der Funktion. Hier ist sie bekannt, außerhalb nicht. Anders sieht es aus, wenn eine Variable im Hauptprogramm definiert wird. Eine solche Variable gehört zum Gültigkeitsbereich des Hauptprogramms und damit auch zum Gültigkeitsbereich aller Funktionen, die innerhalb des Hauptprogramms definiert werden. Man nennt eine solche Variable, die im ganzen Programm gültig ist, eine „globale Variable“ . Das soll an einem Beispiel verdeutlicht werden:

Listing 9.111: verschachtelte Aufrufe von Funktionen

```
#!/usr/bin/env python3

def innereFktErsterEbene():
    var1E = '\tVariable der 1. Ebene'
    print(var1E)
    print('\tAufruf aus 1. Ebene: '+varHauptPgm)

    def innereFktZweiterEbene():
        var2E = '\t\tVariable der 2. Ebene'
        print(var2E)
        print('\t\tAufruf aus 2. Ebene: '+var1E)

    innereFktZweiterEbene()

varHauptPgm = 'VARIABLE DES HAUPTPROGRAMMS'
print(varHauptPgm)
innereFktErsterEbene()
```

Innerhalb der Funktion `innereFktErsterEbene` wird hier die Variable `varHauptpgm` aufgerufen, ebenso in der Funktion `innereFktZweiterEbene`. Das sieht so aus:

Listing 9.112: Ausgabe des Programms zu verschachtelten Funktionen

```
>>> VARIABLE DES HAUPTPROGRAMMS
    Variable der 1. Ebene
    Aufruf aus 1. Ebene: VARIABLE DES HAUPTPROGRAMMS
        Variable der 2. Ebene
        Aufruf aus 2. Ebene:    Variable der 1. Ebene
```

9.6.4. 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[13], 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

Listing 9.113: Funktion `verdoppeln`

```
1 def verdoppeln(zahl):
2     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:

[13] und dabei denken wir nicht nur an Karl-Theodor z. G.

Listing 9.114: Aufruf der Verdoppelung (1.)

```
import textfunktionen

print(textfunktionen.verdoppeln(13))
```

Als nächstes folgt der Import mit „from“ :

Listing 9.115: Aufruf der Verdoppelung (2.)

```
from textfunktionen import verdoppeln

print(verdoppeln(13))
```

9.6.5. Ein Trick, um Funktionen zu testen

Meistens, und wir kommen später noch darauf, ist es so, dass eine Funktion in einer separaten Datei gespeichert wird, und diese Datei dann importiert wird. Die Vorteile dieses Vorgehens sind klar:

- etwas wird nur einmal geschrieben, und kann dann an verschiedenen Stellen, von verschiedenen Programmen benutzt werden
- das Programm, an dem man gerade arbeitet, bleibt kurz und übersichtlich, weil der Code einer Funktion nicht mehr in diesem Programm steht, sondern separat

Trotzdem kann es vorkommen, dass man an einer Funktion nachträglich etwas ändern muss, und dann aber nicht alle Programme testen will, die diese Funktion benutzen. Da wäre es doch schön, wenn man innerhalb der Datei, die die Funktion enthält, einen Test eingebaut hätte, der nur dann ausgeführt wird, wenn diese Funktion getestet werden soll, aber nicht, wenn sie im produktiven Einsatz ist.

Natürlich gibt es diese Möglichkeit in Python. Das Mittel dazu ist die eingebaute Variable `__name__`.

Listing 9.116: Builtin `__name__`

```
>>> print('Die Variable __name__ hat den Wert ', __name__)
Die Variable __name__ hat den Wert  __main__
```

So sieht das aus, wenn der Wert der Variablen `__name__` von der Python-Shell aus abgefragt wird: diese Variable hat jetzt den Wert `__main__`. Als nächstes schreibe ich ein kleines Programm, das nur den obigen Befehl enthält und speichere es unter dem Dateinamen `echoName.py`.

Listing 9.117: Builtin `__name__` in einer Datei

```
#!/usr/bin/python
print('Die Variable __name__ hat den Wert ', __name__)
```

Und dieses Datei importiere ich:

Listing 9.118: Aufruf dieser Datei

```
>>> import echoName
Die Variable __name__ hat den Wert  echoName
```

Erstaunlich!! Jetzt steht in der Variablen `__name__` der Name der importierten Datei! Das ist doch die Lösung des Problems, einen Test einer Funktion nur durchzuführen, wenn wirklich getestet werden soll, nämlich dann, wenn diese Funktion aus der Datei heraus, in der die Funktion steht, aufgerufen wird.

Listing 9.119: Anwendung für den Test einer Funktion

```
#!/usr/bin/python

def irgendwas():
    print('irgendwas')

if __name__ == '__main__':
    irgendwas()
```

9.6.6. 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[14]“ Deswegen gilt: Variable sind immer überall gültig, sie sind global. Das zeigen wir an einem Beispiel:

Listing 9.120: eine globale Variable

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

globaleVar = 'glooooobal!!'

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

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

Die Ausgabe sieht so aus:

Listing 9.121: Ausgabe von globaler Variablen

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

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

[14] [30], Seite 285

Listing 9.122: Globale Variable und eine Funktion mit lokaler Variablen

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

globaleVar = 'glooooobal!!'

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:

Listing 9.123: Ausgabe von globaler und lokaler Variablen

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

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:

Listing 9.124: Globale Variable und eine Funktion mit lokaler Variablen

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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:

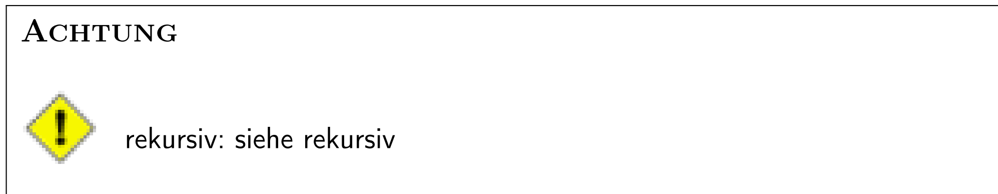
Listing 9.125: Ausgabe von globaler und lokaler Variablen

```
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"!!
```

9.6.7. Rekursive Funktionen

> Die schärfsten Kritiker der Elche
> waren früher selber welche.
>
> *(F.W. Bernstein[15])*

Was heißt „rekursiv“? Schauen wir in einem Wörterbuch (geschrieben von Unix-Fachleuten) nach:



Eine rekursive Funktion ist eine Funktion, die in ihrem Funktionskörper sich selbst aufruft.

Es ist manchmal wirklich sinnvoll, 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.

Listing 9.126: Fakultät rekursiv

```
1 def fak_rek(zahl):
2   if zahl == 1:
3     return 1
4   else:
5     return zahl*fak_rek(zahl-1)
```

Und der Aufruf dieser Funktion sieht ganz einfach aus:

Listing 9.127: Aufruf von Fakultät rekursiv

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

9.6.8. 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.

[15] Die Wahrheit über Arnold Hau, Frankfurt 1974, S. 87

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:

Listing 9.128: 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`

Listing 9.129: Gross-Schreiben eines Begriffs in einer Zeile

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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

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

Listing 9.130: Rahmen für Bearbeitung einer Text-Datei

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

def textBearbeiten(????):
  eingabeDatei = open('testTextFktInFkt.txt', 'r')
  wort = 'Python'
  for eineZeile in eingabeDatei.readlines():
    if wort in eineZeile:
      ????(eineZeile, wort)
  eingabeDatei.close()

if __name__ == '__main__':
  textBearbeiten(????)
```

Auch wenn Dateien erst später im Text (siehe Kapitel 15) 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:

Listing 9.131: Python wird gross geschrieben

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

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!

9.6.8.1. Seiteneffekte

Seiteneffekte sind etwas ganz böses!!!

9.6.9. Aufgaben 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.
7. Ein Radler, der einen Programmierkurs in Python besucht hat, hat sich für die Einstellung seines neuen 4-stelligen Zahlenschlosses folgende Methode ausgedacht:
   - am ersten Tag stellt er eine 4-stellige Zahl ein
   - an jedem der folgenden Tage berechnet er die Quersumme (falls die nicht einstellig ist, die Quersumme der Quersumme, falls die nicht einstellig ist ...) der 4-stelligen Zahl, streicht die erste Ziffer dieser 4-stelligen Zahl und fügt als letzte Ziffer die Quersumme hinten an.

   Z.B. so: Start mit 1954, Quersumme $= 1 + 9 + 5 + 4 = 19$, davon Quersumme $= 1 + 9 = 10$, davon Quersumme $= 1 + 0 = 1$, neue Zahl also 9541.
   Und jetzt stellt er sich wichtige Fragen:
   a) Taucht irgendwann wieder die Anfangszahl bei diesem Verfahren auf?
   b) Wenn ja: hoffentlich nicht so bald!!! Sonst bedeutet diese Methode keinen Gewinn an Sicherheit.
   c) Kann ich ein Programm schreiben, das diese Methode simuliert?
   d) Kann bei diesem Programm etwas Unerwünschtes passieren?

   Da er die 3. Frage nach seinem Python-Kurs bejahen kann, macht er sich an die Arbeit!!! (angepasst an eine Aufgabe der Uni Freiburg, zu finden unter http://www.informatik.uni-freiburg.de/~ki/teaching/ws1011/planningpractical/python_de.html

9.6.10. lambda-Funktionen

lambda-Funktionen sind eigentlich lambda-Ausdrücke, aber sie arbeiten wie Funktionen. Sie werden auch oft „anonyme Funktionen" genannt, denn diese Ausdrücke haben keinen Namen. Allerdings kann ein lambda-Ausdruck einem Namen zugewiesen werden. Ein lambda-Ausdruck gibt immer einen Wert zurück.

Die Syntax eines lambda-Ausdrucks besteht aus dem Schlüsselwort `lambda`, gefolgt von einer oder mehreren Variablen, gefolgt von einem Doppelpunkt. Als letztes steht der Ausdruck, der etwas mit der oder den Variablen macht. Als Beispiel soll hier eine Berechnung der nächsten Stundenzahl (12-Stunden-Angaben) dienen; der Nachfolger von 6 ist 7, der Nachfolger von 12 ist aber 1.

Listing 9.132: Uhrzeiten (Stunden) addieren: der Ausdruck

```
lambda x: (x+1)%12
>>> (lambda x: (x+1)%12)(6)
7
>>> (lambda x: (x+1)%12)(12)
1
```

Dieser lambda-Ausdruck nimmt einen Wert für die Variable `x` entgegen, addiert 1 dazu und berechnet den Rest modulo 12. Um diesen lambda-Ausdruck zu benutzen, muss er in Klammern geschrieben werden, gefolgt von dem Wert des Parameters auch in Klammern.
Jetzt wird diesem Ausdruck ein Variablenname gegeben und dann aufgerufen.

Listing 9.133: Uhrzeiten (Stunden) addieren (benannt)

```
>>> stundeAddieren = lambda x: (x+1)%12
>>> stundeAddieren(6)
7
>>> stundeAddieren(12)
1
```

Elegant sind lambda-Ausdrücke als Parameter für andere Funktionen. Ein Beispiel ist die Sortierung einer Liste nach einem etwas ungewöhnlichen Sortierbegriff. Die Sortierung einer Liste erlaubt als Parameter das Schlüsselwort `key`, dem eine Funktion zugewiesen wird, die die Regel für die Sortierung enthält. Hier soll eine Namensliste nach dem letzten Buchstaben des Namens sortiert werden.

Listing 9.134: Liste sortieren nach letztem Buchstaben

```
>>> namensliste = ['Karl','Eva','Anne','Jens','Kim','Mary']
>>> namensliste.sort(key = lambda x: x[-1])
>>> namensliste
['Eva', 'Anne', 'Karl', 'Kim', 'Jens', 'Mary']
```

Jetzt soll ein Dictionary sortiert werden, das als Schlüssel einen Vornamen hat, als Wert eine Liste bestehend aus dem Nachnamen und dem Wohnort. Sortiert werden soll nach dem Wohnort.

Listing 9.135: Geschachteltes Dictionary sortieren

```
>>> adrDic = {'Eva':['Maier','Stuttgart'], 'Anne':['Wahn','Pforzheim'],
              'Karl':['Heim','Kassel'], 'Kim':['Mann','Hamburg'],
              'Jens':['Schmidt','Berlin'], 'Mary':['Miller','Boston']}
>>> adrListe = list(zip(adrDic.keys(),adrDic.values()))
>>> adrListe.sort(key = lambda x: x[1][1])
>>> for einer in adrListe:
...     print(einer[0]+' '+einer[1][0]+' aus '+einer[1][1])
...
Jens Schmidt aus Berlin
Mary Miller aus Boston
Kim Mann aus Hamburg
Karl Heim aus Kassel
Anne Wahn aus Pforzheim
Eva Maier aus Stuttgart
```

Also wird zuerst aus den Dictionary-Keys und den Dictionary-Values eine Liste gezippt (denn Dictionaries kennen keine Sortierung). Diese Liste wird mittels eines Lambda-Ausdrucks sortiert und dann ausgegeben. Der lambda-Ausdruck holt aus der Liste das 1. Element (die Liste, die aus Nachname und Wohnort besteht) und daraus das 1. Element (den Wohnort).

9.6.11. Sicherheit, Effizienz, Eleganz

In diesem Kapitel soll jetzt gezeigt werden, wie man durch die Erstellung und Benutzung von Funktionen Programme sicherer und effizienter macht. Zusätzlich gewinnt man dabei Übersichtlichkeit und macht dadurch ein Programm besser wartbar. Dazu nehmen wir ein einfaches Problem der Mathematik der Mittelstufe: berechne die Länge der Hypotenuse eines rechtwinkligen Dreiecks, wenn die Längen der beiden Katheten gegeben sind.

Die einzelnen Schritte zur Lösung dieses Problems werden hier aufgelistet:

1. Gib die Längen der beiden Katheten ein.
2. Berechne die Länge der Hypotenuse mit Hilfe des Satzes von Pythagoras.
3. Gib das Ergebnis ordentlich aus.

Der erste Punkt muss für jede der beiden Katheten erledigt werden. Das ist ein guter Grund, dafür eine Funktion zu schreiben, die als Parameter die Ordnungszahl der Kathete (erste oder zweite) erhält und den Wert zurückgibt und in einer Variablen speichert.

Eingedenk der Anforderung, dass eine Funktion nur eine Sache machen soll, diese aber richtig, schreibt man dann noch zwei Funktionen: eine Funktion, die als Parameter die Kathetenlängen erhält und die Hypotenusenlänge zurückgibt, und eine Funktion, die für die Ausgabe zuständig ist.

Hier kommt die erste Funktion, zusammen mit einem Test:

Listing 9.136: Eingabe der Kathetenlängen

```
#!/usr/bin/python

def floatEingeben(hsnr):
    prompt = 'Bitte Länge der '+str(hsnr)+'. Kathete eingeben: '
    wert = float(input(prompt))
    return wert

w1 = floatEingeben(1)
print(w1)
w2 = floatEingeben(2)
print(w2)

>>> Bitte Länge der 1. Kathete eingeben: 3
3.0
Bitte Länge der 2. Kathete eingeben: 3.8
3.8
```

Gut so weit!

Die nächste Funktion berechnet die Länge der Hypotenuse.

Listing 9.137: weiter: Hypotenusenlänge berechnen

```
#!/usr/bin/python

def floatEingeben(hsnr):
    prompt = 'Bitte Länge der '+str(hsnr)+'. Kathete eingeben: '
    wert = float(input(prompt))
    return wert

def hypotenusenLaengeBerechnen(w1, w2):
    from math import sqrt
    return sqrt(w1**2 + w2**2)

k1 = floatEingeben(1)
print(k1)
k2 = floatEingeben(2)
print(k2)
hy = hypotenusenLaengeBerechnen(k1, k2)
print(hy)

>>> Bitte Länge der 1. Kathete eingeben: 3
3.0
Bitte Länge der 2. Kathete eingeben: 4
4.0
5.0
```

Auch das sieht gut aus. Wer das jetzt selbst nachvollzieht, merkt, dass die Struktur des Programms klar wird und mögliche Fehler bei der Programmierung recht leicht vermieden werden, weil die Funktionen kurz und knapp sind.

Jetzt fehlt nur noch die schöne Ausgabe, denn das da oben, wo einfach eine Zahl hingelatzt wird, ist ja nicht sehr benutzerfreundlich.

Listing 9.138: weiter: schöne Ausgabe

```
1  #!/usr/bin/python
2
3  def floatEingeben(hsnr):
4      prompt = 'Bitte Länge der '+str(hsnr)+'. Kathete eingeben: '
5      wert = float(input(prompt))
6      return wert
7
8  def hypotenusenLaengeBerechnen(w1,  w2):
9      from math import sqrt
10     return sqrt(w1**2 + w2**2)
11
12 def ergebnisAusgeben(k1, k2, hy):
13     print('Das rechtwinklige Dreieck mit den Kathetenlängen '
14                 +str(k1)+' und '+str(k2))
15     print('hat eine Länge der Hypotenuse von '+str(hy))
16
17 k1 = floatEingeben(1)
18 k2 = floatEingeben(2)
19 hy = hypotenusenLaengeBerechnen(k1, k2)
20 ergebnisAusgeben(k1, k2, hy)
21
22 >>> Bitte Länge der 1. Kathete eingeben: 3
23 Bitte Länge der 2. Kathete eingeben: 4
24 Das rechtwinklige Dreieck mit den Kathetenlängen 3.0 und 4.0
25 hat eine Länge der Hypotenuse von 5.0
```

Teil VI.

Programmentwicklung und Modularisierung

10. Programmentwicklung mit Test

10.1. Der Doctest

Da jetzt schon größere Programme geschrieben werden, vor allem, da jetzt auch Funktionen (Unterprogramme) eingebaut werden, ist es sinnvoll, dass der Entwickler sich schon beim Programmieren Gedanken macht, was denn von einzelnen Funktionen erwartet wird. Auch da ist Python sehr hilfreich, denn es kennt den `doctest`.

Der `doctest` funktioniert so, dass man in eine Funktion, direkt hinter den Funktionskopf(siehe S. 243) eine Zeichenkette in dreifachen Anführungszeichen schreibt. Diese Zeichenkette enthält in der ersten Zeile nach 3 Größer-Zeichen einen möglichen Funktionsaufruf, darunter in der nächsten Zeile das erwartete Ergebnis.

Selbstverständlich kann man in diese Zeichenkette auch mehr als einen Test schreiben. Dann wiederholt sich wieder das Paar „Zeile mit 3 Größer-Zeichen, gefolgt von einem möglichen Funktionsaufruf, gefolgt von einer neuen Zeile mit dem erwarteten Ergebnis"
.

Die Beschreibung, wie eine solche Funktion mit eingebautem Test aussieht, hört sich ziemlich umständlich an. Ein Beispiel macht das aber hoffentlich klar. Es wird eine Funktion geschrieben, die bei 3 Zahlen (die der Einfachheit halber der Größe nach aufsteigend eingegeben werden) überprüft, ob es sich um ein pythagoräisches Zahlentripel handelt, ob also erste Zahl zum Quadrat plus zweite Zahl zum Quadrat die dritte Zahl zum Quadrat ergibt.

Listing 10.1: Doctest (1. Versuch! Der tut noch nicht richtig!)

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  def pythTripel(a,b,c):
5      """
6        >>> pythTripel(3,4,5)
7        True
8        >>> pythTripel(3,4,6)
9        False
10     """
11
12 import doctest
13 doctest.testmod()
```

Hier wird also getestet, ob (3,4,5) ein pythagoräisches Zahlentripel ist, und bei diesem Test wird ein `True` erwartet, dann wird (3,4,6) getestet, und bei diesem Test wird ein

False erwartet. Das alles steht innerhalb der Zeichenkette. Das Hauptprogramm importiert doctest und ruft danach den Test auf. Die Ausgabe ist noch nicht ganz das, was wir im Endeffekt erwarten. Aber das ist ja klar, denn die Funktion pythTripel macht noch nichts.

Listing 10.2: Ausgabe des Doctest (1. Versuch! Der tut noch nicht richtig!)

```
**********************************************************************
File "/home/fmartin/bin/python/L14a_Doctest/pythTrip0.py",
                     line 6, in __main__.pythTripel
Failed example:
    pythTripel(3,4,5)
Expected:
    True
Got nothing
**********************************************************************
File "/home/fmartin/bin/python/L14a_Doctest/pythTrip0.py",
                     line 8, in __main__.pythTripel
Failed example:
    pythTripel(3,4,6)
Expected:
    False
Got nothing
**********************************************************************
1 items had failures:
   2 of   2 in __main__.pythTripel
***Test Failed*** 2 failures.
```

Man kann es wohl sofort verstehen, was da passiert. Bei beiden Tests steht ja da, was erwartet wird, nämlich `Expected: True` bzw. `Expected: False`, und danach kommt das traurige Ergebnis, dass gar nichts passiert ist: `Got nothing`. Das Fazit: 2 Tests wurden durchgeführt, beide gingen schief (`**Test Failed*** 2 failures`).

Wir wissen ja schon, woran es liegt: die Funktion tut noch gar nichts! Das können wir schnell ändern, indem wir einfach von der Funktion zurückgeben lassen, ob $a^2+b^2 == c^2$ ist, also entweder den Wert True oder den Wert False.

Listing 10.3: Doctest (2. Versuch! Jetzt sollte ein sinnvolles Ergebnis erscheinen)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

def pythTripel(a,b,c):
  """
    >>> pythTripel(3,4,5)
    True
    >>> pythTripel(3,4,6)
    False
  """
  return a*a + b*b - c*c == 0

import doctest
doctest.testmod()
```

Es folgt die Ausgabe:

Listing 10.4: Ausgabe des Doctest (Jetzt sollte ein sinnvolles Ergebnis erscheinen)

```
>>>
>>>
```

Die Ausgabe ist das absolute Nichts!!! Hm, was soll man damit anfangen? Ok, das ist zwar nicht das, was sich der Programmieranfänger wünscht, aber es ist korrekt: es wird nichts ausgegeben, weil alle Tests korrekt absolviert wurden.

Zum Glück gibt es aber noch einen Schalter, der auch in diesem Fall eine Ausgabe liefert, die uns mehr Informationen bietet: `verbose`, also wörtlich (oder vielleicht eher geschwätzig).

Listing 10.5: Doctest (3. Versuch! Jetzt gibt es ein sinnvolles Ergebnis)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

def pythTripel(a,b,c):
  """
    >>> pythTripel(3,4,5)
    True
    >>> pythTripel(3,4,6)
    False
  """
  return a*a + b*b - c*c == 0

import doctest
doctest.testmod(verbose=True)
```

Jetzt sieht es doch schön aus:

Listing 10.6: Geschwätzige Ausgabe des Doctest

```
Trying:
    pythTripel(3,4,5)
Expecting:
    True
ok
Trying:
    pythTripel(3,4,6)
Expecting:
    False
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.pythTripel
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

Jetzt wird also ausdrücklich angegeben, was getestet wird, welches Ergebnis erwartet wird, und zusätzlich die Information, dass der Test „ok“ war.

10.1.1. Aufgaben zu doctest

1. Hol Dein Programm zur Lösung von quadratischen Gleichungen nochmals hervor, mach eine Kopie davon und baue ein:
   a) Eine Funktion, die die Diskriminante berechnet.
   b) Eine Funktion, die eventuelle reelle Lösungen berechnet
   c) In jede der beiden Funktionen einen doctest-Block, der das korrekte Funktionieren überprüft.

11. Module und Pakete

11.1. Mehr Mathematik

Mathematik in Python hat zu tun mit der Art, wie Mathematiker arbeiten. Wenn sie einen neuen Satz gefunden haben, müssen sie dessen Richtigkeit beweisen. Aber kein Mathematiker[1] wird bei Adam und Eva anfangen oder bei den natürlichen Zahlen, sondern jeder wird auf anderen Sätzen aufbauen, die andere Mathematiker zuvor bewiesen haben.

In Python müssen nicht nur John Cleese und Eric Idle zugange gewesen sein, sondern auch Mathematiker: es gibt jede Menge „Python-Sätze“ , von denen man weiß[2], dass sie richtig sind. Damit muss man als Python-Programmierer nicht mehr bei den kleinsten elementaren Anweisungen anfangen, wenn man ein neues Programm schreibt, sondern kann vieles benutzen, was irgendwo schon vorhanden ist. Und davon handelt dieses Kapitel.

Im Kapitel 5 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:

Listing 11.1: Fehler! Python kann kein Mathe!

```
1  >>> rad = 2.68
2  >>> umf = 2 * pi * rad
3     Traceback (most recent call last):
4     File "<pyshell#2>", line 1, in <module>
5       umf = 2 * pi * rad
6     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:

[1] nicht vergessen: Mathematiker sind faul
[2] weil es jemand bewiesen hat

Listing 11.2: 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:

Listing 11.3: 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:

Listing 11.4: 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:

Listing 11.5: 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.
```

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.

Das Inhaltsverzeichnis des Moduls `math` zeigt nicht so detailliert an, aber der Überblick ist hilfreich, denn so weiß man, wonach man weiter blättern kann:

Listing 11.6: Inhaltsverzeichnis des Moduls `math`

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2',
'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod',
'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Listing 11.7: Noch mehr Mathematik

```
>>> from math import cos, sin, pi
>>> x = 1.234
>>> sin(x)
0.9438182093746 3368
>>> sin(pi)
1.2246063538223773e-16
>>> cos(pi)
-1.0
```

11.2. 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:

Listing 11.8: Eine eigene Modul-Datei

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

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

Listing 11.9: Aufruf des eigenen Moduls

```
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3
4 import eigeneModule
5
6 print(eigeneModule.fak(5))
7 print(eigeneModule.fibo(5))
```

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

- `eigene Module.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:

Listing 11.10: Eine eigene Modul-Datei mit Hilfetext

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  '''
4  bisher vorhanden:
5    fak(z): berechnet die Fakultät von z
6    binko(n,k): berechnet n über k (Binomialkoeffizient
7    fibo(z): berechnet die n-te Fibonacci-Zahl
8  '''
9  from fakBinko import fak, binko
10 from fibo import fibo
```

Die Hilfe kann ich nach dem Import des Moduls aufrufen:

Listing 11.11: 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[3] aus. Das ist doch einfach schön, außerdem schön einfach und es freut den Lernenden, weil er etwas wiedererkennt.

11.3. Pakete

Ein Paket ist ein Verzeichnis, in dem verschiedene Module zusammengefasst sind. Damit Python das als Paket erkennt, muss das Verzeichnis eine (eventuell leere) Datei `__init__.py` enthalten.

Wenn ich also ein Verzeichnis `meinPaket` anlege, das die Module `SuperModul` und `SuperMario` enthält, binde ich einen dieser Module durch `from meinPaket import SuperMario` oder durch `import meinPaket.SuperMario` ein.

[3] die man mit man(Befehl) aufruft

Teil VII.

Objektorientierte Programmierung

12. Klassen

12.1. 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. Das ist einer der großen Vorteile von Python: diese Sprache ist objektorientiert, aber man muss nicht objektorientiert programmieren, um sie zu benutzen. Viele kleine Programme, zum Beispiel Werkzeuge, die man auf seinem eigenen Rechner benutzt, schreibt ein geübter Programmierer in Python in ein paar Zeilen, ohne die Objektorientierung auszupacken.

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.

Damit kann man objektorientierte Programmierung so beschreiben: Attribute-Werte beschreiben den Zustand eines Objekts, die Methoden verändern Attribute. Jedes Objekt behandelt die eigenen Daten selbst und ist damit verantwortlich für seinen Zustand. Wie jedes Objekt diese Behandlung richtig macht, weiß das Objekt, weil das Verhalten in der Klasse beschrieben ist.

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, schlechte 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

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

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, schlechte Programmegeschrieben 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. Als nächstes benötigt eine Klasse einen Namen. Während es in der Umgangssprache „die Klasse der Autos“ heißt, ist es in der (Python-)Programmierung üblich, dass der Name einer Klasse ein Substantiv im Singular ist. Hier darf **und soll** zum ersten Mal ein Großbuchstabe am Anfang eines Namens verwendet werden: **Klassennamen beginnen mit einem Großbuchstaben.** Die erste Klasse ist eine Erweiterung des Grusses, den man am Anfang jeder Programmiertätigkeit an die Welt schicken sollte, das „Hallo, Welt“ . Jetzt wird es aber in der Form aufgebohrt, dass eine Person einen Vornamen hat (das soll das einzige Attribut der Person sein) und die Welt höflich grüßen kann, indem sie auch den eigenen Namen nennt (das wird die einzige Methode der Person).

Listing 12.1: Klasse `Person`

```
1 class Person:
2     def __init__(self, vorname):
3         self.vorname = vorname
4 
5     def gruessen(self):
6         print('Ein freundliches "Hallo Welt" von '+self.vorname)
```

Nach dem Schlüsselwort `class` steht hier der Klassenname `Person`.

Es folgt eine besondere Methode, der Konstruktor[2] der Klasse. Der Konstruktor erstellt nach dem Abbild der Klasse spezielle Instanzen, auch Objekte genannt. In Python heißt er immer `__init__`. (Das sind jeweils 2 Unterstriche vorne und hinten.) Innerhalb des Konstruktors wird der übergebene Vorname (siehe gleich den nächsten Absatz) an die jeweilige Instanz der Klasse übergeben.

[2] Genau genommen ist das kein Konstruktor. Wer dazu mehr wissen will, sollte die Dokumentation zu Python nachlesen. Aber die hier beschriebene Methode kommt einem Konstruktor sehr nahe. Also belassen wir es bei dieser Bezeichnung.

Auf die jeweilige Instanz wird immer mit dem Namen `self` zugegriffen.[3] Jede Methode einer Klasse muss als ersten Parameter diesen Verweis auf das jeweilige Objekt enthalten. Die Methode `gruessen` sieht von der Schreibweise wie eine Funktion aus. Was sie macht ist hoffentlich klar!

Wer konstruieren kann, sollte auch zerstören können. Wer mit anderen objektorientierten Programmiersprachen gearbeitet hat, weiß, dass es dort auch einen „Destruktor" gibt. Er vernichtet alles, nachdem ein Objekt aufgehört hat zu existieren. In Python gibt es diesen Destruktor auch, aber man braucht ihn (in 99 Prozent der Fälle) nicht; es kann sogar unangenehm werden, wenn man ihn fälschlich benutzt. Deswegen wird er hier auch nicht aufgeführt, und falls jemand doch meint, ihn benützen zu müssen: nachschlagen!

Jetzt sollen tatsächlich ein paar Objekte nach diesem Muster erstellt werden: Objekte sind in diesem Fall echte Personen. Los gehts:

Listing 12.2: Objekte der Klasse Person können was!

```
1 >>> ich = Person('Martin')
2 >>> ich.gruessen()
3 Ein freundliches "Hallo Welt" von Martin
4 >>> hannah = Person('Hannah')
5 >>> hannah.gruessen()
6 Ein freundliches "Hallo Welt" von Hannah
```

Ja, auch mir tut das weh, dass da als Anweisung „ich.gruessen()" steht, „ich.gruesse()" wäre schöner. Aber auch hier gilt eine Vereinbarung: so wie Klassennamen immer ein Substantiv im Nominativ singular sein sollen, sollen Methodennamen immer ein Verb im Infinitiv sein. Zuerst wird eine Person `ich` angelegt. Die benötigt als Attribut einen Vornamen, und der wird als Parameter beim Aufruf der Klasse `Person` mitgegeben. Innerhalb des Konstruktors `__init__` wird der Wert dieses Parameters an die Instanz übergeben, die für das jeweilige Objekt immer mit `self` angesprochen wird. Die einzige Methode der Klasse wird auch an jedes Objekt weitergegeben, so dass jetzt `ich` es beherrscht, die Welt zu `gruessen`. Das geschieht dadurch, dass der Methodenname durch einen Punkt an den Objektnamen angehängt wird.

Das zweite Beispiel soll ein bißchen mehr sein und können (sein: das sind die Attribute; können: das sind die Methoden) Bauen wir also die Klasse `Angestellter`, die als Attribute den Namen, das Alter, das Einkommen und die Berufsbezeichnung hat.

Listing 12.3: Klasse `Angestellter`

```
1  >>> class Angestellter:
2          def __init__(self, name, alter, einkommen, berufsbezeichnung):
3              self.name = name
4              self.alter = alter
5              self.einkommen = einkommen
6              self.berufsbezeichnung = berufsbezeichnung
```

[3] `self` entspricht dem `this`in den Sprachen C und Java. Anders als in diesen Sprachen ist aber `self` kein reserviertes Wort in Python. Das bedeutet, dass man durchaus einen anderen Namen dafür benutzen könnte. Aber das sollte man auf keinen Fall machen. `self` ist zwar nur eine Empfehlung, aber eine sehr massive!! Also eher: Du willst doch nicht etwa statt `self` etwas anderes nehmen??!!!??!!!

```
>>> 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

Listing 12.4: Klassendefinition

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

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

Listing 12.5: 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 - ganz deutlich im 21. Jahrhundert - 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`, wie es oben beschrieben wurde, so aus:

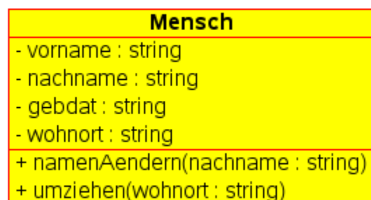


Abbildung 12.1.: Klassendiagramm `Mensch`

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. Das oben stehende 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 auf Seite 252 den

Begriff 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.

12.2. 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.[4] Und machen wir es Schritt für Schritt, wobei wir nach jedem Schritt testen werden, ob alles richtig funktioniert.

Listing 12.6: Eine erste Klasse

```
1  class Mensch():
2    def __init__(self, vorname, nachname, gebdat, wohnort):
3      self.vorname = vorname
4      self.nachname = nachname
5      self.gebdat = gebdat
6      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**

[4] 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.

  - `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.

Listing 12.7: Aufruf der Klasse (Erzeugung eines Objektes)

```
1  #!/usr/bin/python
2  from clMensch import Mensch
3
4  ich = Mensch('Martin', 'Schimmels', '01.01.1988', '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. 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:

Listing 12.8: Erzeugung eines Objekts

```
1  #!/usr/bin/python
2  from clMensch import Mensch
3
4  ich = Mensch('Martin', 'Schimmels', '01.01.1988', 'Rottenburg')
5  print(ich.vorname())
6  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 Abschnitt 12.3 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:

Listing 12.9: 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!

Listing 12.10: Aufruf der Methode anzeigen

```
#!/usr/bin/python
from clMensch import Mensch

ich = Mensch('Martin', 'Schimmels', '01.01.1988', 'Rottenburg')
ich.anzeigen()
```

Wie angekündigt, Schritt für Schritt, wird das Programm unserem Modell angepasst. Die Klasse braucht eine Methode `nachnamenAendern`.

Listing 12.11: 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 der Klasse sieht.

Listing 12.12: Aufruf der neuen Methode

```
#!/usr/bin/python
from clMensch import Mensch

ich = Mensch('Martin', 'Schimmels', '01.01.1988', '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`:

Listing 12.13: 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:

Listing 12.14: Aufruf der neuen Methode

```
#!/usr/bin/python
from clMensch import Mensch

ich = Mensch('Martin', 'Schimmels', '01.01.1988', 'Rottenburg')
ich.anzeigen()
ich.nachnamenAendern('Meier')
ich.anzeigen()

ich.umziehen('Ammerbuch')
ich.anzeigen()
```

12.2.1. Objektvariable und Klassenvariable

Bis hierher wurden nur einem Objekt Attribute zugeordnet; die nennt man auch Objektvariable. Es kann aber auch sinnvoll sein, dass man einer Klasse eine Variable zuordnet; das sind die Klassenvariablen.

Das soll an einem einfachen Beispiel gezeigt werden. Eine Klasse Fahrrad wird beschrieben. Die Fahrräder haben vorläufig nur eine interne Nummer, die beim Anlegen eines Fahrrads eingegeben werden muss. Das ist also eine Objektvariable. Gleichzeitig soll aber mitgezählt werden, wieviele Fahrräder schon angelegt wurden. Das ist eine Klassenvariable.

Hier kommt die Klassendatei des Fahrrads:

Listing 12.15: Klasse Fahrrad mit Klassenvariable

```
1 class Fahrrad():
2     # Klassenvariable
3     anzahl = 0
4
5     def __init__(self):
6         self.lfdNr = input('Nr. des Fahrrads eingeben: ')
7         Fahrrad.anzahl += 1
```

Es folgt die Aufruf-Datei:

Listing 12.16: Aufruf Fahrrad mit Klassenvariable

```
1 for i in range(4):
2     f = Fahrrad()
3     print('jetzt gibt es '+str(Fahrrad.anzahl)+' Fahrräder')
```

Und die Ausgabe des Programms:

```
1 >>> Nr. des Fahrrads eingeben: 1
2 Rahmenhöhe in cm eingeben: 54
3 jetzt gibt es 1 Fahrräder
4 Nr. des Fahrrads eingeben: 2
5 Rahmenhöhe in cm eingeben: 56
6 jetzt gibt es 2 Fahrräder
7 Nr. des Fahrrads eingeben: 3
8 Rahmenhöhe in cm eingeben: 58
9 jetzt gibt es 3 Fahrräder
10 Nr. des Fahrrads eingeben: 4
11 Rahmenhöhe in cm eingeben: 60
12 jetzt gibt es 4 Fahrräder
```

12.3. 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:

Listing 12.17: Keine Geheimnisse (leider)

```
1  #!/usr/bin/python
2  from clMensch import Mensch
3
4  ich = Mensch('Martin', 'Schimmels', '01.01.1988', 'Rottenburg')
5  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.

Stufe	Beispiel
public	vorname
protected	_vorname
private	__vorname

Tabelle 12.1.: Sichtbarkeit und Kapselung

Also sollte man dieses Prögrammchen noch richtig machen (richtig im Sinne der Objektorientierung). Die Attribute werden „privat“ gemacht, so dass nur noch über die Methoden zugegriffen werden kann.

Listing 12.18: Private Attribute

```
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
```

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.

Listing 12.19: Aufruf mit Fehler

```
#!/usr/bin/python
from clMensch import Mensch

ich = Mensch('Martin', 'Schimmels', '01.01.1988', 'Rottenburg')
ich.anzeigen()
print(ich.__nachname)       # Das geht schief! Wenn diese Zeile
                            # entfernt wird, läuft das Programm
                            # natürlich einwandfrei.

ich.nachnamenAendern('Meier')
ich.anzeigen()

ich.umziehen('Ammerbuch')
ich.anzeigen()
```

12.4. Setter und Getter? Oder doch lieber nicht?

In anderen objektorientierten Programmiersprachen ist es gute Sitte, für jedes Attribut zwei in der Regel sehr kurze Methoden zu schreiben:

1. den Getter, meistens `get_attribut` genannt, der nichts anderes macht, als den Wert des Attributs zu lesen und zurückzugeben
2. den Setter, meistens `set_attribut` genannt, der nichts anderes macht, als dem Attribut einen neuen Wert zuzuweisen.

Python-Programmierer sind da nicht so strikt. Aber es soll nicht verschwiegen werden, dass das auch in Python viele Programmierer realisieren.

Zuerst einmal schreibe ich die Methoden und rufe eine davon direkt auf.

Listing 12.20: Getter und Setter (einfach)

```

class Hund():
    def __init__(self, rasse, name):
        self.set_rasse(rasse)
        self.set_name(name)

    def get_rasse(self):
        return self.rasse

    def set_rasse(self,  rasse):
        self.rasse = rasse

    def get_name(self):
        return self.name

    def set_name(self,  name):
        self.name = name


if __name__ == '__main__':
    w = Hund('Dackel',  'Waldi')
    h = Hund('Schäferhund', 'Hasso')
    print(w.get_name())
    print(h.get_name())
```

Das liefert die erwartete Ausgabe:

Listing 12.21: Ausgabe von Getter und Setter (einfach

```
Waldi
Hasso
```

Python-Programmierer machen das aber mit Eigenschaften, das heißt über das Schlüsselwort `property` werden den jeweiligen Attributen Methoden zugeordnet.

Listing 12.22: Getter und Setter (mittels property)

```

#!/usr/bin/python

class Hund():
    def __init__(self, rasse, name):
        self.unsichtbare_rasse = rasse
        self.unsichtbarer_name = name

    def get_rasse(self):
        return self.unsichtbare_rasse

    def set_rasse(self, rasse):
        self.unsichtbare_rasse = rasse

    def get_name(self):
        return self.unsichtbarer_name

    def set_name(self, name):
        self.unsichtbarer_name = name

    name = property(get_name, set_name)
    rasse = property(get_rasse, set_rasse)

if __name__ == '__main__':
    w = Hund('Dackel', 'Waldi')
    h = Hund('Schäferhund', 'Hasso')
    print(w.name)
    print(h.name)
    h.name = 'Titan'
    print('Hasso heißt jetzt '+h.name)
```

Wird jetzt einem Attribut ein Wert zugewiesen, wird der Setter des Attributs aufgerufen; der Getter wird aufgerufen, wenn der Name des Attributs abgefragt wird. Das liefert die erwartete Ausgabe:

Listing 12.23: Ausgabe von Getter und Setter mit Hilfe von property

```
Waldi
Hasso
Hasso heißt jetzt Titan
```

Das ist jetzt besonders wichtig, wenn wir uns wieder der Sichtbarkeit von Attributen zuwenden. Erinnerst Du Dich noch? Sonst lies noch mal nach! Hier kommt also die obige Klasse, nur mit der Veränderung, dass jetzt beide Attribute durch vorangestellte doppelte Unterstriche `privat` geworden sind.

Listing 12.24: Getter und Setter mit Hilfe von property und privater Attribute

```

#!/usr/bin/python

class Hund():
    def __init__(self, rasse, name):
        self.__unsichtbare_rasse = rasse
        self.__unsichtbarer_name = name

    def get_rasse(self):
        return self.__unsichtbare_rasse

    def set_rasse(self, rasse):
        self.__unsichtbare_rasse = rasse

    def get_name(self):
        return self.__unsichtbarer_name

    def set_name(self, name):
        self.__unsichtbarer_name = name

    name = property(get_name, set_name)
    rasse = property(get_rasse, set_rasse)

if __name__ == '__main__':
    w = Hund('Dackel', 'Waldi')
    h = Hund('Schäferhund', 'Hasso')
    print(w.name)
    print(h.name)
    h.name = 'Titan'
    print('Hasso heißt jetzt '+h.name)
    print('Das nächste funktioniert nicht,
        \ndenn "__unsichtbarer_name" ist wirklich unsichtbar')
    print('Hasso heißt jetzt '+h.__unsichtbarer_name)
```

Und hier folgt die Ausgabe des Programms:

Listing 12.25: Ausgabe mit Hilfe von property und privaten Attributen

```
Waldi
Hasso
Hasso heißt jetzt Titan
    # Das nächste funktioniert nicht,
    # denn "__unsichtbarer_name" ist wirklich unsichtbar
Traceback (most recent call last):
  File "./cl_HundPropertySichtbarkeit.py", line 33, in <module>
    print('Hasso heißt jetzt '+h.__unsichtbarer_name)
AttributeError: 'Hund' object has no attribute '__unsichtbarer_name'
```

12.5. Statische Methoden

Manchmal ist es wünschenswert, dass nicht nur bestimmte Objekte eine bestimmte Fähigkeit in einer bestimmten Ausprägung haben. Vielmehr sollen alle Objekte dieselbe Fähigkeit haben, und die Klasse als solche auch. Das nennt man statische Methode. Ein Beispiel verdeutlicht das.

Listing 12.26: Statische Methode

```

>>> class Mensch(object):
    def wohnen():
        print('jeder Mensch wohnt irgendwo')
    wohnen = staticmethod(wohnen)
    def __init__(self, vn, nn):
        self.vn = vn
        self.nn = nn
    def anzeigen(self):
        print(self.vn, self.nn)
        self.wohnen()


>>> Mensch.wohnen()
jeder Mensch wohnt irgendwo
>>> x = Mensch('Martin', 'Schimmels')
>>> x.anzeigen()
Martin Schimmels
jeder Mensch wohnt irgendwo
>>>
```

In Zeile 6 wird die Methode **wohnen** zu einer statischen Methode gemacht. In Zeile 15 sieht man, dass der Aufruf der Methode als Methode der Klasse funktioniert. Nachdem in Zeile 17 ein Objekt der Klasse erzeugt wurde, wird in der darauffolgenden Zeile die Methode **wohnen** des Objekts aufgerufen, und auch das funktioniert.

Die oben angegebene Schreibweise ist anschaulich und verständlich. Mittels „Dekoratoren“ kann die Schreibweise verkürzt werden:

Listing 12.27: Statische Methode

```
>>> class Mensch(object):
    @staticmethod
    def wohnen():
        print('jeder Mensch wohnt irgendwo')
    def __init__(self, vn, nn):
        self.vn = vn
        self.nn = nn
    def anzeigen(self):
        print(self.vn, self.nn)
        self.wohnen()

>>> Mensch.wohnen()
jeder Mensch wohnt irgendwo
>>> x = Mensch('Hannah', 'Schimmels')
>>> x.anzeigen()
Hannah Schimmels
jeder Mensch wohnt irgendwo
```

12.6. 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:

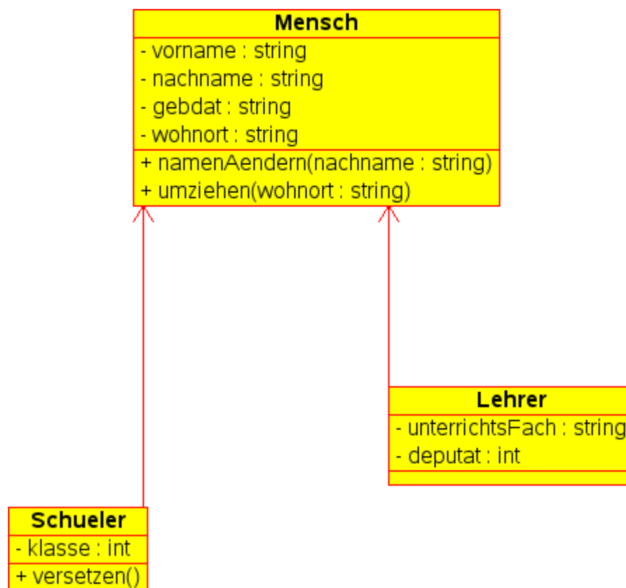


Abbildung 12.2.: Klassendiagramm `Schueler`

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:

Listing 12.28: Abgeleitete Klasse

```
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

class Schueler(Mensch):
    def __init__(self, vorname, nachname, gebdat, wohnort, klasse):
        Mensch.__init__(self, vorname, nachname, gebdat, wohnort)
        self.klasse = klasse

    def versetzen(self):
        self.klasse += 1
```

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.“ Das geschieht durch den ausdrücklichen Aufruf des Konstruktors der Oberklasse: `Mensch.__init__(...)`. Einzig die (Schul-)Klasse muss hier in der Unterklasse `Schueler` 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:

Listing 12.29: Aufruf der abgeleiteten Klasse

```
#!/usr/bin/python
from clMensch import Mensch, Schueler

ich = Mensch('Martin', 'Schimmels', '01.01.1988', 'Rottenburg')

ich.anzeigen()
ich.nachnamenAendern('Meier')
ich.anzeigen()
###
# ab hier gibt es was neues
###
thomas = Schueler('Thomas', 'Renz', '12.11.1989', 'Dettenhausen', 11)

thomas.anzeigen()

thomas.versetzen()
thomas.anzeigen()
```

12.6.1. Klassen neuen Stils

Mit dem Übergang zu Version 2.2 von Python hat Guido van Rossum „neue Klassen“ eingeführt. Um eine stringentere Ordnung zu erhalten, gibt es jetzt auch in Python eine Basis-Klasse, die Klasse `object` (leider ist der Klassenname wirklich kleingeschrieben!!). Um eine Klasse neuen Stils zu erstellen, muss also die Klasse, die wir selber als Basisklasse unserer Anwendung gebaut haben, von `object` erben.

Mit diesen Klassen neuen Stils ist es auch einfacher möglich, sich auf die Methoden der Elternklassen zu beziehen. Im folgenden Beispiel wird eine Vererbung über 2 Generationen gezeigt, bei der die Kind-Generation von der Eltern-Generation und diese von der Großeltern-Generation erbt. Jede der drei Generationen hat eine Methode `anzeigen`, und so sieht das aus, wenn das benutzt wird.

Listing 12.30: Drei Ebenen von Klassen im neuen Stil

```
1  class Ebene1(object):
2      def anzeigen(self):
3          print('das ist Ebene 1, die Grosseltern-Ebene')
4
5  class Ebene2(Ebene1):
6      def anzeigen(self):
7          print('das ist Ebene 2, die Eltern-Ebene')
8
9  class Ebene3(Ebene2):
10     def anzeigen(self):
11         print('das ist Ebene 3')
12         print('aber ich kenne auch meine Eltern und Grosseltern')
13         super(Ebene3, self).anzeigen()
14         super(Ebene2, self).anzeigen()
15
16 >>> eineEbene = Ebene3()
17 >>> eineEbene.anzeigen()
18 das ist Ebene 3
19 aber ich kenne auch meine Eltern und Grosseltern
20 das ist Ebene 2, die Eltern-Ebene
21 das ist Ebene 1, die Grosseltern-Ebene
```

Die Methode `anzeigen` der Klasse `Ebene3` kann einfacher geschrieben werden, da an der Stelle des Aufrufs klar ist, was die „super“ -Ebene ist:

Listing 12.31: Vereinfachte Methode bei eindeutiger Superklasse

```
1  class Ebene3(Ebene2):
2      def anzeigen(self):
3          print('das ist Ebene 3')
4          print('aber ich kenne auch meine Eltern und Grosseltern.
5                  SCHAUT!!!')
6          super().anzeigen()
7          super(Ebene2, self).anzeigen()
```

12.6.2. Weiter mit der Klasse `Mensch`

Wenn wir jetzt zurückgehen zum Beispiel aus dem vorigen Kapitel, sehen wir dass die Klassenbeschreibung verändert wird:

Listing 12.32: Klasse Mensch im new style

```
class Mensch(object):
    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
```

In Bezug auf Vererbung ändert sich auch etwas: der Aufruf der Methoden der Oberklasse soll nicht mehr durch das Voranstellen des Oberklassen-Namens geschehen, sondern durch die Benutzung der Funktion `super`. `super` bekommt immer 2 Parameter: der erste Parameter ist der Name der (aktuellen) Unterklasse, der zweite Parameter ein Verweis auf das aktuelle Objekt, also `self`. Das hört sich kompliziert an, vereinfacht aber das Vorgehen wieder erheblich; es bedeutet nämlich umgangssprachlich formuliert wieder „Hey, Objekt, Du weißt doch, von was Du erbst, also ruf mal das oder das von Deiner Oberklasse auf“ . Die Intelligenz steckt also wieder in der Klasse.

Aus dem vorigen Kapitel wird hier der Aufruf des Konstruktors der Oberklasse neu formuliert. Zusätzlich bekommt die Unterklasse aber eine Methode `anzeigen`, um zu demonstrieren, dass auch eine beliebige Methode mit Hilfe von `super` aufgerufen werden kann. Das Beispiel aus dem vorigen Kapitel mit der abgeleiteten Klasse `Schueler` verändert sich zu:

Listing 12.33: Abgeleitete Klasse mit Super-Aufruf (new style)

```
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

class Schueler(Mensch):
    def __init__(self, vorname, nachname, gebdat, wohnort, klasse):
        super(Schueler, self).__init__(vorname, nachname, gebdat, wohnort)
        self.klasse = klasse

    def versetzen(self):
        self.klasse += 1

    def anzeigen(self):
        super(Schueler, self).anzeigen()
        print('in Klasse:', self.klasse)
```

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.

12.7. Methoden überschreiben

12.7.1. Selbstgeschriebene Methoden

Das nächste Element der Objektorientierung, das die Arbeit erleichtert, ist das Überschreiben von Methoden. Darunter versteht man, dass eine abgeleitete Klasse von ihrer Superklasse nicht mehr alle Methoden erbt, sondern dass manche Methoden vererbt werden, andere in der abgeleiteten Klasse neu geschrieben werden. Das soll an einem einfachen Beispiel gezeigt werden. Personen können in diesem Beispiel entweder Männer oder Frauen sein, und diese Personen können verschieden angeredet werden: eine Person,

die nicht näher beschrieben wird, wird ganz formlos mit einem freundlichen „Hallo“ und ihrem Namen angeredet, eine Frau und ein Mann mit einem höflichen „Guten Tag“ und einem „Frau“ bzw. „Herr“ vor ihrem Namen.

Nach so vielen Informationen über Objektorientierung in Python verstehen die Meisten den Quelltext des Beispiels beim Durchlesen, oder?

Listing 12.34: Überschreiben der Methode anreden

```
#!/usr/bin/python

class Person():
    def __init__(self, name, anrede):
        self.name = name
        self.anrede = anrede
    def namenAusgeben(self):
        return self.name
    def anreden(self):
        return self.anrede +', '+self.name

class Herr(Person):
    def anreden(self):
        return self.anrede+', Herr '+self.name


class Frau(Person):
    def anreden(self):
        return self.anrede+', Frau '+self.name

m = Person('Martin', 'Hi hello')
print(m.anreden())
d = Frau('Schreiner', 'Einen wunderschönen guten Tag')
print(d.anreden())
d = Herr('Maler', 'Guten Tag')
print(d.anreden())
```

Listing 12.35: Ausgabe des Beispiels mit verschiedenen anreden-Methoden

```
Hi hello, Martin
Einen wunderschönen guten Tag, Frau Schreiner
Guten Tag, Herr Maler
```

Jede der 3 Klassen hat eine Methode `anzeigen`. Die Methode einer der beiden spezialisierten Klassen `Herr` bzw. `Frau` überschreiben die allgemeinere Version. Das bedeutet, dass die speziellere Version von `anzeigen` benutzt wird, wenn ein Objekt der spezielleren Klasse angelegt wird und diese Methode aufgerufen wird.

12.7.2. „Magische“ Methoden

Spezielle Methoden, nämlich Methoden, die zu den grundlegenden Klassen wie zum Beispiel der Klasse der ganzen Zahlen oder der Klasse der Zeichenketten gehören, können

auch überschrieben werden. Diese magischen Methoden beginnen immer mit zwei Unterstrichen und enden mit zwei Unterstrichen. Die Vergleichsoperationen etwa gelten für Zahlen, aber auch für Zeichenketten. Während es klar ist, was die Frage, ob eine Zahl größer als eine andere ist, bedeutet, muss man sich zuerst mal daran gewöhnen, dass die Frage, ob eine Zeichenkette größer als eine andere ist, beantwortet wird mit der lexikalischen Ordnung der Zeichenketten.

Listing 12.36: Vergleich von 2 Zeichenketten

```
>>> 'Martina' > 'Martin'
True
>>> 'doof' > 'dof'
True
>>> 'Doof' > 'Doofie'
False
```

Also schaue ich zuerst nach, was Zeichenketten denn alles können:

Listing 12.37: Inhalt von `str`

```
>>> dir(str)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__','__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__','__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__','__iter__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__','__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__','capitalize',
'casefold', 'center', 'count', 'encode', 'endswith','expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust',
'lower','lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust','rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip','swapcase', 'title',
'translate', 'upper', 'zfill']
```

Was ist also zu tun, wenn ich von Zeichenketten erwarte, dass eine Zeichenkette größer ist als die andere, wenn sie länger ist? Ganz einfach: ich baue mir eine eigene Klasse für Zeichenketten und überschreibe die größer-Funktion.

Listing 12.38: eigener Vergleich von Zeichenketten

```
>>> class MeinStr(object):
        def __init__(self,text):
            self.t = text
        def __gt__(self,t2):
            return len(self.t) > len(t2)
>>> ms = MeinStr('abcde')
>>> ms > 'xyz'
True
>>> ms > 'xyzabcde'
False
>>> ms = MeinStr('A')
>>> ms > 'a'
False
>>> ms > ''
True
>>> ms > '  '
False
```

Als nächstes folgt ein Beispiel aus der Elementarmathematik. Ganze Zahlen (wie auch rationale Zahlen) können addiert werden. Dafür gibt es die Operation +. Das war ja eine der ersten Aktionen, die in diesem Skript angesprochen wurden:

```
>>> 3+4
7
```

Aber rationale Zahlen werden ja (oft, sinnvollerweise) als Brüche geschrieben. Da ist der Algorithmus für die Addition ja nicht der selbe wie bei der Addition von natürlichen oder ganzen Zahlen:

1. größten gemeinsamen Teiler der beiden Nenner suchen; das ist der Hauptnenner
2. beide Brüche durch eventuelles Erweitern auf den Hauptnenner bringen
3. die beiden Zähler addieren

Das kann auch in einer anderen Reihenfolge gemacht werden, und das ist für die Programmierung etwas einfacher.

1. Zähler des ersten Bruchs mit dem Nenner des zweiten Bruchs multiplizieren, dazu das Produkt von Zähler des zweiten Bruchs mit Nenner des ersten Bruchs addieren
2. dieses Produkt ist der neue Zähler
3. die beiden Nenner miteinander multiplizieren
4. dieses Produkt ist der neue Nenner
5. dieser neue Bruch ist noch nicht gekürzt

Diese zweite Reihenfolge ist in dem unten dargestellten Programmschnipsel gezeigt. Die Methode `kuerzen` sollte jeder selbst programmieren und wird hier nicht gezeigt.

Wichtig ist hier, dass die Methode „addieren" der anderen Zahlen überschrieben wird. Diese Methode ist die magische Methode `__add__` (wieder 2 Unterstriche vor und nach dem Wort „add"). Wenn ich jetzt in meiner Klasse `Bruch` eine neue Methode `__add__` codiere, dann kann ich zwei Objekte der Klasse `Bruch` einfach durch ein einfaches Pluszeichen addieren.

Listing 12.39: Ein Schnipsel der Klasse Bruch

```

class Bruch():
    def __init__(self, zaehler=0, nenner=1):
        self.zaehler = zaehler
        self.nenner = nenner

    def __add__(self, b2):
        self.zaehler = self.zaehler * b2.nenner + b2.zaehler * self.nenner
        self.nenner = self.nenner * b2.nenner
        self.kuerzen()
        return Bruch(self.zaehler, self.nenner)
```

Das wird jetzt so aufgerufen:

Listing 12.40: Aufruf: 2 Brüche addieren

```
z1 = int(input('Zähler eingeben (ganze Zahl): '))
n1 = int(input('Nenner eingeben (ganze Zahl): '))
b1 = Bruch(z1, n1) # wird verändert

z2 = int(input('Zähler eingeben (ganze Zahl): '))
n2 = int(input('Nenner eingeben (ganze Zahl): '))
b2 = Bruch(z2, n2)

erg = b1+b2
```

Als erstes sollte man jetzt eine Methode `kuerzen` schreiben; dazu benötigt man noch den Euklid'schen Algorithmus für die Suche des größten gemeinsamen Teilers.

Dann drängt es sich geradezu auf, dass man eine eigene Methode `anzeigen` für Brüche codiert. Und dann wird jeder wahrscheinlich die anderen Methoden für die Subtraktion, Multiplikation und Division von Brüchen schreiben wollen!

Hier kommt noch eine Liste der magischen Methoden, die überschrieben werden können:

- Vergleichs-Methoden
  - `__eq__` : gleich
  - `__ne__` : ungleich
  - `__gt__` : größer
  - `__ge__` : größer oder gleich
  - `__lt__` : kleiner
  - `__le__` : kleiner oder gleich
- mathematische Methoden
  - `__add__` : addieren (auch für Strings!)
  - `__sub__` : subtrahieren
  - `__mul__` : multiplizieren
  - `__truediv__` : dividieren („echtes“ , d.h. 7 / 4 = 1.75)
  - `__floordiv__` : Ganzzahldivision (d.h. 7 / 4 = 1)
  - `__mod__` : Modulus (Rest bei der Ganzzahldivision, d.h. 7 % 4 = 3)
  - `__pow__` : potenzieren (3**4 = 81)
- sonstige
  - `__str__` : gibt die Zeichenkette zurück
  - `__len__` : gibt die Länge des Objektes zurück

12.8. Gute Programme . . . wann schreibt man objektorientiert?

Schon mehrmals in diesem Text wurde das Thema Gute Programme, schlechte Programme angesprochen. Was heißt „gute Programme“ im Zusammenhang mit Objektorientierung? Und wann sollte man ein objektorientiertes Programm schreiben? Das ist eine Frage, die ich mir im Laufe der letzten Jahre immer wieder gestellt habe. Als Programmierer bin ich auch nicht „objektorientiert auf die Welt gekommen“ . Und viele Programme, die ich für den Hausgebrauch geschrieben habe, vor allem kleine Hilfsmittel, die mir die Bedienung des eigenen Computers erleichtern sollten, habe ich über Jahre (eher über Jahrzehnte!) mit der Sprache „perl“ geschrieben. Das ist eine schöne Sprache, in der Tierwelt der Programmiersprachen die „eierlegende Wollmilchsau“ : man kann alles mit ihr machen, vor allem auch jede Schweinerei. Aber leider versteht man nach ein paar Tagen schon nicht mehr, was man da alles in perl geschrieben hat. Perl ist von Natur aus nicht objektorientiert.

Nachdem ich Python kennengelernt habe, habe ich begonnen, auch solche Tools für den Hausgebrauch in Python zu schreiben. Wahrscheinlich gehen viele Programmierer so vor: wenn ich von einer neuen Sprache höre, überlege ich, wie ich ein Programm, das ich in einer anderen Sprache geschrieben habe, in der neuen Sprache realisieren würde: Programme, die mit Dateien lesen, schreiben oder verändern; solche, die ganze Dateisysteme bearbeiten; Programme, die auf Datenbanken zugreifen; Programme, die irgendetwas im Internet machen; und vieles mehr.

Auf einmal wurden meine Hilfsmittel verständlich. Und mit der Benutzung von Python wurde es mir auch klar, warum manchmal Objektorientierung zum besseren Verständnis von Programmen beiträgt. Ein paar dieser Gedanken sollen hier festgehalten werden.

1. Das, was ich selber oft nicht mache, wenn ich „mal schnell" eine Lösung suche, sollte bei umfangreicheren Programmen am Anfang stehen: ein gutes Design. Ich selber male wirklich nicht gerne UML-Diagramme ... aber es hilft!!!
2. Wenn ein Programm zu lang wird, überlege, ob es nicht innerhalb des Programms zusammenhängende „Aktionseinheiten" gibt, die Du zusammenfassen kannst.
3. Das ist oft dann der Fall, wenn ein Ding ganz wenige (oder bestenfalls nur eine) Eigenschaft hat und diese Eigenschaft immer auf eine sehr ähnliche (oder bestenfalls auf die selbe) Art verändert wird. Das hört sich doch sehr nach einer Klasse mit einem (oder ganz wenigen) Attribut(en) und mit einer Methode an.
4. Wenn Du das Gefühl hast, dass zwei Dinge in Deinem Programm sehr ähnlich aussehen und sich sehr ähnlich verhalten, überlege Dir, ob das eine Ding nicht eine Spezialisierung (oder in der anderen Richtung eine Verallgemeinerung) des anderen Dings ist. Wenn Du diese Überlegung total oder ansatzweise mit „ja" beantworten kannst, spricht vieles dafür, dass Du es hier mit Vererbung zu tun hast.
5. Wenn Du Dich für Objektorientierung entschieden hast, und dann feststellst, dass Deine Klassendatei sehr groß geworden ist, überleg Dir noch einmal, ob das alles, was in der Klassendatei steht, wirklich zu EINER Klasse gehört oder ob nicht noch eine weitere Klasse mitspielt.
6. Das ist vor allem dann wahrscheinlich, wenn innerhalb der Klassendatei auf Daten zugegriffen wird, die keine Attribute der Klasse sind.
7. Schau Dir nochmal alle Bezeichner durch: sind Attributnamen wirklich Substantive und Methodennamen wirklich Verben im Infinitiv? Mach das auch in die andere Richtung: suche alle Verben und überprüfe, ob das wirklich alles Methoden sind; suche alle Substantive und checke, ob das alles Attribute sind. Wenn nicht, ändere das schnell. Und wenn nach der Änderung sich irgendetwas eher seltsam anhört: dann ist wahrscheinlich irgendetwas eher faul!!!

13. Ein etwas längeres Programm

13.1. 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.

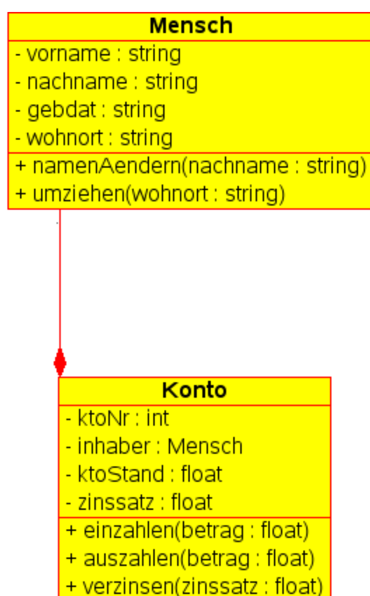


Abbildung 13.1.: Konten

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.

13.2. Objektorientierte Programmierung

13.2.1. Die Klasse Konto

Die Umsetzung der Klassendefinition in Python sieht so aus:

Listing 13.1: Die Klasse Konto

```
from clMensch import Mensch

class Konto(object):

    def __init__(self, ktoNr, vorname, nachname, gebdat, wohnort,
                 ktoStand, zinssatz):
        self.ktoNr = ktoNr
        self.inhaber = Mensch(vorname, nachname, gebdat, wohnort)
        self.ktoStand = ktoStand
        self.zinssatz = zinssatz

    def einzahlen(self, betrag):
        pass

    def auszahlen(self, betrag):
        pass

    def verzinsen(self, zinssatz):
        pass
```

Das wird wieder in einer eigenen Datei gespeichert, in clKonto.

13.2.2. 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:

Listing 13.2: Aufruf der Klasse Konto

```
#!/usr/bin/python

from clKonto import Konto

meinKonto = Konto('101010', 'Martin', 'Schimmels', '01.01.1988',
                  'Rottenburg', 100.00, 0.02)
print(meinKonto.inhaber.vorname, meinKonto.inhaber.nachname,
                                  '\n',  meinKonto.inhaber.wohnort)
print('Kontonr.:', meinKonto.ktoNr,
        ' Kontostand: ', meinKonto.ktoStand)
```

13.2.3. Realisierung der Methoden „Einzahlen“ und „Auszahlen“

Einzahlungen und Auszahlungen müssen nun programmiert werden. Das sind aber sehr kurze Methoden.

Listing 13.3: Realisierung der ersten Methoden

```
from clMensch import Mensch

class Konto(object):

    def __init__(self, ktoNr, vorname, nachname, gebdat, wohnort,
                        ktoStand, zinssatz):
        self.ktoNr = ktoNr
        self.inhaber = Mensch(vorname, nachname, gebdat, wohnort)
        self.ktoStand = ktoStand
        self.zinssatz = zinssatz

    def einzahlen(self, betrag):
        self.ktoStand += betrag

    def auszahlen(self, betrag):
        self.ktoStand -= betrag

    def verzinsen(self, zinssatz):
        pass
```

Und das Aufruf-Programm ruft jede der beiden Methoden einmal auf und gibt jeweils anschließend den aktuellen Stand aus:

Listing 13.4: Aufruf der beiden neuen Methoden

```
#!/usr/bin/python

from clKonto import Konto

meinKonto = Konto('101010', 'Martin', 'Schimmels', '01.01.1988',
                  'Rottenburg', 100.00, 0.02)
print(meinKonto.inhaber.vorname, meinKonto.inhaber.nachname,
                  '\n', meinKonto.inhaber.wohnort)
print('Kontonr.:', meinKonto.ktoNr, ' Kontostand: ',
                  meinKonto.ktoStand)

meinKonto.einzahlen(111.23)
print(meinKonto.inhaber.vorname, meinKonto.inhaber.nachname, '\n',
                  meinKonto.inhaber.wohnort)
print('Kontonr.:', meinKonto.ktoNr, ' Kontostand: ', meinKonto.ktoStand)

meinKonto.auszahlen(5.01)
print(meinKonto.inhaber.vorname, meinKonto.inhaber.nachname, '\n',
                  meinKonto.inhaber.wohnort)
print('Kontonr.:', meinKonto.ktoNr, ' Kontostand: ', meinKonto.ktoStand)
```

13.2.4. 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.

Listing 13.5: Die Verzinsung

```
from clMensch import Mensch

class Konto(object):
    taZaehler = 0

    def __init__(self, ktoNr, vorname, nachname, gebdat, wohnort,
                        ktoStand, zinssatz):
        self.ktoNr = ktoNr
        self.inhaber = Mensch(vorname, nachname, gebdat, wohnort)
        self.ktoStand = ktoStand
        self.zinssatz = zinssatz

    def einzahlen(self, betrag):
        self.ktoStand += betrag
        self.taZaehler += 1
        if self.taZaehler % 3 == 0:
            self.verzinsen(self.zinssatz)

    def auszahlen(self, betrag):
        self.ktoStand -= betrag
        self.taZaehler += 1
        if self.taZaehler % 3 == 0:
            self.verzinsen(self.zinssatz)

    def verzinsen(self, zinssatz):
        self.ktoStand += self.ktoStand*self.zinssatz
```

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.

13.2.5. 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.)

Listing 13.6: 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!

Listing 13.7: Verbesserungen des Aufruf-Programms

```
#!/usr/bin/python

from clKonto import Konto

meinKonto = Konto('101010', 'Martin', 'Schimmels', '01.01.1988',
                  'Rottenburg', 100.00, 0.02)
meinKonto.kontoStandAusgeben()

meinKonto.einzahlen(111.23)
meinKonto.kontoStandAusgeben()

meinKonto.auszahlen(5.01)
meinKonto.kontoStandAusgeben()

meinKonto.einzahlen(100.00)
meinKonto.kontoStandAusgeben()

meinKonto.einzahlen(100.00)
meinKonto.kontoStandAusgeben()
```

Die zweite Verbesserung ist nur eine Kleinigkeit. Nur die neu geschaffene Methode `kontoStandAusgeben` muss an einer Stelle verändert werden. Siehe dazu im Kapitel 7.3 über Zeichenketten den Abschnitt über Formatierung von Zeichenketten. Der Quelltext der veränderten Klassenmethode sieht so aus:

Listing 13.8: 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!

Listing 13.9: Konten bearbeiten in einer Schleife

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  from clKonto import Konto
4
5  meinKonto = Konto('101010', 'Martin', 'Schimmels', '01.01.1988',
6                                          'Rottenburg', 100.00, 0.02)
7  meinKonto.kontoStandAusgeben()
8
9  def meld():
10     print("Konto-Verwaltung\ne für Einzahlung\na für Auszahlung
11               \ns für Schluss")
12
13 eingabe = 'q'
14 while (eingabe != 's'):
15     meld()
16     eingabe = input('Bitte Eingabe:')
17     if eingabe == 'e':
18         betrag = float(input('Einzahlungsbetrag: '))
19         meinKonto.einzahlen(betrag)
20     elif eingabe == 'a':
21         betrag = float(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()
```

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.

13.2.6. Aufgaben 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.
3. Diese Übungsaufgabe hört sich leicht an, ist aber recht kompliziert. Es geht um das bekannte Spiel unter der Schulbank „Schiffe versenken" . Gegeben ist ein Spielfeld der Größe 10 x 10, auf dem ein Schiff der Länge 4, 2 Schiffe der Länge 3 und 3 Schiffe der Länge 2 versteckt sind. Der Dialog mit dem Spieler könnte so aussehen:

Listing 13.10: Schiffe versenken

```
Zeilen und Spalten sind von 0 bis 9 numeriert
In welche Zeile ..1
und in welche Spalte willst Du zielen?0

0   0   0   0   .   0   .   0   .   0
k   X   k   .   0   0   0   k   0   0
0   0   0   0   0   0   V   V   V   V
0   0   k   0   0   0   0   0   0   0
V   V   V   k   0   0   0   .   .   0
0   0   0   0   0   V   0   0   k   k
0   0   0   0   k   V   k   V   V   V
0   0   0   0   0   k   k   0   0   0
0   0   .   0   .   k   k   0   0   0
0   0   0   0   0   k   X   0   0   0
```

- 0 : noch nicht beschossenes Feld
- . : Splash! Das war voll ins Wasser!
- k : knapp daneben
- X : Treffer
- V : Schiff total versenkt

13.3. Ein klassisches Beispiel: Stack und Queue

Es gibt im Prinzip nur zwei Zugriffsmethoden auf eine Menge von Dingen:

1. FIFO: die Schlange (queue) Jeder Engländer versteht dieses Prinzip (und vielleicht gar kein anderes). Es wird brav sich angestellt, und in der Reihenfolge des Anstellens kommt man dran. So sieht es auch an der Supermarkt-Kasse aus: wer zuerst ankommt, wird auch zuerst bedient. FIFO heißt eben „first in, first out“ .

2. LIFO: der Stapel (stack) Für Frauen ist das wahrscheinlich unverständlich, aber bei Männern funktioniert der Inhalt eines Kleiderschranks nach diesem Prinzip. Der Mann selber (oder die liebe Mama oder die noch liebere Freundin oder Frau) räumt die T-Shirts in den Kleiderschrank ein, und am Morgen langt er hinein und nimmt das oberste, das als letztes reingelegt wurde, heraus und zieht es an. LIFO heißt eben „last in, first out“ .

13.3.1. Die Queue

Nein, da muss man nichts dazu sagen. Höchstens muss man noch einmal nachschlagen, wie man Veränderung von Listen durchführt! Die Klasse `Queue` ist dann selbstverständlich!

Listing 13.11: Klasse Queue

```
class Queue(object):
    def __init__(self):
        self.dieQueue = []

    def hinzufuegen(self, ding):
        self.dieQueue.append(ding)

    def entfernen(self):
        entf = self.dieQueue.pop(0)

    def anzeigen(self):
        print(self.dieQueue)
```

Ein Programm, das diese Queue testet, ist auch nicht viel schwerer:

Listing 13.12: Aufruf Queue

```
from cl_Queue import Queue
meineQueue = Queue()
for i in range(5):
    meineQueue.hinzufuegen(i)
meineQueue.anzeigen()
while meineQueue.dieQueue:
    meineQueue.entfernen()
    meineQueue.anzeigen()
```

Listing 13.13: Ausgabe einer Queue

```
[0, 1, 2, 3, 4]
[1, 2, 3, 4]
[2, 3, 4]
[3, 4]
[4]
[]
```

13.3.2. Der Stack

Hier muss man noch weniger sagen!

Listing 13.14: Klasse Stack

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

    def hinzufuegen(self, ding):
        self.derStack.append(ding)

    def entfernen(self):
        entf = self.derStack.pop()

    def anzeigen(self):
        print(self.derStack)
```

Der Aufruf

Listing 13.15: Aufruf Stack

```
from cl_Stack import Stack

meinStack = Stack()
for i in range(5):
    meinStack.hinzufuegen(i)
meinStack.anzeigen()
while meinStack.derStack:
    meinStack.entfernen()
    meinStack.anzeigen()
```

Listing 13.16: Ausgabe Stack

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 2]
[0, 1]
[0]
[]
```

Teil VIII.

Fehler und Ausnahmen

14. Fehler ...

Because something is happening
here
And you don't know what it is

(Bob Dylan[1])

14.1. ... 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

[1] Ballad of a thin man *auf:* Highway 61 revisited

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, 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.

[2] So hat man das wirklich vor nicht allzu langer Zeit gemacht. Eine extrem lästige Arbeit!

14.2. Fehler werden abgefangen

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:

Listing 14.1: Falscher Alter

```
1  >>> alter = int(input('Alter? '))
2   Alter? ich sage meistens Papa zu ihm
3   Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   File "<string>", line 1
6   ich sage meistens Papa zu ihm
7            ^
8   SyntaxError: invalid syntax
```

Was man sich wünschen würde, wäre aber etwas in der Art:

Listing 14.2: Richtiges Alter

```
1  >>> alter = int(input('Alter? '))
2   Alter? ich sage meistens Papa zu ihm
3     Fehlerhafte Eingabe! Es ist das Alter in Jahren gefragt.
4     Probier es nochmal!
5  >>> alter = int(input('Alter? '))
```

Und man bekommt eine zweite Chance (und zur Not eine dritte und eine vierte . . .)

Der obige Dialog mit dem Benutzer wurde realisiert durch folgendes Programm:

Listing 14.3: Möglichen Fehler abgefangen

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  alterOK = False
4  while not alterOK:
5      try:
6          alter = int(input('Alter? '))
7          alterOK = True
8      except:
9          print('''Fehlerhafte Eingabe!
10                Es ist das Alter in Jahren gefragt.
11                Probier es nochmal!''')
```

Schöner allerdings ist es, den entsprechenden Fehler abzufangen. Das ist ein Fehler der Klasse `ValueError`, also ein Wert-Fehler. Dazu lasse ich einen Text eingeben und versuche, diesen Text in eine Ganzzahl umzuwandeln:

Listing 14.4: Speziellen Fehler abgefangen

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
alterOK = False
while not alterOK:
    try:
        alter = int(input('Alter? '))
        alterOK = True
    except ValueError:
        print('''Fehlerhafte Eingabe!
                 Es ist das Alter in Jahren gefragt.
                 Probier es nochmal!''')
```

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.

In der Frühzeit der Programmierung war einer der häufigsten Fehler, der einen Rechner (zum Beispiel unter DOS) zum Absturz brachte, dass man versuchte, durch Null zu dividieren.

Listing 14.5: Division

```
>>> x = 27
>>> y = 3
>>> ergebnis = x/y
>>> print(ergebnis)
>>> 9
```

So weit, so gut. Beim nächsten Versuch geht es schief:

[3] Diese Sprechweise hat sich auch im Deutschen durchgesetzt: sie ist eine wörtliche Übersetzung von „to throw an exception“

Listing 14.6: Fehler: Division durch 0

```
>>> x = 27
>>> y = 0
>>> erg = x/y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Die Fehlermeldung ist verständlich, aber das Programm hat sich verabschiedet. [4] Man kann das in Python durch ein `try - except` Konstrukt zum Glück verhindern:

Listing 14.7: Fehler: Division durch 0 abgefangen

```
>>> try:
>>>     x = 27
>>>     y = 0
>>>     erg = x / y
>>> except:
>>>     print('Da lief etwas schief. Schau noch mal nach den Werten
            von x und y')
'Da lief etwas schief. Schau noch mal nach den Werten von x und y'
```

Sinnvollerweise sollte man diesen ganzen Programm-Schnipsel in eine `while`-Schleife einbinden, die es dem Benutzer erlaubt, einen Wert für `y` so lange einzugeben, bis der ungleich 0 ist.

Das Otto-Waalkes-Gedächtnis-Beispiel sieht dann in Python so aus:

```
  >>> try:
          alter = int(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.

[4] Zum Glück wurden die Betriebssysteme inzwischen so verbessert, dass nicht mehr das ganze Betriebssystem sich verabschiedet.

Dieses einfache Muster einer Fehlerbehandlung

```
>>> try:
        Anweisung(en)
    except:
        Fehlermeldung
```

kann verbessert werden zu

```
>>> 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.

14.3. Spezielle Fehler

Oft weiß man aber genauer, welche Art von Fehler passieren könnte. Das erste Beispiel von oben mit der Division durch Null könnte so verbessert werden:

Listing 14.8: ZeroDivisionError abgefangen

```
>>> try:
>>>     x = 27
>>>     y = 0
>>>     erg = x / y
>>> except ZeroDivisionError:
>>>     print('y darf nicht 0 sein!!!')
>>> except:
>>>     print('irgendein anderer Fehler ist aufgetaucht')
y darf nicht 0 sein!!!
```

Hier kommt ein weiteres Beispiel, bei dem aus Versehen versucht wird, über das Ende einer Liste hinauszulesen:

Listing 14.9: Listenbearbeitung korrekt

```
>>> eineListe = ['nulltes Element', 'erstes Element', 'zweites Element']
>>> eineListe[1]
'erstes Element'
```

So weit kein Problem. Aber was passiert, wenn man jetzt versucht, das siebte Element auszugeben?

Listing 14.10: Listenbearbeitung mit Fehler

```
>>> 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:

Listing 14.11: IndexError 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 können aber in einem Block verschiedenartige Fehler auftreten. Hier zeige ich es bei unserem Eingangsbeispiel mit der Division. Nicht nur die Division durch 0 führt zu einem Abbruch, sondern auch die Division durch etwas anderes, durch das man nicht dividieren kann:

Listing 14.12: Falscher Datentyp

```
>>> x = 3
>>> y = 'Always look on the bright side of life'
>>> erg = x / y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Durch Text kann man nicht teilen! Es sollen also jetzt 2 Fehlerarten abgefangen werden.

Listing 14.13: Mehrere spezielle Fehler abgefangen

```
>>> try:
>>>     x = 27
>>>     y = 0
>>>     erg = x / y
>>> except ZeroDivisionError:
>>>     print('y darf nicht 0 sein!!!')
>>> except TypeError:
>>>     print('Der Wert einer der beiden Variablen ist keine Zahl')
>>> except:
>>>    print('Irgendein anderer Fehler ist aufgetreten')
y darf nicht 0 sein!!!
```

Und ein nächster Testlauf:

```
>>> try:
>>>     x = 27
>>>     y = 'Always look on the bright side of life'
>>>     erg = x / y
>>> except ZeroDivisionError:
>>>     print('y darf nicht 0 sein!!!')
>>> except TypeError:
>>>     print('Der Wert einer der beiden Variablen ist keine Zahl')
>>> except:
>>>    print('Irgendein anderer Fehler ist aufgetreten')
Der Wert einer der beiden Variablen ist keine Zahl
```

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 sollte man sich den „Exception-Hierarchie“-Baum unter http://docs.python.org/3/library/exceptions.html#exception-hierarchy anschauen. Die Regel bei der Abarbeitung von exceptions ist diese: Python geht im Exception-Hierarchie-Baum so weit in die Äste, bis es eine zutreffende Exception gefunden hat. Im folgenden Beispiel steht fälschlicherweise der `LookupError` vor dem `IndexError`. Wenn Du den Hierarchie-Baum anschaust, siehst Du, dass der `IndexError` ein Teil-Ast des Astes `LookupError` ist. Python ist also mit dem `LookupError` zufrieden ... ich aber nicht!

Listing 14.14: Diese Exception wollte ich nicht!!

```
liste1 = ['a', 'b', 'c', 'd', 'e']

try:
    x = liste1[5]
except LookupError:
    print("Lookup error ist aufgetreten")
except IndexError:
    print("Falscher Schlüssel in der Tabelle")
```

Das Programm liefert die Ausgabe

```
Lookup error ist aufgetreten
```

Listing 14.15: Richtige Reihenfolge der Exceptions!!

```
liste2 = ['a', 'b', 'c', 'd', 'e']

try:
    x = liste2[5]
except IndexError:
    print("Falscher Schlüssel in der Tabelle")
except LookupError:
```

```
8       print("Lookup error ist aufgetreten")
```

Das Programm liefert die Ausgabe

```
1 Falscher Schlüssel in der Tabelle
```

Eine Fehlermeldung „Lookup error ist aufgetreten“ würde erzeugt, wenn genau dieser Fehler oder ein Fehler im Baum unterhalb dieses Astes, aber nicht der `IndexError` aufgetreten wäre.

14.3.1. Ein Menu-Schnipsel mit Fehlerbehandlung

Immer wieder schreibt man ein Programm und freut sich, dass das Programm so läuft, wie man es sich ausgedacht hat. Kaum setzt sich der erste Benutzer an das Programm, bricht es ab: der Benutzer hat bei einer Eingabeaufforderung etwas eingegeben, was uns noch nie in den Sinn kam. Vor allem bei Auswahl-Menus passiert so etwas immer wieder.

Deswegen folgt hier ein Menu, das die Auswahl aus drei Aktionen erlaubt. Die Funktionen tun nichts, außer dass sie sich melden. Das Menu ist in einem Dictionary hinterlegt, wobei der Schlüssel für eine Aktion eine ganze Zahl ist und der Wert der Name der Funktion. Zusätzlich soll der Benutzer durch die Eingbe von „e“ das Programm beenden können. Typische Fehler durch den Benutzer sind

- die Eingabe eines anderen Buchstaben als „e“
- die Eingabe einer Zahl, die nicht als Schlüssel im Dictionary steht

Das Menu-Programm sieht dann so aus[5]:

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  eingabe = ''
5
6  def tu1():
7      print('hier ist TU 1')
8
9  def tu2():
10     print('hier ist TU 2')
11
12 def tu3():
13     print('hier ist TU 3')
14
15 menu = {1:tu1, 2:tu2, 3:tu3}
16
17 while eingabe != 'e':
18     eingabe = input('1 oder 2 oder 3 oder e')
19     print(eingabe)
20     try:
```

[5] Danke an Kati Naumann

```
21          einNum = int(eingabe)
22      except ValueError:
23          if eingabe == 'e':
24              break
25      try:
26          menu[einNum]()
27      except KeyError:
28          print('keine gültige Zahl')
```

Hier sind noch die wichtigsten 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)

Tabelle 14.1.: Fehler-Konstanten

Teil IX.

Permanente Speicherung

15. Dateizugriff

15.1. Dateien allgemein

Look out kid
It's somethin' you did

(Bob Dylan[1])

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. [2]

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
meineDatei = open('meinDateiName.txt', 'rb') # rb steht für binary Lesen,
                                             # z.B. für Bild/Musik etc.
                                             # -Daten
meineDatei = open('meinDateiName.txt', 'wb') # wb steht für binary
                                             # Schreiben z.B. für
                                             # Bild/Musik etc. Daten
meineDatei = open('meinDateiName.txt', 'r+') # r+ Lesen und Schreiben bei
                                             # existierender Datei
meineDatei = open('meinDateiName.txt', 'w+') # w+ Lesen und Schreiben,
                                             # nicht existierende Datei
                                             # wird angelegt,
                                             # existierende Datei wird
                                             # überschrieben
```

Damit ist klar: ab diesem Zeitpunkt ist `meineDatei` ein Objekt der Klasse `file`.

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

[2] 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?

Wie weiter oben schon einmal angemerkt: so lange man sich in Unix-artigen Betriebssystemen bewegt, gibt es nicht das Problem der Kodierung. Wenn man aber nicht weiß, wo überall eine Datei, die man gerade schreibt, gelesen werden soll, ist es sicherer, das encoding anzugeben:

```
meineDatei = open('meinDateiName.txt', 'w', encoding='utf8')
```

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.

Listing 15.1: Datei schreiben

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

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.

Listing 15.2: Datei lesen (alles auf einmal)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
# 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:

Listing 15.3: Datei zeilenweise lesen

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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[3] 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:

[3] siehe voriges Kapitel

Listing 15.4: Datei zeilenweise mit Iterator lesen

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

dat = open('bloedsinn.txt', 'r')
ende = False
while not(ende):
  try:
    zeile = next(dat)
    print(zeile)
  except StopIteration:
    ende = True
    break
dat.close()
```

Listing 15.5: Die Ausgabe des Programms

```
>>> Dies ist ziemlicher Blödsinn.
Es geht nur darum, einige Zeilen zu füllen.
Damit hat das Programm die Möglichkeit, mehrere Zeilen zu lesen.
Zur Probe werden die Zeilen dann ausgegeben.
```

Der Befehl `open` öffnet die Datei, macht aber noch nichts mit der Datei. Allerdings kann man den Datei-Handle als Liste benutzen und mit einer `for`-Schleife zeilenweise über die Datei schleifen:

Listing 15.6: Datei lesen mit `for`-Schleife

```
#!/usr/bin/python

meineDat = open("bloedsinn.txt","r")
for zeile in meineDat:
  print(zeile)
meineDat.close()

>>> Dies ist ziemlicher Blödsinn.
Es geht nur darum, einige Zeilen zu füllen.
Damit hat das Programm die Möglichkeit, mehrere Zeilen zu lesen.
Zur Probe werden die Zeilen dann ausgegeben.
```

Seit Version 2.5 gibt es in Python das Schlüsselwort `with`. Mit Hilfe von `with` wird ein Dateiobjekt geöffnet und vor allem nach dem Dateizugriff geschlossen. Das sieht dann so aus:

Listing 15.7: Gesamte Datei lesen mit `with`

```
#!/usr/bin/python

with open('bloedsinn.txt', 'r') as dat:
    zeilen = dat.read()
for eineZeile in zeilen:
    print(zeile)
```

Und die Ausgabe des Programms ist die selbe wie oben.

Die nächste Datei benutzt, dass durch `with` ein Iterator über die Datei erzeugt wird. Damit kann die `for`-Schleife diesen Iterator benutzen, womit das Programm nochmals kürzer wird.

Listing 15.8: Datei zeilenweise lesen mit `with`

```
#!/usr/bin/python

with open('bloedsinn.txt', 'r') as dat:
        for zeile in dat:
        print(zeile[:-1])
```

15.1.1. Weitere Datei-Zugriffsmodi

x erstellt Datei nur, wenn sie noch nicht existiert + öffnet Datei zum Update (= Lesen und Schreiben) wt öffnet im Textmodus wb öffnet im Binärmodus

15.1.2. Zippen

Listing 15.9: gezippte Datei lesen

```
>>> zipDName = dName+'.gz'
>>> zipDName

'/home/fmartin/temp/bliblablub.txt.gz'
>>> with gzip.open(zipDName, 'rt') as zD:
    inhalt = zD.read()


>>> print(inhalt)
Das ist eine blabla-Datei.
Da steht nur blub-Zeugs drin.
Hauptsächlich soll sie gezippt werden.

Erwartet man eine eher kleine Datei, ...usw.
```

15.1.3. print funktioniert auch!

Mit der print-Funktion (siehe auch S. 73) ab Python 3.x kann man auch einfach Dateien beschreiben. Dabei wird ausgenutzt, dass der Funktion ein benannter Parameter mit dem Namen **file** mitgegeben werden kann.

Listing 15.10: Datei mit **print** schreiben

```
>>> print('hallo Martin', file=open('hm.txt','w'))
```

Dies öffnet eine Datei und schreibt etwas hinein.

15.1.4. Textanalyse

Im nächsten Beispiel soll ein Text daraufhin untersucht werden, welches die häufigsten Wörter sind, die er enthält. Dazu wird

1. ein leeres Dictionary angelegt, in dem am Ende die Wörter mit ihrer Häufigkeit stehen werden
2. eine Datei geöffnet
3. die Datei zeilenweise gelesen
4. a) von jeder Zeile der Zeilenvorschub am Ende entfernt
   b) aus jeder Zeile die Satz- und Sonderzeichen entfernt
   c) jede Zeile in ihre einzelnen Wörter aufgesplittet
   d) i. von jedem Wort nachgeschaut, ob es schon im Dictionary steht
      ii. wenn ja, wird der Zähler um 1 erhöht
      iii. sonst wird der Zähler auf 1 gesetzt
5. das Dictionary ausgegeben

Listing 15.11: Häufigkeit von Wörtern

```
#!/usr/bin/python

import string
einDat = open('/home/fmartin/texte/sonstiges/sprueche.txt','r')

zaehler = {}

for zeile in einDat:
    zeile = zeile.rstrip()
    zeile = zeile.translate(zeile.maketrans('','',string.punctuation))
    woerter = zeile.split()
    for einWort in woerter:
        if einWort in zaehler:
            zaehler[einWort] += 1
        else:
            zaehler[einWort] = 1
print(zaehler)
```

Das Programm ist getestet! Schreib Dir selbst (mit einem Editor, nicht mit einer Textverarbeitung!!!) eine Datei mit irgendwelchem Text, speichere sie und ändere den Pfad und den Namen der zu öffnenden Datei. Und schau Dir dann die Ausgabe des Programms an.

Schau Dir an, was die Hilfe-Funktion zu

1. `string.punctuation`
2. `str.translate`
3. `str.maketrans`

sagt.

15.2. 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:

Listing 15.12: Der bisherige Konstruktor für Konten

```
1   def __init__(self, ktoNr, vorname, nachname, gebdat, wohnort,
2                     ktoStand, zinssatz):
3     self.ktoNr = ktoNr
4     self.inhaber = Mensch(vorname, nachname, gebdat, wohnort)
5     self.ktoStand = ktoStand
6     self.zinssatz = zinssatz
```

und hintendran gleich die neue Version, nach der Änderung, die ich ein paar Zeilen zuvor angesprochen habe:

Listing 15.13: Neuer Konstruktor: Parameter sind teilweise initialisiert

```
1   def __init__(self, ktoNr, vorname, nachname, gebdat, wohnort,
2                     ktoStand = 0,  zinssatz = 0.02):
3     self.ktoNr = ktoNr
4     self.inhaber = Mensch(vorname, nachname, gebdat, wohnort)
5     self.ktoStand = ktoStand
6     self.zinssatz = zinssatz
```

Was hier passiert, wurde schon im Abschnitt über Rückgabewerte und Parameterlistenbeim 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.

Listing 15.14: Ein paar kleine Hilfen für die Benutzer

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from clKonto import Konto

alleKonten = {}

def meldModus():
  print("Konto-Verwaltung\ne für Einzahlung\na für Auszahlung\ns für Schluss")

def meldAnl():
  print("Konto-Verwaltung\nn für 'Neues Konto anlegen'\ns für Schluss")
```

Außerdem wird ein leeres Dictionary `alleKonten` angelegt.

Danach startet man eine Schleife, um einige Konten anzulegen:

Listing 15.15: Eine Schleife für das Anlegen eines Kontos

```
eingabe = 'q'
while (eingabe != 's'):
  meldAnl()
  eingabe = input('Bitte Eingabe:')
  if eingabe == 'n':
    neuKtoNr = input('Bitte neue Konto-Nummer eingeben: ')
    neuVorname = input('Bitte Vornamen eingeben: ')
    neuNachname = input('Bitte Nachnamen eingeben: ')
    neuGebdat = input('Bitte Geburtsdatum eingeben: ')
    neuWohnort = input('Bitte Wohnort eingeben: ')
    aktuellesKonto = Konto(neuKtoNr, neuVorname, neuNachname,
                                   neuGebdat, neuWohnort)
    alleKonten[neuKtoNr] = aktuellesKonto
  elif eingabe == 's':
    pass
  else:
    print("Falsche Eingabe. nur n und s erlaubt")
```

In den Zeilen mit `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:

Listing 15.16: 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 = input('Bitte Eingabe:')
      if eingabe == 'e':
        betrag = float(input('Einzahlungsbetrag: '))
        aktuellesKonto.einzahlen(betrag)
      elif eingabe == 'a':
        betrag = float(input('Auszahlungsbetrag: '))
        aktuellesKonto.auszahlen(betrag)
      elif eingabe == 's':
        print('und tschüß')
      else:
        print("Falsche Eingabe. nur a, e, s erlaubt")
      aktuellesKonto.kontoStandAusgeben()
  else:
    pass
```

15.3. In die Datei damit!!!

15.3.1. Der Modul `pickle`

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:

Listing 15.17: Modul importieren: Ein Objekt in eine Datei schreiben

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

Listing 15.18: Ein Objekt in eine Datei schreiben

```
    dateiName = 'alleKonten.pck'
    ktoDatei = open(dateiName, 'wb')
    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.

Listing 15.19: Lesen eines Objektes aus einer Datei

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from clKonto import Konto
import pickle

alleKonten = {}

dateiName = 'alleKonten.pck'
ktoDatei = open(dateiName, 'rb')
while ktoDatei:
  try:
    alleKonten = pickle.load(ktoDatei)
  except EOFError:
    break
ktoDatei.close()

print("angelegte Konten: ", alleKonten.keys())
```

Wichtig!

Unter Python 2.x war der Dateizugriff mittels `pickle` nicht mehr mit Attributen rb und wb, sondern mit r und w.

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.

15.3.2. Der Modul `shelve`

Mit Hilfe des Moduls **`shelve`** kann man eine Art Datenbank simulieren. Dazu gibt es gleich ein Beispiel.

Listing 15.20: Schreiben einer Struktur mittels **shelve**

```
class Person():
    def __init__(self,v,n,o):
        self.vn = v
        self.nn = n
        self.ort = o

class ShelveNamen():
    def __init__(self):
        import shelve
        self.namensDB = shelve.open('namen','w')
    def satzSchreiben(self, schluessel, p):
        self.namensDB[schluessel] = p
    def schliessen(self):
        self.namensDB.close()

datenbank = ShelveNamen()

for i in range(5):
    vn = input('Vorname: ')
    nn = input('Nachname: ')
    ort = input('Ort: ')
    per = Person(vn,nn,ort)
    schluessel = input('Schluessel: ')
    datenbank.satzSchreiben(schluessel, per)
datenbank.schliessen()
```

In den ersten 5 Zeilen wird eine Klasse `Person` beschrieben, die nur die 3 Attribute Vorname, Nachname und Ort hat. Methoden benötigt diese Klasse nicht, da die einzelnen Objekte nichts machen sollen, sondern nur gespeichert werden sollen.

In den Zeilen 7 bis 14 wird eine Klasse `ShelveNamen` beschrieben, die im Konstruktor eine Datei anlegt, die die Datensätze (also die Objekte) aufnehmen soll. Diese Klasse hat die Methode `satzSchreiben`, die genau das macht, und die Methode `schliessen`, die die Datei am Ende schließt.

Im eigentlichen Programm wird dann zuerst eine `datenbank` angelegt, danach werden 5 Objekte der Klasse `Person` erzeugt, jedes mit einem Schlüssel versehen und jeweils in die Datenbank geschrieben. Zum Schluß wird die Datei geschlossen.

Das ergibt folgenden Dialog beim Aufruf:

Listing 15.21: Bildschirmausgabe des Programms

```
Vorname: Martin
Nachname: Schimmels
Ort: Oxxxxx
Schluessel: Martin
Vorname: Hannah
Nachname: yyyy
Ort: zzzz
Schluessel: Hannah
Vorname: Theresa
Nachname: xyz
Ort: xyz
Schluessel: Theresa
Vorname: Berthold
Nachname: zyx
Ort: zyxx
Schluessel: Berti
Vorname: Peter
Nachname: abc
Ort: abc
Schluessel: Peter

>>> db = shelve.open('namen','r')
>>> db['Peter'].vn
'Peter'
>>> db['Peter'].nn
'abc'
>>> db['Martin'].nn
'Schimmels'
```

Die ersten 20 Zeilen sind die Eingaben der je 3 Attribute jedes Objektes sowie des Schlüssels jedes Objektes. Die folgende Zeile öffnet die Datei zum lesen. Und dann kommt das „Datenbank-ähnliche“ dieser Speicherung: die einzelnen Attribute eines Datenbanksatzes (also eines Objekts) können mittels des Schlüssels und des Attributnamens ausgelesen werden.

15.4. Aufgaben 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.

16. Datenbanken

If your mem'ry serves you well

(Bob Dylan[1])

16.1. 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:

Listing 16.1: Schreiben in eine Datenbank

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  import MySQLdb
5  verbindung = MySQLdb.connect("localhost", db="nameort",
6                  user="karldepp", passwd="doof")
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()
```

[1] This Wheel's on Fire *auf:* The Basement Tapes

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:

Listing 16.2: Schreiben in eine Datenbank mit Fehlerbehandlung

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  import MySQLdb
5  try:
6    verbindung = MySQLdb.connect("localhost", db="nameort",
7                     user="karldepp", passwd="doof")
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()
```

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:

Listing 16.3: Schreiben in eine Datenbank mit differenzierter Fehlerbehandlung

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb
try:
  verbindung = MySQLdb.connect("localhost", db="plzort",
                               user="karldepp", passwd="doof")
  cursor = verbindung.cursor()
except MySQLdb.DatabaseError:
  print('Verbindung konnte nicht hergestellt werden')
except:
  print('Cursor konnte nicht gesetzt werden')
else:
  try:
    cursor.execute("INSERT INTO plzort (plz, ort)
                   VALUES (%s, %s)", ('72070', 'Tübingen'))
  except:
    print('Insert fehlgeschlagen')
  else:
    print('ok')
    verbindung.commit()
    cursor.close()
    verbindung.close()
```

MySQL (wie jeder SQL-Dialekt) puffert Befehle. Wenn man mit einer Entwicklungsumgebung wie zum Beispiel der `MySQL-Workbench` arbeitet, merkt man das gar nicht. Bei einem solchen Werkzeug, das für das interaktive Arbeiten ausgerichtet ist, ist es wichtig, dass das Ergebnis der Eingabe eines Befehls gleich sichtbar ist. Befehle werden hier sofort ausgeführt. Wenn man nicht interaktiv arbeitet, muss man das Schreiben aus dem Puffer in die Datenbank meistens von Hand anstoßen. Das geschieht mit dem SQL-Befehl **commit**.

Das Lesen aus einer Datenbank ist ein ganzes Stück komplizierter. Das sollte eigentlich klar sein: wenn ich (in eine Datenbank) schreibe, weiß ich, was ich schreibe; da kann nichts unerwartetes passieren. Wenn ich ein Wort schreibe, dann ist das Ergebnis *ein Wort*, wenn ich zehn Wörter schreibe, ist das Ergebnis *zehn Wörter*, wenn ich 3 Zahlen schreibe, ist das Ergebnis *3 Zahlen.*

Wenn ich (aus einer Datenbank) lese, weiß ich nicht, was da auf mich zukommt. Das kann ein Wort, das können 10 Wörter oder 3 Zahlen sein. Auf eine Datenbank bezogen bedeutet das, dass ich das Ergebnis eines **SELECT**-Befehls nicht vorhersagen kann. Da kann es noch schlimmer kommen, als es im vorigen Satz geschildert wurde: es kann sogar gar nichts als Ergebnis zurückgegeben werden! Die Ergebnismenge kann 0, 1 oder viele Elemente enthalten.

Deswegen ist das Vorgehen ein leicht anderes. Die erste Änderung ist, dass das Ergebnis der Ausführung des Befehls, also das, was durch **cursor.execute()** geschieht, einer Variablen zugewiesen werden muss. Die zweite Änderung ist dann eigentlich logisch: über

das, was in dieser Variablen steht, muss jetzt eine Schleife gemacht werden, damit ich die einzelnen Datensätze nacheinander verarbeiten kann.

Listing 16.4: Lesen aus einer Datenbank

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb
connection = MySQLdb.connect("localhost", db="nameort",
                    user="karldepp", passwd="doof")
cursor = connection.cursor()
t2 = cursor.execute("SELECT * FROM namen")
for i in range(t2):
  print(cursor.fetchone())

cursor.close()
connection.close()
```

16.2. ...mit Python-Bordmitteln

In Python gibt es einen weiteren Daten-Verpacker, der wie `pickle` Objekte entgegen nimmt, aber einige Fähigkeiten mehr hat. Der Modul `shelve` hat Eigenschaften, die an eine Datenbank erinnern.

Teil X.
Noch ein paar Beispiele zur OOP

17. Ein weiteres Projekt: ein Getränkeautomat

17.1. 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:

```
automat
---------------
kredit
waren
---------------
__init__
auswahlTreffen
bestandPruefen
geldEinwerfen
guthabenAnzeigen
neueWare
wareWaehlen
```

Abbildung 17.1.: 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?

Listing 17.1: Klassenentwurf des Getränkeautomaten

```
class Automat():
  def __init__(self, waren=[], kredit = 0):
    self.waren = waren
    self.kredit = kredit

  def geldEinwerfen(self):
    print('Geld eingeworfen')

  def guthabenAnzeigen(self):
    print('Guthaben anzeigen')

  def wareWaehlen(self):
    print('Ware auswählen')

  def neueWareAufnehmen(self):
    print('neue Ware hinzugenommen')

  def bestandPruefen(self):
    print('Bestand der gewählten Ware geprüft')
```

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!

Listing 17.2: Klassenentwurf des Menus

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from cl_automat import Automat

class Menu():

  def __init__(self):
    self.automat = Automat()
    self.prompt = 'g = Geld einwerfen,
                   w = Ware wählen, n = neue Ware einführen,
                   e = Ende\n'

  def auswahlTreffen(self):
    auswahl = ['e', 'w', 'g', 'n']
    eingabe = 'x'
    while eingabe <>'e':
      eingabe = input(self.prompt)
      if eingabe == 'g':
        self.automat.geldEinwerfen()
      elif eingabe == 'w':
        self.automat.wareWaehlen()
      elif eingabe == 'n':
        self.automat.neueWareAufnehmen()
      elif eingabe == 'e':
        pass
      else:
        print('fehlerhafte Eingabe; nur e, g, n, w zulässig')
```

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.

Listing 17.3: Das aufrufende Programm für den Getränkeautomaten

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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.

17.2. 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 17.2.: 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.

Listing 17.4: Die Waren kommen hinzu

```
1  class Ware():
2    def __init__(self, bezeichnung, verkPreis, bestand):
3      self.bezeichnung = bezeichnung
4      self.verkPreis = verkPreis
5      self.bestand = bestand
6
7  class Automat():
8      ...
9      ...
10   def warenAnzeigen(self):
11     print('folgende Waren können gekauft werden: ')
12     print('%20s %10s %10s' % ('Ware', 'Preis', 'Bestand'))
13     for ware in self.waren:
14       print('%20s %10.2f %10i' % (ware.bezeichnung,
15                           ware.verkPreis, ware.bestand))
16
17   def neueWareAufnehmen(self):
18     bez = input('Bezeichnung der neuen Ware: ')
19     vkPr = float(input('Verkaufspreis der neuen Ware: '))
20     best = float(input('Lagerbestand der neuen Ware: '))
21     nW = ware(bez, vkPr, best)
22     self.waren.append(nW)
23     print('neue Ware hinzugenommen')
24     self.warenAnzeigen()
```

Das aufrufende Programm ändert sich überhaupt nicht! (Ist das ein gutes Zeichen??? Schon, oder?)

17.3. 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.

Listing 17.5: Geld einwerfen und Guthaben anzeigen

```
1   def geldEinwerfen(self):
2     muenzen = [1, 2, 5, 10, 20, 50, 100, 200]
3     einwurf = float(input('Münze einwerfen: '))
4     if einwurf in muenzen:
5       self.kredit += einwurf
6     else:
7       print('ungültige Münze, gültig sind: ', muenzen)
8     self.guthabenAnzeigen()
9
10  def guthabenAnzeigen(self):
11    print('aktuelles Guthaben: %5.2f <:€:>' % (self.kredit/100.0))
```

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 . . .

17.4. 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.

Listing 17.6: Der Kauf

```
def wareWaehlen(self):
  print('Ware auswählen')
  self.warenAnzeigen()
  ausgewaehlt = input('Nr. der gewünschten Ware eingeben: ')
  if not self.bestandPruefen(ausgewaehlt):
    print('zu wenig Ware auf Lager')
  elif self.kredit <= self.waren[ausgewaehlt].verkPreis:
    print('Du hast zuwenig Geld eingeworfen')
  else:
    print('Ware wird ausgegeben')
    self.waren[ausgewaehlt].bestand -= 1
    print('Restgeld %4.2f <:€:> wird ausgegeben' %
              (self.kredit - self.waren[ausgewaehlt].verkPreis))
    self.kredit = 0.0

def bestandPruefen(self, nr):
  if self.waren[nr].bestand <= 1:
    return False
  else:
    return True
```

Hier gehört nur noch ein blöder Spruch hin: Das aufrufende Programm ändert sich überhaupt nicht! (Ist das ein gutes Zeichen??? Super!!!!)

17.5. 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.

18. Noch ein Beispiel: Zimmerbuchung in einem Hotel

18.1. 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.

18.2. 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).

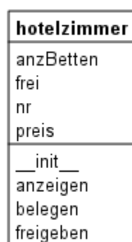


Abbildung 18.1.: Klassendigramm `Hotelzimmer`

Die Realisierung in Python folgt:

Listing 18.1: Eine erste Klasse

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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

18.3. 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.

Listing 18.2: Menu fürs Hotel

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  from clZimmer import Hotelzimmer
4
5  def menu():
6    auswahl = input('\n#### AUSWAHL ####\nZimmer anzeigen: Z
7                     \nZimmer belegen: B\nZimmer freigeben: F
8                     \nEnde: E\n#### ....... ####
9                     \b\b\b\b\b\b\b\b\b\b\b\b')
10   menuListe = {'Z': zimmer1.anzeigen,  'B': zimmer1.belegen,
11                'F': zimmer1.freigeben,  'E': beenden}
12   menuListe[auswahl]()
13
14 def beenden():
15   global weiter
16   print('danke für Deinen Besuch')
17   weiter = 0
18
19
20 weiter = 1
21 zimmer1 = Hotelzimmer(1, 2)
22 while weiter:
23   menu()
```

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

18.4. 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`, 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:

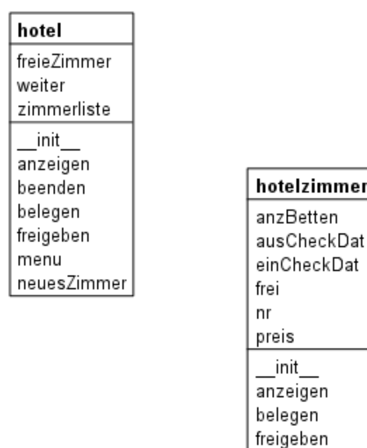


Abbildung 18.2.: Klassendiagramm `Hotel`

Listing 18.3: Entwurf der Klasse `Hotel`

```
class Hotel():
  def __init__(self):
    self.zimmerliste = []
    self.weiter = True
    while self.weiter:
      self.menu()

  def menu(self):
          auswahl = 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\b')
    menuListe = {'Z': self.anzeigen, 'B': self.belegen,
                 'F': self.freigeben, 'N':self.neuesZimmerAnlegen,
                 'E': self.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 = int(input('Anzahl Betten des Zimmers: '))
    preis = float(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 !!!!!

Listing 18.4: Aufrufendes Programm für das Hotel

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from clZiHo import Hotel

meinHotel = Hotel()
```

18.5. 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.

Listing 18.5: 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 = 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.neuesZimmerAnlegen,
                 'E': self.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:
```

```
31                 zimmer.anzeigen()
32           else:
33             for zimmer in self.zimmerliste:
34               zimmer.anzeigen()
35
36       def belegen(self):
37         print('folgende Zimmer sind verfügbar: ')
38         # erster Parameter = True:  freie Zimmer anzeigen
39         self.anzeigen(True)
40         auswahl =  int(input('gewünschte Zimmer-Nr. angeben: '))
41         self.zimmerliste[auswahl - 1].belegen()
42
43       def freigeben(self):
44         self.anzeigen(False)
45         auswahl =  int(input('freizugebendes Zimmer: '))
46         self.zimmerliste[auswahl - 1].freigeben()
47
48       def neuesZimmerAnlegen(self):
49         anzBetten = int(input('Anzahl Betten des Zimmers: '))
50         preis = float(input('Preis des Zimmers: '))
51         neuesZi = Hotelzimmer(len(self.zimmerliste)+1, anzBetten,  preis)
52         self.zimmerliste.append(neuesZi)
```

19. Und noch ein Beispiel: ein Adressbuch

19.1. 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.

19.2. Der erste Entwurf: eine Liste von Listen

Der erste Versuch speichert die Daten einer Person in einer Liste. Das sieht also so aus:

Listing 19.1: Adresse in einer Liste

```
1
2  #!/usr/bin/python
3  # -*- coding: utf-8 -*-
4
5  martin = ['Martin', 'Schimmels',54]
6  ekki = ['Eckard', 'Krauss',41]
```

Die Personen-Daten von mehreren Personen speichern wir in einer Liste von Listen, also:

```
1  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.

Listing 19.2: Zugriff auf die Adress-Liste

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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
```

19.3. Der zweite Entwurf: ein fauler Trick

Hier soll nur eine weiter oben bei 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:

Listing 19.3: Adress-Liste (mit faulem Trick)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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])
```

19.4. 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.

Listing 19.4: Adress-Liste als Dictionary

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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!

19.5. 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:

Listing 19.5: Funktionen für die Adress-Liste(noch Dummies)

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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():
  print(tut mir leid, das Menu ist noch nicht fertig')

initialisiere()
menu()
```

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 „\“ 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 Programme 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.

Listing 19.6: Adress-Liste mit Menu

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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 = 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()
```

19.6. 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.

Listing 19.7: Adress-Liste anzeigen

```
1  def anzeigen():
2    global namen
3    print('##### Namensliste #####')
4    for einName in namen:
5      print('##\t', end=' ')
6      print(einName['vorname'],einName['nachname'],einName['alter'])
7    print('##### ------------- #####')
```

Das Einfügen eines neuen Namens in die Liste ist auch kein Hexenwerk. Die einzelnen Informationen werden über `input` eingelesen, das ganze wird zu einem Dictionary verbunden und dieses Dictionary wird an die Liste `namen` angehängt

Listing 19.8: Eintrag in Adress-Liste machen

```
1  def einfuegen():
2    global namen
3    print('Neuer Name wird eingefügt!')
4    vorname = input('Vorname: ')
5    nachname = input('Nachname: ')
6    alter = input('Alter: ')
7    namen.append({'vorname':vorname,  'nachname':nachname,  'alter':alter})
```

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`.

Listing 19.9: Adress-Liste mit Nummern anzeigen

```
def anzeigen(mitNummer = False):
  global namen
  i = 0
  print('##### Namensliste #####')
  for einName in namen:
    print('##\t', end=' ')
    if mitNummer:
      print(i, end=' ')
      i += 1
    print(einName['vorname'],einName['nachname'],einName['alter'])
  print('##### ——————— #####')
```

Und die Funktion „löschen“ gleich hintendran:

Listing 19.10: 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.

Listing 19.11: 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 = input(' neuer Vorname(Enter für beibehalten): ')
  if neuerVorname != u'':
    namen[aendNr]['vorname'] = neuerVorname
  #### Nachname
  print('alter Nachname: ', namen[aendNr]['nachname'])
  neuerNachname = 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:

Listing 19.12: 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 = input(' neu (Enter für beibehalten): ')
    if aenderung != u'':
      namen[aendNr][attribut] = aenderung
```

[1]

19.7. 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

[1] capitalize schreibt den ersten Buchstaben eines Wortes groß

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.

Listing 19.13: gespeicherte Adress-Liste lesen

```
1  def nDatLesen(namenDatei):
2    global namen
3    while namenDatei:
4      try:
5        namen = pickle.load(namenDatei)
6      except EOFError:
7        break
8    namenDatei.close()
9
10 def initialisiere():
11   global namen
12   nDat = 'meineNamen.dat'
13   try:
14     namenDatei = open(nDat, 'r')
15     nDatLesen(namenDatei)
16   except IOError:
17     martin = {'vorname':'Martin', 'nachname':'Schimmels','alter':54}
18     ekki = {'vorname':'Eckard', 'nachname':'Krauss','alter':41}
19     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.

Listing 19.14: Adress-Liste schreiben

```
1  def beenden():
2    print('Jetzt wird das Programm beendet')
3    global namen
4    nDat = 'meineNamen.dat'
5    try:
6      namenDatei = open(nDat, 'w')
7    except IOError:
8      print('na so was')
9    pickle.dump(namen, namenDatei)
10   namenDatei.close()
```

19.8. Jetzt wird es objektorientiert

19.8.1. 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.

Adresse
alter nachname vorname
__init__ aendern alterAendern anlegen anzeigen nachnameAendern vornameAendern

Abbildung 19.1.: Klassendiagramm `Adresse`

Der Klassenentwurf ist danach selbsterklärend:

Listing 19.15: Klasse `Adresse`

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

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 = input(aString)

  def nachnameAendern(self):
    aString = 'alter Nachname: '+self.nachname+' neuer Nachname: '
    self.nachname = input(aString)

  def alterAendern(self):
    aString = 'altes Alter: '+str(self.alter)+' neues Alter: '
    self.alter = input(aString)

  def aendern(self):
    auswahl = 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 = input('Vorname: ')
    self.nachname = input('Nachname: ')
    self.alter = input('Alter: ')
```

Dazu benötigt man wieder ein Programm, das diese Klasse aufruft:

Listing 19.16: Aufruf der Klasse `Adresse`

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from clAdr import Adresse

ich = Adresse('Martin', 'Schimmels', 54)
ich.anzeigen()
ich.aendern()
ich.anzeigen()

du = Adresse()
du.anzeigen()
```

19.8.2. 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 Der fünfte Entwurf: Aktionen (jetzt tut sich wirklich etwas)!! sehr ähnlich.

Listing 19.17: Adressbuch-Klasse

```
class AdressBuch():
  def __init__(self):
    self.weiter = True
    self.adrListe = []
    while self.weiter:
      self.menu()

  def menu(self):
    auswahl = input('\n#### 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':self.einfuegen, 'L':self.loeschen,
                 'A':self.aendern, 'E':self.beenden, 'Z':self.anzeigen}
    menuListe[auswahl]()

  def einfuegen(self):
    neuerName = Adresse()
    self.adrListe.append(neuerName)

  def loeschen(self):
    print('Folgende Namen sind in der Liste: ')
```

```
    self.anzeigen()
    loeschNr = input('Nummer des zu löschenden Datensatzes angeben: ')
    del self.adrListe[loeschNr]
    print("So sieht's jetzt aus: ")
    self.anzeigen()

  def aendern(self):
    print('Folgende Namen sind in der Liste: ')
    self.anzeigen()
    aendNr = input('Nummer des zu ändernden Datensatzes angeben: ')
    self.adrListe[aendNr].aendern()

  def beenden(self):
    self.weiter = False

  def anzeigen(self):
    lfdNr = 0
    print('\n##### Namensliste #####')
    for einName in self.adrListe:
      print('##\t',lfdNr,'\t')
      einName.anzeigen()
      lfdNr += 1
    print('##### ------------ #####')
```

Das Adressbuch selber wird durch das folgende aufrufende Programm realisiert:

Listing 19.18: Aufruf des Adressbuchs

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from clAdr import AdressBuch

meineAdressen = AdressBuch()
```

Das soll wirklich alles sein? Wer es nicht glaubt, soll es halt einfach ausprobieren!!

19.8.3. Dauerhafte Speicherung

Das wird wieder fast wortwörtlich aus dem fünften Entwurf übernommen:

Listing 19.19: Adressbuch-Klasse mit Speicherung

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import pickle
class Adresse():
  def __init__(self, vorname='', nachname='', alter=0):
    if vorname == '' and nachname == '' and alter == 0:
      self.anlegen()
    else:
```

```
9         self.vorname = vorname
10        self.nachname = nachname
11        self.alter = alter
12
13    def anzeigen(self):
14      print(self.vorname, ' ', self.nachname,  ' ',  self.alter)
15
16    def vornameAendern(self):
17      aString = 'alter Vorname: '+self.vorname+ ' neuer Vorname: '
18      self.vorname = input(aString)
19
20    def nachnameAendern(self):
21      aString = 'alter Nachname: '+self.nachname+' neuer Nachname: '
22      self.nachname = input(aString)
23
24    def alterAendern(self):
25      aString = 'altes Alter: '+str(self.alter)+' neues Alter: '
26      self.alter = input(aString)
27
28    def aendern(self):
29      auswahl = input('Was soll geändert werden?
30                   (V)orname, (N)achname, (A)lter ' )
31      menuAendern = {'V': self.vornameAendern,
32                     'N': self.nachnameAendern,
33                     'A': self.alterAendern}
34      menuAendern[auswahl]()
35
36    def anlegen(self):
37      print('neuer Name wird aufgenommen!')
38      self.vorname = input('Vorname: ')
39      self.nachname = input('Nachname: ')
40      self.alter = input('Alter: ')
41
42  class AdressBuch():
43
44    def __init__(self):
45      self.weiter = True
46
47      self.nDat = 'meineNamen.dat'
48      try:
49        namenDatei = open(self.nDat, 'r')
50        self.nDatLesen(namenDatei)
51      except IOError:
52        self.adrListe = []
53      while self.weiter:
54        self.menu()
55
56    def nDatLesen(self, namenDatei):
57      while namenDatei:
```

```
58          try:
59            self.adrListe = pickle.load(namenDatei)
60          except EOFError:
61            break
62        namenDatei.close()
63
64      def menu(self):
65        auswahl = input('\n#### AUSWAHL ####\n
66                     alle Namen anzeigen: Z\n
67                     Namen einfügen: N\n
68                     Namen löschen: L\n
69                     Namen ändern: A\n
70                     Ende: E\n
71                     #### ....... ####\b\b\b\b\b\b\b\b\b\b\b\b')
72        menuListe = {'N':self.einfuegen,
73                     'L':self.loeschen,
74                     'A':self.aendern,
75                     'E':self.beenden,
76                     'Z':self.anzeigen}
77        menuListe[auswahl]()
78
79      def einfuegen(self):
80        neuerName = Adresse()
81        self.adrListe.append(neuerName)
82
83      def loeschen(self):
84        print('Folgende Namen sind in der Liste: ')
85        self.anzeigen()
86        loeschNr = input('Nummer des zu löschenden Datensatzes angeben: ')
87        del self.adrListe[loeschNr]
88        print("So sieht's jetzt aus: ")
89        self.anzeigen()
90
91      def aendern(self):
92        print('Folgende Namen sind in der Liste: ')
93        self.anzeigen()
94        aendNr = input('Nummer des zu ändernden Datensatzes angeben: ')
95        self.adrListe[aendNr].aendern()
96
97      def beenden(self):
98        self.weiter = False
99        try:
100         namenDatei = open(self.nDat, 'w')
101       except IOError:
102         print('na so was')
103       pickle.dump(self.adrListe, namenDatei)
104       namenDatei.close()
105
106     def anzeigen(self):
```

```
107         lfdNr = 0
108         print('\n##### Namensliste #####')
109         for einName in self.adrListe:
110             print('##\t',lfdNr,'\t')
111             einName.anzeigen()
112             lfdNr += 1
113         print('##### ------------ #####')
```

20. Immer noch das Adressbuch — nur schöner

20.1. 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:

Listing 20.1: Klasse `Adresse` mit laufender Nummer

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  import MySQLdb
4  class Adresse():
5    def __init__(self, lfdnr = 0, vorname='', nachname='',
6                     strasse = '', plz = '',  ort = ''):
7      if vorname == '' and nachname == '':
8        self.anlegen()
9      else:
10       self.lfdnr =  lfdnr
11       self.vorname = vorname
12       self.nachname = nachname
13       self.strasse =  strasse
14       self.plz = plz
15       self.ort = ort
16
17   def anzeigen(self):
18     print('\n######## Nr. des Datensatzes: ', self.lfdnr,
19           '\n## ',self.vorname, ' ', self.nachname,
20           '\n## ',  self.strasse, '\n## ',
21           self.plz, ' ',  self.ort, '\n########')
22
23   def vornameAendern(self):
24     print('häää?? Vornamen ändern??')
```

```
        aString = 'alter Vorname: '+self.vorname+ ' neuer Vorname: '
        self.vorname = input(aString)

    def nachnameAendern(self):
        aString = 'alter Nachname: '+self.nachname+' neuer Nachname: '
        self.nachname = input(aString)

    def strAendern(self):
        aString = 'alte Strasse: '+str(self.strasse)+' neue Strasse: '
        self.strasse = input(aString)

    def plzAendern(self):
        aString = 'alte PLZ: '+str(self.plz)+' neue PLZ: '
        self.plz = input(aString)

    def ortAendern(self):
        aString = 'alter Ort: '+str(self.ort)+' neuer Ort: '
        self.ort = input(aString)

    def aendern(self):
        auswahl = input('Was soll geändert werden?
                          (V)orname, (N)achname, (A)lter,
                          (S)trasse, (P)LZ, (O)rt ' )
        menuAendern = {'V': self.vornameAendern, 'N': self.nachnameAendern,
                       'S': self.strAendern, 'P': self.plzAendern,
                       'O': self.ortAendern}
        menuAendern[auswahl]()
```

Listing 20.2: Klasse `Adresse` mit laufender Nummer (Forts.)

```
    def anlegen(self):
        print('neuer Name wird aufgenommen!')
        self.vorname = input('Vorname: ')
        self.nachname = input('Nachname: ')
        self.strasse = input('Strasse: ')
        self.plz = input('PLZ: ')
        self.ort = 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.

Listing 20.3: 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.

Listing 20.4: Methoden der Klasse `AdressBuch`

```
    def nDatLesen(self):
      self.meinCursor.execute('SELECT * FROM namen ORDER BY
            nachname, vorname')
      for (lfdnr, vn, nn, strasse, plz, ort)
            in self.meinCursor.fetchall():
        geleseneAdr = Adresse(lfdnr, vn, nn, strasse, plz, ort)
        self.adrListe.append(geleseneAdr)

    def menu(self):
      auswahl = input('\n#### 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':self.einfuegen, 'L':self.loeschen,
                   'A':self.aendern, 'E':self.beenden,
                   'Z':self.anzeigen}
      menuListe[auswahl]()

    def einfuegen(self):
      neuerName = Adresse()
      self.meinCursor.execute('INSERT INTO namen
                                  (vorname, nachname, str, plz, ort)
                                  VALUES(%s, %s, %s, %s, %s)',
                                  (neuerName.vorname, neuerName.nachname,
                                   neuerName.strasse, neuerName.plz,
                                   neuerName.ort))

    def loeschen(self):
      print('Folgende Namen sind in der Liste: ')
      self.anzeigen()
      loeschNr = input('Nummer des zu löschenden Datensatzes angeben: ')
      loeschBefehl = 'DELETE FROM namen WHERE lfdnr = '+str(loeschNr)
      self.meinCursor.execute(loeschBefehl)

    def aendern(self):
      print('Folgende Namen sind in der Liste: ')
      self.anzeigen()
      aendNr = input('Nummer des zu ändernden Datensatzes angeben: ')
      vorleseBefehl = 'SELECT * FROM namen WHERE lfdnr = '+str(aendNr)
      self.meinCursor.execute(vorleseBefehl)
      for (lfdnr, vn, nn, strasse, plz, ort)
            in self.meinCursor.fetchall():
        geleseneAdr = Adresse(lfdnr, vn, nn, strasse, plz, ort)

      geleseneAdr.aendern()
```

```
49        updBefehl = "UPDATE namen
50                      SET vorname = '"+geleseneAdr.vorname+"',
51                          nachname = '"+geleseneAdr.nachname+"',
52                          str = '"+geleseneAdr.strasse+"',
53                          plz = '"+geleseneAdr.plz+"',
54                          ort ='"+geleseneAdr.ort+"'
55                          WHERE lfdnr = "+str(aendNr)
56        self.meinCursor.execute(updBefehl)
57
58      def beenden(self):
59        self.weiter = False
60
61      def anzeigen(self):
62        lfdNr = 0
63        print('\n##### Namensliste #####')
64
65        for einName in self.adrListe:
66          einName.anzeigen()
67        print('##### ------------ #####')
```

21. Eine Ampel

21.1. 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 $ightarrowrot - gelb$ ightarrow grün $ightarrowgelb$ ightarrow 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:

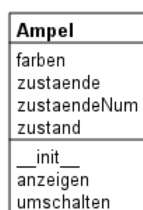


Abbildung 21.1.: Klassendiagramm `Ampel`

21.2. Die Realisierung

Der Quellcode für diese Klasse sieht so aus:

[1] The wind cries Mary *auf:* Are you experienced?

Listing 21.1: Klassenentwurf der Ampel

```
#!/usr/bin/python

class Ampel():
  def __init__(self):
    self.zustaendeNum = {0:'rot', 1:'rot-gelb', 2:'gruen', 3:'gelb'}
    self.farben = {'rot':'#FF0000', 'gelb':'#FFFF00',
                   'gruen':'#00FF00', 'schwarz':'#000000'}
    self.zustaende = {
      'rot':[self.farben['rot'], self.farben['schwarz'],
             self.farben['schwarz']],
      'rot-gelb':[self.farben['rot'], self.farben['gelb'],
             self.farben['schwarz']],
      'gruen':[self.farben['schwarz'], self.farben['schwarz'],
             self.farben['gruen']],
      'gelb':[self.farben['schwarz'], self.farben['gelb'],
             self.farben['schwarz']]}
    self.zustand = 0

  def umschalten(self):
    # das eigentliche Umschalten. BEACHTE: hier wird mod(3) gerechnet
    self.zustand = (self.zustand + 1)%len(self.zustaendeNum)

  def anzeigen(self):
    print('\n==========\nDie Ampel ist %s' % self.zustaendeNum[self.zustand])
    print('Farben: ', self.zustaende[self.zustaendeNum[self.zustand]])
```

Das aufrufende Programm ist wieder sehr einfach. Hier wird eine Endlosschleife benutzt, um die Ampel umzuschalten.

Listing 21.2: Aufruf einer Ampel

```
#!/usr/bin/python

from clAmpel import Ampel

eingabe = '12345'
meineAmpel = Ampel()
while eingabe != 'x':
  eingabe = input('Umschalten!!! Ende mit x')
  meineAmpel.umschalten()
  meineAmpel.anzeigen()
```

21.3. 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.

Listing 21.3: Klasse Ampel mit Speicherung des Zustands

```
1  #!/usr/bin/python
2
3  class Ampel():
4    def __init__(self, bezeichnung = 'Nord'):
5      self.bezeichnung = bezeichnung
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'],
11              self.farben['schwarz']],
12       'rot-gelb':[self.farben['rot'], self.farben['gelb'],
13              self.farben['schwarz']],
14       'gruen':[self.farben['schwarz'], self.farben['schwarz'],
15              self.farben['gruen']],
16       'gelb':[self.farben['schwarz'], self.farben['gelb'],
17              self.farben['schwarz']]}
18     self.lampen = ['oben', 'mitte', 'unten']
19
20   def umschalten(self):
21     # aus Datei den aktuellen Zustand lesen
22     try:
23       self.dateiOeffnen('r')
24       self.zustand = int(self.meineDat.read())
25       self.dateiSchliessen()
26     except:
27       self.zustand = 0
28
29     # das eigentliche Umschalten. BEACHTE: hier wird mod(3) gerechnet
30     self.zustand = (self.zustand + 1)%len(self.zustaendeNum)
31
32     # in Datei den aktuellen Zustand schreiben
33     self.dateiOeffnen('w')
34     self.dateiSchreiben()
35     self.dateiSchliessen()
36
37   def anzeigen(self):
38     print('\n==========\nDie Ampel ist %s' % self.zustaendeNum[self.zustand])
39     print('Farben: ', self.zustaende[self.zustaendeNum[self.zustand]])
40
```

```
41 ### Datei-Operationen
42   def dateiOeffnen(self,  modus):
43     dateiName = self.bezeichnung+'.dat'
44     self.meineDat = open(dateiName,modus)
45 
46   def dateiSchreiben(self):
47     self.meineDat.write(str(self.zustand))
48 
49   def dateiSchliessen(self):
50     self.meineDat.close()
```

Das aufrufende Programm muss nicht verändert werden!

21.4. 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.

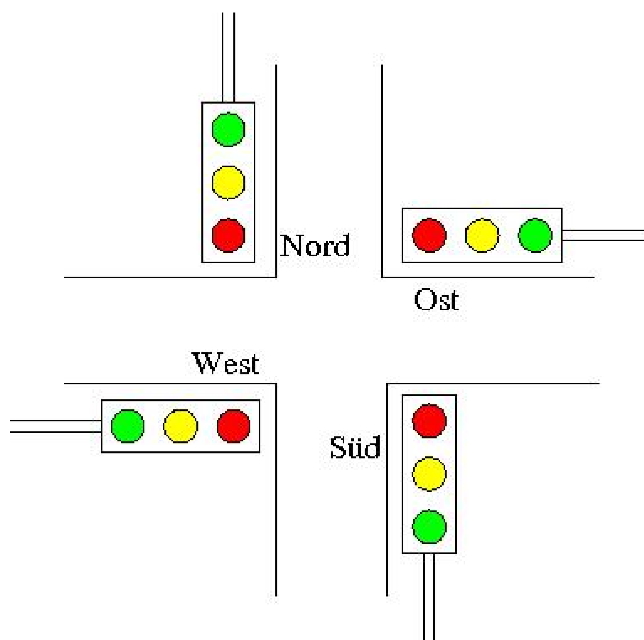


Abbildung 21.2.: Eine Kreuzung

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.

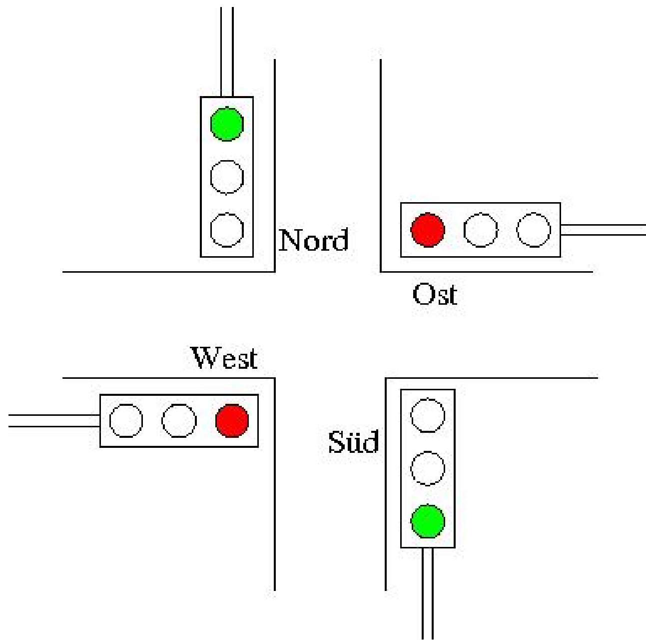


Abbildung 21.3.: Kreuzung: Freie Fahrt von Nord nach Süd

Dann wird umgeschaltet in die Gelb-Phase.

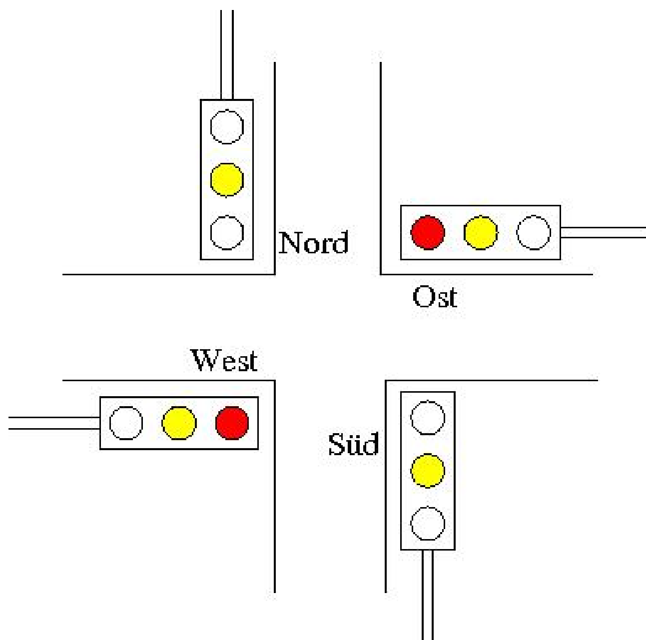


Abbildung 21.4.: Kreuzung: Alle warten

Dann darf man von West nach Ost (und umgekehrt) fahren.

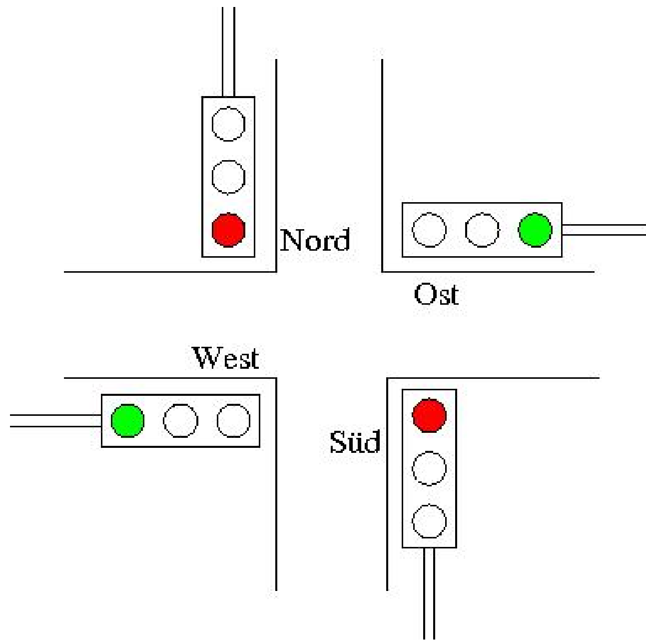


Abbildung 21.5.: Kreuzung: Freie Fahrt von West nach Ost

Dann wird umgeschaltet in die andere Gelb-Phase.

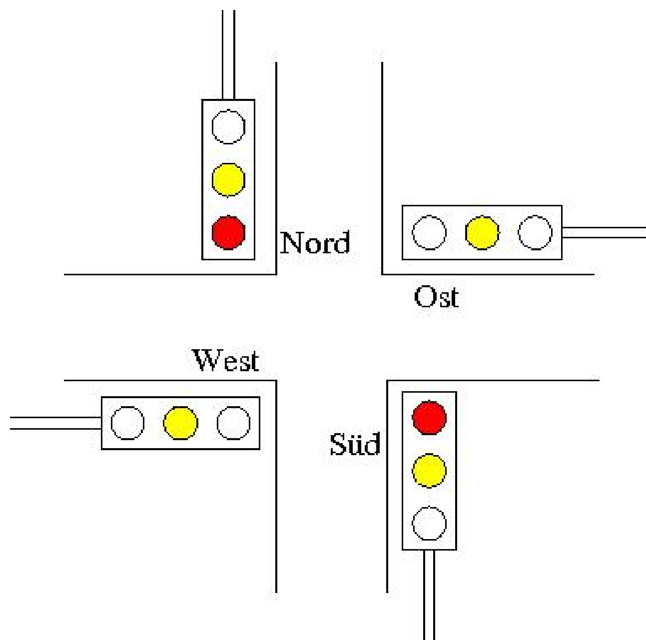


Abbildung 21.6.: Kreuzung: Wieder warten alle

In einer Tabelle sieht das so aus:

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

Tabelle 21.1.: Die Ampel-Phasen

21.5. 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.

Listing 21.4: Klassenentwurf der Kreuzung

```
#!/usr/bin/python

from clAmpel import Ampel

class Kreuzung():
  def __init__(self):
    self.richtungen = ['Nord', 'West', 'Sued', 'Ost']

    self.nordAmpel = Ampel('Nord')
    self.westAmpel = Ampel('West')
    self.suedAmpel = Ampel('Sued')
    self.ostAmpel = Ampel('Ost')
    self.alleAmpeln = [self.nordAmpel, self.westAmpel,
                       self.suedAmpel, self.ostAmpel]

  def umschalten(self):
    for eineAmpel in self.alleAmpeln:
      eineAmpel.umschalten()
```

Weiter hinten im Kapitel 23, 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 XI.

Grafik! Internet!

22. CGI-Programme

22.1. 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.

22.2. Allgemeines zu HTML

22.2.1. 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.

[1] siehe hierzu: http://de.selfhtml.org/

22.2.2. 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 kleiner- und größer-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.

22.2.3. Obligatorische HTML-Befehle

Jedes HTML-Dokument benötigt wenigstens folgende HTML-tags:

1. <HTML> und </HTML>
2. <HEAD> und </HEAD>
3. <TITLE> und </TITLE>
4. <BODY> und </BODY>

Damit sieht ein einfaches HTML-Dokument (ohne jeglichen Inhalt) so aus:

```
1 <HTML>
2   <HEAD>
3     <TITLE>
4     </TITLE>
5   </HEAD>
6   <BODY>
7   </BODY>
8 </HTML>
```

22.2.4. Struktur eines HTML-Dokuments

Es gibt 6 verschiedene Hierarchiestufen für Überschriften. Die oberste Stufe für Überschriften wird durch das tag-Paar <H1> </H1>, die unterste entsprechend durch das tag-Paar <H6> </H6> eingeschlossen.

22.2.5. 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 <BR> erledigt. Vorsicht: zu diesem „tag“ gibt es kein Ende-tag.

22.2.6. 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> `ist für Programmcode gedacht. Dieser Text wird in Schreibmaschinenschrift wiedergegeben.`
- <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.

22.2.7. 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.

Ungeordnete Listen werden durch Rauten, Rechtecke, etc. gekennzeichnet. Ungeordnete Listen werden in den <UL>-Container gepackt. Listenelemente werden in den <LI>-Container gepackt.

Definitionslisten 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.

22.2.8. 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.

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.

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“ .)

22.2.9. Formulare

Ein Formular wird durch das tag-Paar <FORM> und </FORM> eingeschlossen. Dabei benötigt das <FORM>-tag noch Attribute, damit überhaupt mit dem Formular gearbeitet werden kann. Vollständig sieht das Attribut so aus: <FORM action=glqq prozedurName“ method=glqq XXX“>. Dabei steht `prozedurName` für den Namen des Programms, das das Formular auswerten soll; `XXX` steht für die Art, wie die Feldinhalte des Formulars an das Programm weitergegeben werden. Es gibt die beiden Methoden `GET` und `POST`. Auf die Unterschiede der beiden Methoden soll hier nicht weiter eingegangen werden. Die Methode `POST` ist aber aus Sicherheitsgründen vorzuziehen.

Ein Formular muss natürlich neben Text auch noch Eingabefelder enthalten. Eingabefelder werden durch das tag <INPUT> realisiert. Es gibt verschiedene Typen von Eingabefelder, von denen hier zwei vorgestellt werden.

Auf jeden Fall sollte ein Formular einen Knopf enthalten, der das Absenden des Formulars bewirkt. Das ist ein spezielles Eingabe-Feld, das durch den tag <INPUT type=glqq submit“ name=glqq absenden“> erzeugt wird, wobei der Text des Attributs `name` natürlich frei gewählt werden kann. Der `type` mit dem Wert `submit` erledigt das Absenden.

Die wahrscheinlich häufigste Art, in ein Formular Werte einzugeben, ist über ein Textfeld, in das man mit der Tastatur Text oder Zahlen schreibt. Das geschieht mit dem tag <INPUT type=glqq text“ name= „a“ value=glqq “>. Das Attribut mit dem Wert `type=\grqq\/text` gibt an, dass hier ein Texteingabefeld geöffnet wird. Damit das weiterverarbeitende Programm auch weiß, welche Variable mit dem in diesem Eingabefeld eingegebenen Wert belegt werden soll, muss dem Textfeld noch der Name einer Variablen angegeben werden; dies geschieht durch das Attribut

```
text=\grqq\/ xxx”
```

.

Sofern man dem Texteingabefeld einen Startwert (den man dann überschreiben kann) mitgeben will, wird der dem Attribut

```
value=\grqq\/ ”
```

zugewiesen.

Da einem Eingabefeld in einem Formular ein Vortext vorangestellt werden sollte, damit der Benutzer weiß, was er überhaupt eingeben sollte, sehen Formulare erst einmal zerfleddert aus. Das kann man glätten, indem man das Formular in eine Tabelle einbettet.

22.3. 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.

22.4. 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:

Listing 22.1: HTML-Seite „Hallo world“

```
1
2  <html>
3    <head>
4      <title>erste HTML-Seite</title>
5    </head>
6    <body>
7      <p>hello world</p>
8    </body>
9  </html>
```

So weit, so gut, aber das soll ja keine Anleitung für HTML sein, und das ist eine einfache HTML-Seite.

22.5. 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:

Listing 22.2: Header für dynamische HTML-Seiten

```
1  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.

Listing 22.3: Die HTML-Seite als String

```
1   seite = """
2  <html>
3    <head>
4      <title>erste HTML-Seite</title>
5    </head>
6    <body>
7      <p>hello world</p>
8    </body>
9  </html>  """
```

Diese Variable wird dann nach der Header-Zeile ausgegeben.

Listing 22.4: 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.

22.6. Ein eigener Mini-Webserver

Dieser Webserver ist übernommen aus M. Lutz, [31]. 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.

Listing 22.5: Ein Web-Server in Python

```
1  #!/usr/bin/python
2  import os, sys
3  from BaseHTTPServer import HTTPServer
4  from CGIHTTPServer import CGIHTTPRequestHandler
5
6  webdir = '?????'  ## webdir = './wwwDateien'
7  port = 80         ## oder ein anderer Port
8
9  if sys.platform[:3] == 'win':
10   CGIHTTPRequestHandler.have_popen2 = False
11   CGIHTTPRequestHandler.have_popen3 = False
12   sys.path.append('cgi-bin')
13
14 os.chdir(webdir)
15 srvaddr = ("", port)
16 srvobj = HTTPServer(srvaddr, CGIHTTPRequestHandler)
17 srvobj.serve_forever()
```

So ein schön kurzes Programm, aber es reicht wirklich als Webserver für den Hausgebrauch.

22.7. 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.

22.7.1. 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.

Listing 22.6: For-Schleife in einer dynamischen HTML-Seite

```
1
2  #!/usr/bin/python
3
4  seite = '''
5  <html>
6    <head>
7      <title>FOR-Schleife</title>
8    </head>
9    <body>
10     %s
11   </body>
12 </html>'''
13
14 zeile = "Ich bin eine for-Schleife und gerade in Durchlauf %d<br/>"
15 text = ''
16
17 for i in range(5):
18   text += zeile % i
19
20 print("Content-type: text/html\n\n")
21
22 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.

22.7.2. . . . 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.

Listing 22.7: For-DOWNTO-Schleife in einer dynamischen HTML-Seite

```
1
2  #!/usr/bin/python
3
4  seite = '''
5  <html>
6    <head>
7      <title>FOR-DOWNTO-Schleife</title>
8    </head>
9    <body>
10     %s
11   </body>
12 </html>
13 '''
14
15 zeile = "Ich bin eine FOR DOWNTO Schleife
16          und bin gerade beim Zaehler %d<br/>"
17 text = ''
18
19 for i in range(6, 0, -1):
20   text += zeile % i
21
22 print("Content-type: text/html\n\n")
23
24 print(seite % text)
```

Ich bin eine FOR DOWNTO Schleife und bin gerade beim Zaehler 6
Ich bin eine FOR DOWNTO Schleife und bin gerade beim Zaehler 5
Ich bin eine FOR DOWNTO Schleife und bin gerade beim Zaehler 4
Ich bin eine FOR DOWNTO Schleife und bin gerade beim Zaehler 3
Ich bin eine FOR DOWNTO Schleife und bin gerade beim Zaehler 2
Ich bin eine FOR DOWNTO Schleife und bin gerade beim Zaehler 1

22.7.3. 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.

Listing 22.8: Das kleine 1 mal 1 auf einer dynamischen HTML-Seite

```
1
2  #!/usr/bin/python
3
4   headerZeile = "Content-type: text/html\n\n"
5
6   seite = '''
7  <html>
8    <head>
9      <title>Kleines 1 x 1</title>
10   </head>
11   <body>
12   %s
13   </body>
14 </html>
15 '''
16
17  def klEinMalEins():
18    klEinMalEinsString = ''                  # eine leere Zeichenkette wird
19                                             # bereit gestellt
20    for z in range(1, 11):
21      for s in range(1, 11):
22        klEinMalEinsString += str(z*s) + ' '  # die nächste Zahl wird nach einem
```

```
23                                                  # Leerzeichen eingefügt
24          klEinMalEinsString += '<br/>'           # die Zeile wird mit einem
25                                                  # Zeilenvorschub beendet
26      return klEinMalEinsString                   # die Zeichenkette wird zurück-
27                                                  # gegeben
28                                                  # Ende der Funktion
29  print(headerZeile)                              # Header ausgeben
30
31  print(seite % klEinMalEins())                   # Seite ausgeben
```

Das funktioniert. Aber es sieht wirklich nicht schön aus, das muss man wohl zugeben.

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

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.

Listing 22.9: Das kleine 1 mal 1 in einer Tabelle (dynamische HTML-Seite)

```
1  #!/usr/bin/python
2
3  headerZeile = "Content-type: text/html\n\n"
4
5  seite = '''
6  <html>
7    <head>
8      <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

22.8. 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.

Listing 22.10: Eingabeformular in HTML

```

<html>
  <head>
    <title>Eingabe</title>
  </head>
  <body>

    <form action="./cgi-bin/FranksAufgaben/auswerten.py" method="POST">
    <font size="4">Formulareingaben vom Besucher der Website</font>
    <br>

    Vorname:<br>
    <input type="text" name="vorname"><br>
    Nachname:<br>
    <input type="text" name="nachname"><br>
    Betrieb:<br>
    <input type="text" name="betrieb"><br>
    <br>

    <input type="Submit" value="Daten senden">
    <input type="Reset" value="Daten zur&uuml;cksetzen">
    <br>
    <br><b><font size="4">Vielen Dank!</font></b><br>

    </form>
  </body>
</html>
```

Das sieht dann so aus:

[2]Das Leben des Brian

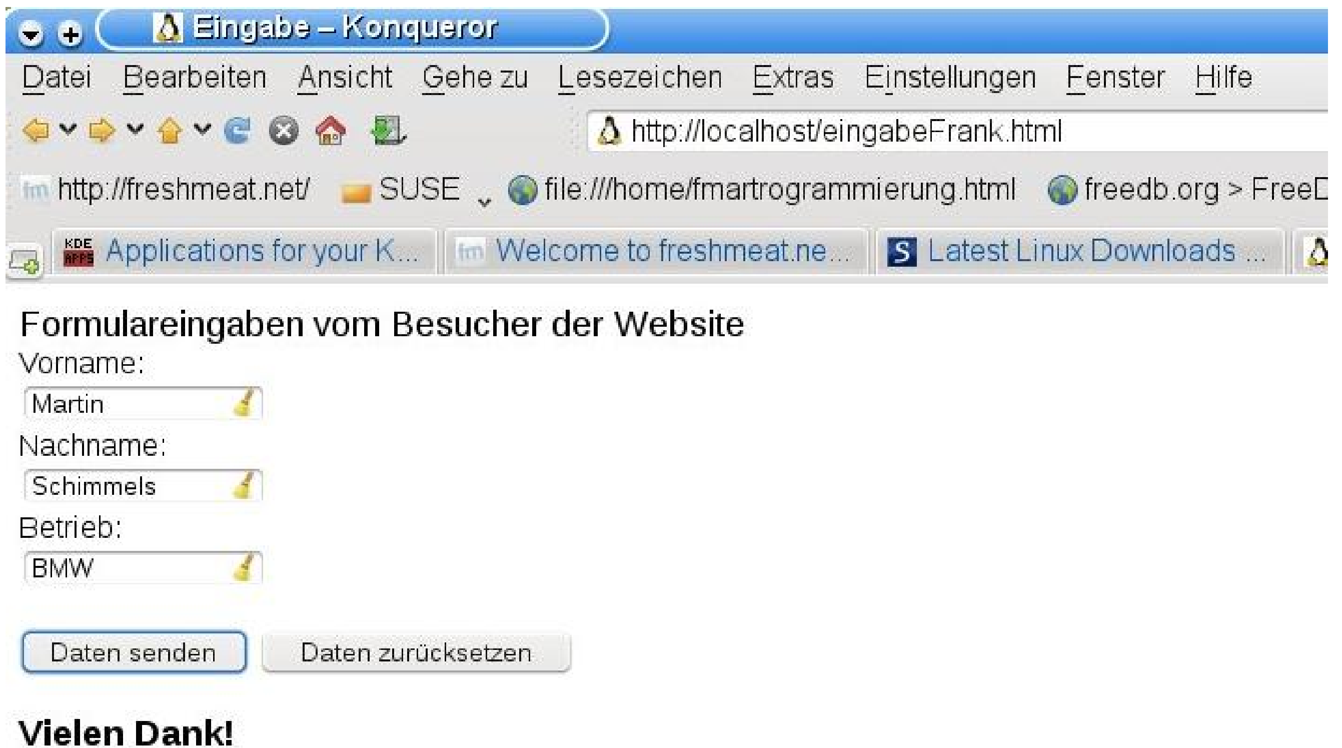


Abbildung 22.1.: Eingabe-Formular in HTML(Namen)

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.

Listing 22.11: Auswertung des Eingabeformulars(Namen)

```
1
2 #!/usr/bin/python
3 # -*- coding: utf-8 -*-
4
5 import cgi
6 formularFelder = cgi.FieldStorage()
7
8 seite = '''
9 <html>
```

```
10      <head>
11        <title>Formularauswertung</title>
12      </head>
13      <body>
14        Auswertung eines Formulars
15        <br>
16        <br>
17        %s
18      </body>
19    </html>'''
20
21  zeilen = ' ... folgende Daten wurden übermittelt:<br><br>'
22  zeilen += 'Vorname:   '+formularFelder['vorname'].value+'<br>'
23  zeilen += 'Nachname:  ' +formularFelder['nachname'].value+'<br>'
24  zeilen += 'Betrieb:   '+formularFelder['betrieb'].value+'<br>'
25
26  print("Content-type: text/html\n\n")
27  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:

```
1 formularFelder = {'vorname':'Martin', 'nachname':'Schimmels',
2          'betrieb':'BMW'}
```

Auswertung eines Formulars

... folgende Daten wurden übermittelt:

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`

22.9. 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 22.2.: Eingabe-Formular für eine Rechnung in HTML

Der dazugehörende HTML-Code ist im nächsten Bild zu sehen:

Listing 22.12: Eingabeformular für Rechnung

```

<html>

  <head>
    <title>Rechnung Eintritt Sport</title>
    <meta name="generator" content="Bluefish 2.2.3" >
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  </head>
  <body>
    <h1>Eintrittskarten für Sportveranstaltung</h1>
    <p>
      <form action="./cgi-bin/FranksAufgaben/rechnSchreiben.py"
              method="POST">
        <br>
        <table>
          <tr>
            <td>Anzahl Erwachsene:</td>
            <td><input type="text" name="erw"></td>
          </tr>
          <tr>
            <td>Anzahl Kinder:</td>
            <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>
```

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.

Listing 22.13: Klasse Rechnung

```
1   #!/usr/bin/python
2
3   class Rechnung():
4     def __init__(self, anzErw=0, anzKids=0, anzSen=0, anzStud=0):
5       self.anzErw = anzErw
6       self.anzKids = anzKids
7       self.anzSen = anzSen
8       self.anzStud = anzStud
9       self.normalPreis = 10.0
10      self.ermKids = 0.5
11      self.ermStud = 0.3
12      self.ermSen = 0.4
13
14    def rechnungSchreiben(self):
15      self.rechnBetrag = self.anzErw * self.normalPreis
16                        + self.anzKids * self.normalPreis*self.ermKids
17                        + self.anzSen*self.normalPreis*self.ermSen
18                        + self.anzStud*self.normalPreis*self.ermStud
19
```

```
20      def rechnungAnzeigen(self):
21        print('Rechnungsbetrag: '+ str(self.rechnBetrag))
22
23    if __name__ == '__main__':
24      meineRechnung = Rechnung(3, 2, 4, 1)
25      meineRechnung.rechnungSchreiben()
26      meineRechnung.rechnungAnzeigen()
```

Nachdem der Test erfolgreich war, wird das Skript, das die Auswertung durchführt, geschrieben. Das sieht so aus:

Listing 22.14: Auswertung des Eingabeformulars Rechnung

```
1
2   #!/usr/bin/python
3
4   import cgi
5   formularFelder = cgi.FieldStorage()
6
7   seite = '''
8     <html>
9       <head>
10        <title>Rechnung Sportveranstaltung</title>
11      </head>
12      <body>
13        <h1>Rechnung</h1>
14        <br>
15        <br>
16        %s
17      </body>
18    </html>'''
19
20
21  from clRechnungEinfach import Rechnung
22  try:
23    erw = int(formularFelder['erw'].value)
24  except:
25    erw = 0
26  try:
27    kind = int(formularFelder['kind'].value)
28  except:
29    kind = 0
30  try:
31    sen = int(formularFelder['sen'].value)
32  except:
33    sen = 0
34  try:
35    stud = int(formularFelder['stud'].value)
36  except:
37    stud = 0
```

```
    neueRechnung = Rechnung(erw, kind, sen, stud)
    neueRechnung.rechnungSchreiben()

    print("Content-type: text/html\n\n")

    print(seite % str(neueRechnung.rechnBetrag))
```

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 22.3.: 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.

Listing 22.15: Klasse Rechnung (verbessert)

```

#!/usr/bin/python

class Rechnung():
    def __init__(self, anzErw=0, anzKids=0, anzSen=0, anzStud=0):
        self.anzErw = anzErw
        self.anzKids = anzKids
        self.anzSen = anzSen
        self.anzStud = anzStud
        self.normalPreis = 10.0
        self.ermKids = 0.5
        self.ermStud = 0.3
        self.ermSen = 0.4

```

```
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        self.rechnStringHTML = '<table border="1">'
21
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
32        # Zeile f. Kinder
33        self.rechnStringHTML += '<tr>'
34        self.rechnStringHTML += '<td>Kinder:</td>
35                     <td align="right">'+str(self.anzKids)+'</td>'
36        t1 = '%4.2f' % (self.normalPreis * self.ermKids)
37        self.rechnStringHTML += '<td align="right">'+str(t1)+' Euro</td>'
38        t2 = '%4.2f' % (self.normalPreis * self.ermKids * self.anzKids)
39        self.rechnStringHTML += '<td align="right">'+str(t2)+' Euro</td>'
40        self.rechnStringHTML += '</tr>'
41
42        # Zeile f. Senioren
43        self.rechnStringHTML += '<tr>'
44        self.rechnStringHTML += '<td>Senioren:</td>
45                     <td align="right">'+str(self.anzSen)+'</td>'
46        t1 = '%4.2f' % (self.normalPreis * self.ermSen)
47        self.rechnStringHTML += '<td align="right">'+str(t1)+' Euro</td>'
48        t2 = '%4.2f' % (self.normalPreis * self.ermSen * self.anzSen)
49        self.rechnStringHTML += '<td align="right">'+str(t2)+' Euro</td>'
50        self.rechnStringHTML += '</tr>'
51
52        # Zeile f. Studis
53        self.rechnStringHTML += '<tr>'
54        self.rechnStringHTML += '<td>Studenten:</td>
55                     <td align="right">'+str(self.anzStud)+'</td>'
56        t1 = '%4.2f' % (self.normalPreis * self.ermStud)
57        self.rechnStringHTML += '<td align="right">'+str(t1)+' Euro</td>'
58        t2 = '%4.2f' % (self.normalPreis * self.ermStud * self.anzStud)
59        self.rechnStringHTML += '<td align="right">'+str(t2)+' Euro</td>'
60        self.rechnStringHTML += '</tr>'
61
62        # Summen-Zeile
63        self.rechnStringHTML += '<tr> <td colspan="3">Summe:</td> <td>'
```

```
64          t1 = '%4.2f' % (self.rechnBetrag)
65          self.rechnStringHTML += str(t1)+' Euro'+ '</td> </tr>'
66          # Ende
67          self.rechnStringHTML += '</table>'
68
69      def rechnungAnzeigen(self):
70          print('Rechnungsbetrag: '+ str(self.rechnBetrag))
71
72  if __name__ == '__main__':
73      meineRechnung = Rechnung(3, 2, 4, 1)
74      meineRechnung.rechnungSchreiben()
75      meineRechnung.rechnungAnzeigen()
76      print(meineRechnung.rechnStringHTML)
```

In dem CGI-Skript muss dann nur die letzte Zeile ersetzt werden durch

Listing 22.16: Verbesserung im auswertenden Program

```
1                   print(seite % str(neueRechnung.rechnStringHTML))
```

Abbildung 22.4.: Ausgabe einer Rechnung in HTML (so kann man das akzeptieren)

22.10. 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.

Listing 22.17: Die Klasse Zähler

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  class Zaehler(object):
4      def __init__(self):
5          self.dateiOeffnen('r')
6          self.zWert = int(self.meineDat.read())
7          self.initWert = 0
8          self.dateiSchliessen()
9
10     def dateiOeffnen(self, modus):
```

```
11        self.meineDat = open("zaehlerDat",modus)
12
13    def dateiSchreiben(self, z):
14        self.meineDat.write(str(z))
15
16    def dateiSchliessen(self):
17        self.meineDat.close()
18
19    def wertZuruecksetzen(self):
20        self.dateiOeffnen('w')
21        self.zWert = self.initWert
22        self.dateiSchreiben(self.zWert)
23        self.dateiSchliessen()
24
25    def inkrement(self):
26        self.dateiOeffnen('w')
27        self.zWert = self.zWert + 1
28        self.dateiSchreiben(self.zWert)
29        self.dateiSchliessen()
30
31    def dekrement(self):
32        self.dateiOeffnen('w')
33        self.zWert = self.zWert - 1
34        self.dateiSchreiben(self.zWert)
35        self.dateiSchliessen()
```

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.

Listing 22.18: Der Besucherzähler (HTML-Teil)

```
1
2  <html>
3    <head>
4      <title>Martins Test</title>
5      <meta http-equiv="Content-type" content="text/html;
6            charset=latin1" />
7    </head>
8    <body>
9      <h1>Ein Besucherzähler</h1>
10     <p>Hallo Du da!!</p>
11     <p>Du bist der ???. Besucher</p>
12   </body>
13 </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 7.3 Link zu dieser Seite.)

Listing 22.19: Der Besucherzähler (vollständig)

```
1   #!/usr/bin/python
2
3   seite = '''
4   <html>
5     <head>
6       <title>Martins Test</title>
7       <meta http-equiv="Content-type" content="text/html;
8             charset=latin1" />
9     </head>
10    <body>
11      <h1>Ein Besucherzähler</h1>
12      <p>Hallo Du da!!</p>
13      <p>Du bist der %s . Besucher</p>
14    </body>
15  </html>
16  '''
17
18  print("Content-type: text/html\n\n")
19
20  from cl_zaehler import Zaehler
21  besucherZaehler = Zaehler()
22  besucherZaehler.inkrement()
23
24  print(seite % besucherZaehler.zWert)
```

23. Eine Kreuzung auf meiner Web-Seite

23.1. Die Ampel kommt ins Netz

Weiter oben im Kapitel 21 haben wir eine Ampel und eine Kreuzung realisiert. Nun soll daraus eine dynamische Web-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:

Listing 23.1: HTML-Code für eine Ampel

```
<html>
  <head>
    <title>Eine Ampel</title>
    <meta name="generator" content="Bluefish 2.2.3" />
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  </head>
  <body bgcolor="#C6C6C6">
    <h1>Eine Ampel</h1>
    <table width="120" cellspacing="10" border="10" cellpadding="10"
               align="center" bgcolor="white">
      <tbody>
        <tr>
          <td></td>
        </tr>
        <tr>
          <td></td>
        </tr>
        <tr>
          <td></td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

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

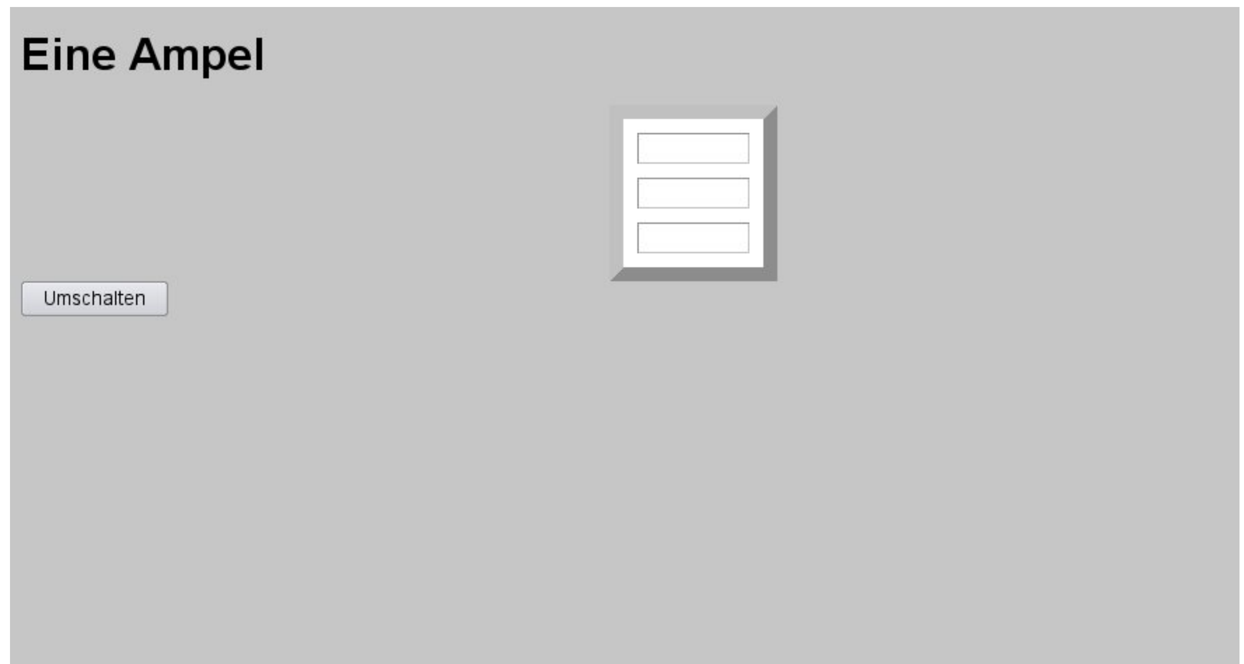


Abbildung 23.1.: Web-Seite Ampel

Lichter der Ampel umgeschaltet wurden, werden die Farben der Lichter in einen Tupel geschrieben, der mit dem %-Operator in die Variable `seite` eingefügt wird.

Listing 23.2: HTML und Python-Code für Ampel

```
#!/usr/bin/python

seite = '''
<html>
<head>
  <title>Eine Ampel</title>
</head>
<body bgcolor="#C6C6C6">
    <h1>Eine Ampel</h1>
    <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>
```

```
22     </tbody>
23   </table>
24   <FORM action="ampelHTML.py" method="POST">
25     <INPUT type="submit" name="umschalten" value="Umschalten" size="120">
26   </FORM>
27   </body>
28   </html>
29   '''
30
31   from clAmpel import Ampel
32
33   meineAmpel = Ampel()
34   meineAmpel.umschalten()
35
36   headerZeile = "Content-type: text/html\n\n"
37
38   print(headerZeile)
39   print(seite %
40        tuple(meineAmpel.zustaende[meineAmpel.zustaendeNum[meineAmpel.zustand]]))
```

23.2. 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.

Listing 23.3: HTML und Python für die Kreuzung

```
1   <html>
2     <head>
3       <title>Eine Kreuzung</title>
4       <meta name="generator" content="Bluefish 2.2.3" >
5       <meta http-equiv="Content-Type" content="text/html">
6     </head>
7     <body bgcolor="#C6C6C6">
8       <h1>Eine Kreuzung</h1>
9       <table width="720" cellspacing="10" border="10" cellpadding="10"
10              align="center" bgcolor="white">
11        <tr>
12          <td></td>
13          <td>
14            <table width="120" cellspacing="10" border="10" cellpadding="10"
15              align="center" bgcolor="white">
16              <tbody>
17                <tr>
18                  <td bgcolor="#FF0000"></td>
19                </tr>
20                <tr>
21                  <td bgcolor="#000000"></td>
22                </tr>
```

```
23                <tr>
24                  <td bgcolor="#000000"></td>
25                </tr>
26              </tbody>
27            </table>
28          </td>
29          <td></td>
30        </tr>
31        <tr>
32          <td>
33            <table width="120" cellspacing="10" border="10" cellpadding="10"
34              align="center" bgcolor="white">
35              <tbody>
36                <tr>
37                  <td bgcolor="#FF0000"></td>
38                </tr>
39                <tr>
40                  <td bgcolor="#000000"></td>
41                </tr>
42                <tr>
43                  <td bgcolor="#000000"></td>
44                </tr>
45              </tbody>
46            </table>
47          </td>
48          <td></td>
49          <td>
50            <table width="120" cellspacing="10" border="10" cellpadding="10"
51              align="center" bgcolor="white">
52              <tbody>
53                <tr>
54                  <td bgcolor="#FF0000"></td>
55                </tr>
56                <tr>
57                  <td bgcolor="#000000"></td>
58                </tr>
59                <tr>
60                  <td bgcolor="#000000"></td>
61                </tr>
62              </tbody>
63            </table>
64          </td>
65        </tr>
66        <tr>
67          <td></td>
68          <td>
69            <table width="120" cellspacing="10" border="10" cellpadding="10"
70              align="center" bgcolor="white">
71              <tbody>
```

```
72                      <tr>
73                        <td bgcolor="#FF0000"></td>
74                      </tr>
75                      <tr>
76                        <td bgcolor="#000000"></td>
77                      </tr>
78                      <tr>
79                        <td bgcolor="#000000"></td>
80                      </tr>
81                    </tbody>
82                  </table>
83                </td>
84                <td></td>
85              </tr>
86            </table>
87
88          <FORM action="ampelHTML.py" method="POST">
89            <INPUT type="submit" name="umschalten" value="Umschalten" size="120">
90          </FORM>
91          </body>
92        </html>
```

Im Browser sieht das so aus:

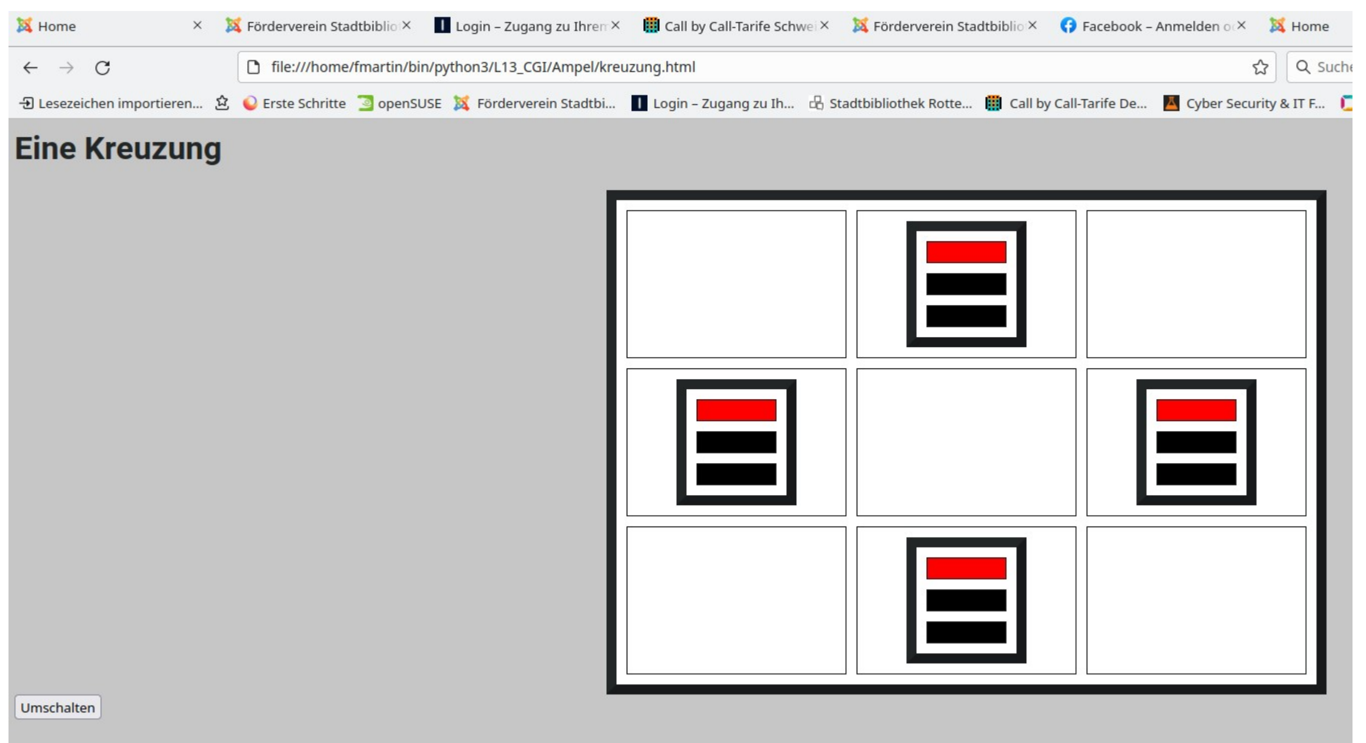


Abbildung 23.2.: Kreuzung im Browser

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.

Listing 23.4: Die Kreuzung

```
1  #!/usr/bin/python
2
3  seite = '''
4  <html>
5    <head>
6      <title>Eine Kreuzung</title>
7    </head>
8    <body bgcolor="#C6C6C6">
9      <h1>Eine Kreuzung</h1>
10     <table width="720" cellspacing="10" border="10" cellpadding="10"
11             align="center" bgcolor="white">
12       <tbody>
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="%s">o</td>
21               </tr>
22               <tr>
23                 <td bgcolor="%s">o</td>
24               </tr>
25               <tr>
26                 <td bgcolor="%s">o</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="%s">o</td>
40                </tr>
41                <tr>
42                  <td bgcolor="%s">o</td>
43                </tr>
44                <tr>
45                  <td bgcolor="%s">o</td>
46                </tr>
47              </tbody>
48            </table>
49          </td>
50
51          <td></td>
52          <td>
53            <table width="120" cellspacing="10" border="10" cellpadding="10"
54                          align="center" bgcolor="white">
55              <tbody>
56                <tr>
57                  <td bgcolor="%s">o</td>
58                </tr>
59                <tr>
60                  <td bgcolor="%s">o</td>
61                </tr>
62                <tr>
63                  <td bgcolor="%s">o</td>
64                </tr>
65              </tbody>
66            </table>
67          </td>
68        </tr>
69        <tr>
70          <td></td>
71          <td>
72            <table width="120" cellspacing="10" border="10" cellpadding="10"
73                         align="center" bgcolor="white">
74              <tbody>
75                <tr>
76                  <td bgcolor="%s">o</td>
77                </tr>
78                <tr>
79                  <td bgcolor="%s">o</td>
80                </tr>
81                <tr>
82                  <td bgcolor="%s">o</td>
```

```
83                  </tr>
84                </tbody>
85              </table>
86            </td>
87            <td></td>
88          </tr>
89          </tbody>
90        </table>
91
92
93    <FORM action="kreuzungHTML.py" method="POST">
94      <INPUT type="submit" name="umschalten" value="Umschalten" size="120">
95    </FORM>
96    </body>
97  </html>
98  '''
99
100 from clKreuzung import Kreuzung
101
102 meineKreuzung = Kreuzung()
103 meineKreuzung.umschalten()
104
105 zustandsListe = []
106 for einZustand in meineKreuzung.nordAmpel.zustaende
107       [meineKreuzung.nordAmpel.zustaendeNum[meineKreuzung.nordAmpel.zustand]]:
108   zustandsListe.append(einZustand)
109 for einZustand in meineKreuzung.westAmpel.zustaende
110       [meineKreuzung.westAmpel.zustaendeNum[meineKreuzung.westAmpel.zustand]]:
111   zustandsListe.append(einZustand)
112 for einZustand in meineKreuzung.ostAmpel.zustaende
113       [meineKreuzung.ostAmpel.zustaendeNum[meineKreuzung.ostAmpel.zustand]]:
114   zustandsListe.append(einZustand)
115 for einZustand in meineKreuzung.suedAmpel.zustaende
116       [meineKreuzung.suedAmpel.zustaendeNum[meineKreuzung.suedAmpel.zustand]]:
117   zustandsListe.append(einZustand)
118 #print(zustandsListe)
119 headerZeile = "Content-type: text/html\n\n"
120
121 print(headerZeile)
122
123 print(seite % tuple(zustandsListe))
```

24. Programme mit grafischer Oberfläche

The empty handed painter from your streets
Is drawing crazy patterns on your sheets

(Bob Dylan[1])

24.1. 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.

Anmerkung

Unter Python3 wird tk eingebunden durch `import tkinter`. Unter Python2 wurde tk eingebunden durch `import Tkinter`.

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.

[1] It’s all over now, Baby Blue *auf:* Bringing it all back home

24.2. 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:

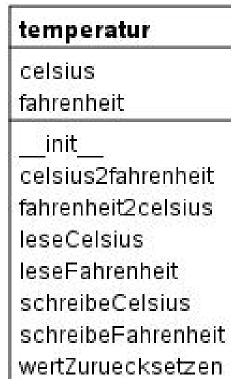


Abbildung 24.1.: Klassendiagramm `Temperatur`

In Python sieht der dazugehörige Quellcode so aus:

Listing 24.1: Klassendefinition: Temperaturen

```
1   #!/usr/bin/python
2   # -*- coding: utf-8 -*-
3
4   class Temperatur():
5     def __init__(self):
6           self.celsius = 0.001
7           self.fahrenheit = 0.002
8
9     def leseCelsius(self):
10          self.celsius = float(input("Temperatur in
11                                      Grad Celsius eingeben: "))
12
13    def leseFahrenheit(self):
14          self.fahrenheit = float(input("Temperatur in
15                                      Grad Fahrenheit eingeben: "))
16
17    def celsius2fahrenheit(self):
18          self.fahrenheit = 9.0 / 5.0 * self.celsius + 32.0
19
20    def schreibeFahrenheit(self):
```

```
21          print("Temperatur in Grad Fahrenheit: ",self.fahrenheit)
22
23      def fahrenheit2celsius(self):
24          self.celsius = 5.0 / 9.0 * (self.fahrenheit - 32)
25
26      def schreibeCelsius(self):
27          print("Temperatur in Grad Celsius: ",self.celsius)
28
29      def wertZuruecksetzen(self):
30          self.celsius = 0.01
31          self.fahrenheit = 0.02
```

Wie immer gehört dazu ein aufrufendes Programm, das wieder einmal in einer While-Schleife Eingaben ermöglicht und dann diese bearbeitet.

Listing 24.2: Aufruf der Klasse Temperatur

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  from cl_Temperatur import Temperatur
4
5  modus = 'x'
6  t1 = temperatur()
7  while modus != 'E':
8    modus = input('\nAuswahl: \nCelsius nach Fahrenheit mit C\n
9          Fahrenheit nach Celsius mit F\nEnde mit E: ')
10   if modus == 'C':
11     t1.leseCelsius()
12     t1.celsius2fahrenheit()
13     t1.schreibeFahrenheit()
14   elif modus == 'F':
15     t1.leseFahrenheit()
16     t1.fahrenheit2celsius()
17     t1.schreibeCelsius()
18   elif modus == 'E':
19     pass
20   else:
21     print('Falsche Eingabe. Nur C, F, E erlaubt')
22
23 print('Ciao')
```

24.3. . . . 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`

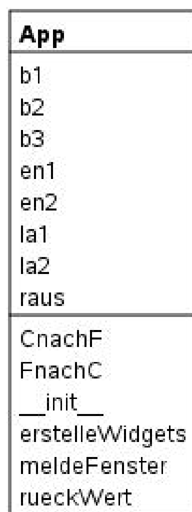


Abbildung 24.2.: Klassendefinition: GUI für die Temperatur

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.

Listing 24.3: Die Klasse für das Fenster

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  from Tkinter import *
4  from Tkconstants import *
5  from cl_Temperatur import temperatur
6  import tkMessageBox
7
8  t1 = temperatur()
9
10 class App(Frame):
11   def __init__(self, master = None):
12       Frame.__init__(self, master)
13       self.master.title('Temperatur-Rechner')
14       self.grid(padx = 10, pady = 10)
15       self.erstelleWidgets()
16
17   def erstelleWidgets(self):
```

```
18          self.raus = Button(self, text="Ende", bg="#C1281F",
19                          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
60      def CnachF(self):
61          t1.celsius = float(self.en1.get())
62          t1.celsius2fahrenheit()
63          self.en2.delete(0, END)
64          self.en2.insert(0,t1.fahrenheit)
65
66      def FnachC(self):
```

```
67          t1.fahrenheit = float(self.en2.get())
68          t1.fahrenheit2celsius()
69          self.en1.delete(0, END)
70          self.en1.insert(0,t1.celsius)
```

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!

Listing 24.4: Aufruf des Fensters

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  from cl_Gui import *
4  import tkMessageBox
5
6  def callback():
7    if tkMessageBox.askokcancel('Quit', 'Wirklich aufhören?'):
8        root.destroy()
9
10 if __name__ == "__main__":
11   root = Tk()
12   meineTemp = App(root)
13   root.protocol('WM_DELETE_WINDOW', callback)
14   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 24.3.: Temperaturprogramm

Und das oben erwähnte Zusatzfenster, mit dem man das Programm endgültig beendet, erscheint so:

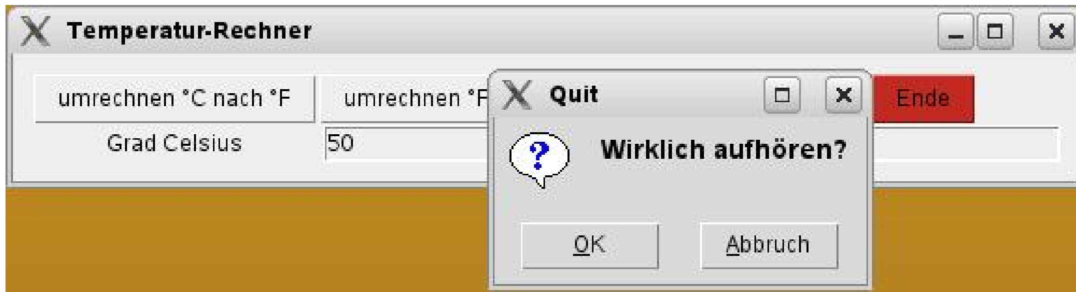


Abbildung 24.4.: Beendigung des Temperaturprogramms

Teil XII.

Python für Verwaltungsaufgaben

25. Python als Sprache für den eigenen Rechner

25.1. 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:

Listing 25.1: Versuche mit `sys`

```
>>> import sys
>>>
>>> print(sys.platform)
linux2
>>> print(sys.version)
3.4.6 (default, Mar 22 2017, 12:26:13) [GCC]
>>> print(sys.copyright)
Copyright (c) 2001-2017 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.

Listing 25.2: 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 Informationen über einen Modul abgefragt werden können, nämlich mit **help(sys)**. Eine Inhaltsangabe des Moduls `sys` erhält man durch

Listing 25.3: Hilfe zum Modul `sys`

```
>>> dir(sys)
['__displayhook__', '__doc__', '__egginsert', '__excepthook__',
'__interactivehook__', '__loader__', '__name__', '__package__',
'__plen', '__spec__', '__stderr__', '__stdin__', '__stdout__',
'_clear_type_cache','_current_frames', '_debugmallocstats',
'_getframe', '_home', '_mercurial','_xoptions', 'abiflags',
'api_version', 'arch', 'argv', 'base_exec_prefix','base_prefix',
'builtin_module_names', 'byteorder', 'call_tracing','callstats',
'copyright', 'displayhook', 'dont_write_bytecode','exc_info',
'excepthook', 'exec_prefix', 'executable', 'exit', 'flags',
'float_info', 'float_repr_style', 'getallocatedblocks',
'getcheckinterval','getdefaultencoding', 'getdlopenflags',
'getfilesystemencoding', 'getprofile','getrecursionlimit',
'getrefcount', 'getsizeof', 'getswitchinterval','gettrace',
'hash_info', 'hexversion', 'implementation', 'int_info','intern',
'last_traceback', 'last_type', 'last_value', 'lib', 'maxsize',
'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
'path_importer_cache', 'platform', 'prefix', 'setcheckinterval',
'setdlopenflags', 'setprofile', 'setrecursionlimit',
'setswitchinterval','settrace', 'stderr', 'stdin', 'stdout',
'thread_info', 'version','version_info', 'warnoptions']
```

Was ist denn die Standard-Kodierung für Dateien in unserem System?

Listing 25.4: Standardkodierung abfragen

```
>>> sys.getfilesystemencoding()
'UTF-8'
```

Wichtiges Element von `sys` ist `exit`. Damit kann man einen Programmlauf beenden. Das ist die saubere Art, ein Programm zu beenden, wenn man ein Menu für das Programm geschrieben, in dem durch Eingabe von `Ende` das Programm beendet werden soll.

25.2. Verzeichnisse und Dateien

25.2.1. Die Grundlagen

Wie schon weiter oben gesehen, besteht die Arbeit mit Dateien aus 3 Schritten:

1. Öffnen der Datei
2. Arbeiten in der Datei
3. Schließen der Datei

Beim Öffnen musste man bereits angeben, was man mit der Datei machen will: Lesen oder Schreiben.

Python hat sich seit den Anfängen enorm weiterentwickelt, und deswegen gibt es in Python jetzt „Iteratoren“ , das heißt Methoden, die über eine Datei schleifen. Das erleichtert die Arbeit mit Dateien. Dabei wird eine Datei als eine Liste von Zeilen gesehen, die man mit einer `for`-Schleife abarbeiten kann. Python verwaltet die Struktur selbständig.

Die einfachste Möglichkeit ist dabei,

Listing 25.5: Bearbeiten einer Datei mit Iterator

```
1 >>> datei = open('blibla.blub', 'r')
2 >>> for zeile in datei:
3         print(zeile)
4 das
5 sind
6 alle
7 Zeilen
8 >>> datei.close()
```

Python stellt aber noch weitere Iteratoren zur Verfügung. Mit `readline` wird eine Zeile nach der anderen gelesen. Am Ende der Datei wird von der Methode `readline` eine leere Zeichenkette zurückgegeben. Das ist für die Weiterverarbeitung unter Umständen ein kleines Problem, aber es funktioniert.

Listing 25.6: Zeilenweises Lesen einer Datei mit `readline`

```
1 >>> datei = open('blibla.blub', 'r')
2 >>> print(datei.readline())
3 das
4 >>> print(datei.readline())
5 sind
6 >>> print(datei.readline())
7 alle
8 >>> print(datei.readline())
9 Zeilen
10 >>> print(datei.readline())
11
12 >>> datei.close()
```

Ein Programm, das eine gesamte Datei zeilenweise liest, könnte so aussehen:

Beispiel 25.1 Programm für zeilenweises Lesen einer Datei mit `readline`

```
1  #!/usr/bin/python3
2  import sys
3
4  datName = "sinnvolleDatei"
5
6  datW = open(datName, 'w')
7  for i in range(1, 11):
8      datW.write('Zeile '+str(i)+' der Datei'+'\n')
9  datW.close()
10
11 datR = open(datName, 'r')
12 while True:
13     zeile = datR.readline()[:-1]
14     if not zeile:
15         sys.exit()
16     print(zeile)
```

Wichtig ist hier, dass das Ergebnis von `readline` in einer Variablen gespeichert wird, und danach gefragt wird, ob `readline` wirklich etwas liefert; wenn nicht wird die Schleife verlassen.

25.2.2. Das Modul `os`

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.

Wichtige Befehle im Zusammenhang mit jedem Betriebssystem sind solche, die sich mit dem Verzeichnisbaum beschäftigen. Zuerst lasse ich das aktuelle Verzeichnis anzeigen, danach wechsele ich im Verzeichnisbaum eine Stufe hinauf:

Listing 25.7: Aktuelles Verzeichnis und Verzeichniswechsel (Unix)

```
1 >>> import os
2 >>> os.getcwd()
3 '/home/martin/texte'
4 >>> os.path.abspath('./texte')
5 '/home/fmartin/texte'
6 >>> os.chdir('..')
7 >>> os.getcwd()
8 '/home/martin'
```

Interessant dabei ist `os.path.abspath`: dieser Befehl wandelt eine relative Pfad-Angabe in einen absoluten Pfad um.

Im vorigen Absatz sind die Ausgaben auf einem typischen Unix(Linux)-Rechner zu sehen. Unter Windows sähe das vielleicht so aus:

Listing 25.8: Aktuelles Verzeichnis und Verzeichniswechsel (Windows

```
>>> import os
>>> os.getcwd()
'c:\\home\\martin\\texte'
>>> os.chdir('..')
>>> os.getcwd()
'c:\\home\\martin'
```

Interessant an dem Modul `os` ist, dass man auch aus Python heraus damit Befehle des Betriebssystems aufrufen kann. Dazu dient die Methode `system`. Ich lasse also hier einmal alle Dateien des aktuellen Verzeichnisses ausgeben. Unter Linux dient dazu der Befehl **ls**, um eine ausführliche Darstellung zu erhalten in der Variante **ls -lsia**

Listing 25.9: Dateien des aktuellen Verzeichnisses

```
>>> import os
>>> os.system('ls')
adressen.sql  plzort.sql  schueler.sql  tatorte.sql  test.sql  vbleme.sql
warenhaus.sql
0
>>> os.system('ls -lsia')
insgesamt 2900
1867970    4 drwxr-xr-x 2 fmartin users    4096 26. Aug 17:35 .
1867903    4 drwxr-xr-x 5 fmartin users    4096 26. Aug 17:32 ..
1841285  152 -rw-r--r-- 1 fmartin users  155404 26. Aug 17:35 adressen.sql
1864164 2228 -rw-r--r-- 1 fmartin users 2280472 26. Aug 17:35 plzort.sql
1864159  132 -rw-r--r-- 1 fmartin users  135013 26. Aug 17:33 schueler.sql
1864165   84 -rw-r--r-- 1 fmartin users   84142 26. Aug 17:35 test.sql
1864162  184 -rw-r--r-- 1 fmartin users  186444 26. Aug 17:35 warenhaus.sql
0
```

Im vorletzten Beispiel habe ich die unterschiedliche Darstellung des Verzeichnisbaumes unter Unix und Windows gezeigt. Während unter Unix (und allen Abkömmlingen) das Trennzeichen zwischen Verzeichnissen oder zwischen Verzeichnis und Datei der einfache Slash `/` ist, benutzt Windows den Backslash `\`. Leider schert Windows nicht nur bei diesem Trennzeichen aus der Reihe, sondern auch bei anderen Elementen des Verzeichnisbaums und der Darstellung von Dateinamen. Wenn man mit Hilfe eines Programms also auf Datei- und Verzeichnisebene arbeiten und sein Programm portabel halten will, muss man beide (oder vielleicht noch mehr?) Möglichkeiten berücksichtigen. Auch hier hilft das Modul `os`: es kennt die Variablen

- `sep` für das Trennzeichen zwischen Verzeichnissen,
- `pardir` für das übergeordnete Verzeichnis (parent directory)

- `curdir` für das aktuelle Verzeichnis (current directory)

Ein interessanter Ast des Moduls `os` ist `os.path`. Hierin sind Attribute und Methoden enthalten, die sich mit dem Dateipfad innerhalb des Verzeichnisbaumes beschäftigen. Als Beispiel benutze ich die Methoden `basename` und `dirname`, die einen voll-qualifizierten Dateinamen in den Verzeichnis- und den Datei-Teil zerlegen:

Listing 25.10: Verzeichnisname und Dateiname

```
>>> import os
>>> dateiVollQualifiziert =
        '/home/martin/texte/PythonBuch/XML/Python-Kurs.xml'
>>> os.path.dirname(dateiVollQualifiziert)
'/home/fmartin/texte/PythonBuch/XML'
>>> os.path.basename(dateiVollQualifiziert)
'Python-Kurs.xml'
```

25.2.2.1. Beispiel: Massenumbenennung von Dateien

Eine immer wiederkehrende Aufgabe: manche Programme speichern eine Bilddatei mit der Endung `.jpg`, andere mit der Endung `.JPG`, und beide Varianten gibt es auch noch mit einem `e` zwischen `p` und `g`. Das hätte ich gerne einheitlich und zwar so, dass alle diese Dateinamen auf `jpg` enden. `os` enthält eine Methode `splitext`, kurz für „split extension“ , die ein Tupel zurückgibt, dessen 0. Eintrag der Dateiname und dessen 1. Eintrag die Erweiterung ist.

Listing 25.11: Dateiendungen vereinheitlichen

```
for eineDatei in glob.glob('*'):
    if eineDatei.endswith('.JPG','.JPEG','.jpeg'):
        nD = os.path.splitext(eineDatei)[0]+'.jpg'
        os.system('mv '+ eineDatei+' '+ nD)
```

25.2.3. Modul subprocess

Für den Aufruf von Betriebssystem-Befehlen wird empfohlen, `os.system` nicht mehr zu benutzen, sondern das Modul `subprocess`. Dazu muss entweder das gesamte Modul `subprocess` oder nur die daraus benutzten Teile importiert werden. Bei den folgenden Beispielen wurden nur Teile importiert. Für die vielen Möglichkeiten, die `subprocess` bietet, empfiehlt es sich, die Dokumentation zu lesen. Hier sollen nur einige Beispiele gezeigt und erläutert werden. Zuerst sollen wieder die Dateien eines Verzeichnisses in Langform angezeigt werden.

Listing 25.12: ls mit `subprocess.Popen`

```
>> with Popen(['ls', '-lsia'], stdout=PIPE, universal_newlines=True)
            as pc:
        langeListe = pc.stdout.read()
```

```
4
5 >>> for zeile in langeListe.split('\n'):
6         print(zeile)
```

Mit dem ersten Befehl wird der selbe Befehl wie im vorigen Beispiel aufgerufen (nur in einem anderen Verzeichnis). Die Argumente müssen in einer Liste, durch Kommata voneinander getrennt, den subprocess-Befehlen, hier `Popen` mitgegeben werden. Die Ausgabe wird auf `PIPE` umgeleitet. Mit dem Parameter `universal_newlines = True` wird die Ausgabe von `Popen` als Zeichenkette weitergegeben. In der zweiten Zeile des ersten Befehls wird dann diese Pipe gelesen und in der Variablen `langeListe` gespeichert. Mit dem zweiten Befehl wird der Inhalt von `langeListe` an den Zeilenvorschüben (`'\n'`) getrennt und dann zeilenweise ausgegeben. So sieht die Ausgabe aus:

Listing 25.13: Ausgabe von ls mit `subprocess.Popen`

```
1 insgesamt 132
2 7482317 4 drwxr-xr-x  3 fmartin users 4096 16. Jan 2020  .
3 7482062 4 drwxr-xr-x 59 fmartin users 4096 19. Sep 16:05  ..
4 7496430 4 -rwxr--r--  1 fmartin users  504 24. Sep 2012  anhalteweg2.py
5 7498056 4 -rwxr--r--  1 fmartin users  633 24. Sep 2012  anhalteweg3.py
```

Listing 25.14: Ausgabe von ls mit `subprocess.run` und ohne `read`

```
1 >>> listPgm = run(['ls','-lha'], stdout=PIPE, universal_newlines=True)
2 >>> ausgabe = listPgm.stdout
3 >>> print(ausgabe)
4 insgesamt 132
5 7482317 4 drwxr-xr-x  3 fmartin users 4096 16. Jan 2020  .
6 7482062 4 drwxr-xr-x 59 fmartin users 4096 19. Sep 16:05  ..
7 7496430 4 -rwxr--r--  1 fmartin users  504 24. Sep 2012  anhalteweg2.py
8 7498056 4 -rwxr--r--  1 fmartin users  633 24. Sep 2012  anhalteweg3.py
```

Einfacher geht das mittels `check_output` aus `subprocess`.

Listing 25.15: ls mit `subprocess.check_output`

```
1 >>> lsAusgabe = check_output(["ls","-lha"],universal_newlines=True)
2 >>> print(lsAusgabe)
3 insgesamt 132K
4 drwxr-xr-x  3 fmartin users 4,0K 16. Jan 2020  .
5 drwxr-xr-x 59 fmartin users 4,0K 19. Sep 16:05  ..
6 -rwxr--r--  1 fmartin users  504 24. Sep 2012  anhalteweg2.py
7 -rwxr--r--  1 fmartin users  633 24. Sep 2012  anhalteweg3.py
```

Das Modul `subprocess` erzeugt einen neuen Unterprozess. Vorteil gegenüber dem Starten mittels `os.system()` ist, dass man diesen Prozess gut behandeln kann. Hier soll das gezeigt werden anhand einer Diashow (mit Hilfe des Grafikprogramms `xv`):

Listing 25.16: Diashow mit `subprocess`

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

```
2
3 import subprocess
4 import os
5 from time import sleep
6
7 verzeichnis = input('Welches Verzeichnis? ')
8
9 import glob
10 bildListe = glob.glob(verzeichnis+'/*.jpg')
11 for bild in bildListe:
12     x = subprocess.Popen(['xv', bild])
13     sleep(5)
14     os.system("kill " + str(x.pid))
```

Zum Anzeigen der einzelnen Bilder wird hier das Programm `xv` benutzt. Wem das nicht gefällt, der kann das einfach ändern. Außerdem ist fest verdrahtet, dass nur Dateien mit der Endung `jpg` (klein geschrieben) angezeigt werden.

25.3. Zeit und Datum

25.3.1. Das Modul `time`

Zeit, und auch damit Datum, wird in Unix-Betriebssystemen als Dezimalzahl geführt. Während ich das schreibe, ist es 1681996314.412224. Das habe ich dadurch erfahren, dass ich auf der Python-Shell

```
1 >>> import time
2 >>> time.time()
3 1681996314.412224
```

eingegeben habe. Wie ist das zu verstehen? Die Zahl gibt die Anzahl der Sekunden an, die seit dem Zeitpunkt 0 vergangen sind.

Was sollen aber 6 Nachkommastellen: das ist für das tägliche Leben natürlich nicht angemessen: was fange ich damit an, dass ich jetzt weiss, dass es 412224 Millionstel-Sekunden nach einer bestimmten Zeit an einem bestimmten Tag ist? Für Computer und Programme kann aber eine solch exakte Angabe relevant sein.

Nun gut. Es wäre natürlich grauenvoll, wenn jeder Programmierer eine solche Zeitangabe umrechnen müsste in eine Angabe von Tag, Monat, Jahr, Stunde, Minute, Sekunde. Also gibt es Klassen und Funktionen innerhalb von `time`, die solche Hilfsarbeiten leisten.

Zurück zum Start der Zeitrechnung (in der IT). Zeitpunkt 0 ist natürlich ein willkürlich festgelegter Zeitpunkt, nämlich der 1. Januar 1970, 0 Uhr, Greenwich-Zeit. Diese Tatsache erhält man durch

```
1 >> time.gmtime(0)
2 time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0,
3     tm_sec=0, tm_wday=3, tm_yday=1, tm_isdst=0)
```

Um eine für Menschen verständliche Zeitangabe zu bekommen, rufe ich `time.ctime` auf; `ctime` liefert eine Zeitangabe für die auf meinem Rechner gültige Zeitzone ab:

```
>>> time.ctime()
'Thu Apr 20 15:29:42 2023'
```

Damit kann jeder etwas anfangen. Allerdings wird auch immer mal wieder die Normalzeit benötigt, die Greenwich-Zeit:

```
>>> time.gmtime()
time.struct_time(tm_year=2023, tm_mon=4, tm_mday=20, tm_hour=13,
    tm_min=34, tm_sec=51, tm_wday=3, tm_yday=110, tm_isdst=0)
```

Aha, ein kleiner Unterschied! `ctime` liefert eine Zeichenkette zurück, `gmtime` ein Tupel.

Der Funktion `gmtime` kann man aber auch eine beliebige Zeit im Format von `time.time` mitgeben. Damit erhält man das obige Tupel, dessen Elemente man mit den oben genannten Schlüsselbegriffen einzeln ausgeben kann:

```
>>> t2 = time.time()
>>> t3 = time.gmtime(t2)
>>> t3.tm_year
2023

>>> t3.tm_year,t3.tm_mon,t3.tm_mday
(2023, 4, 20)
```

25.3.1.1. Zeitmessung beim Programmlauf

Wie lange braucht denn mein Programm, um 100000 Zufallszahlen zu generieren? Ähnliche Fragen stellt man sich, wenn ein Programm viele Operationen durchführen muss: der Benutzer soll ja nicht ewig vor dem Bildschirm sitzen und warten. Da teste ich doch lieber vorher.

Innerhalb des Moduls `time` gibt es da etwas richtig gutes: den `perf_counter`. Das folgende Beispiel erklärt sich von selbst.

Beispiel 25.2 Zeitmessung

```
>>> from time import perf_counter
>>> help(perf_counter)
Help on built-in function perf_counter in module time:
perf_counter(...)
    perf_counter() -> float

    Performance counter for benchmarking.
>>> import random

>>> def zZahlenErzeugen(bis):
      t_0 = perf_counter()
      zZahlen = {i:random.random() for i in range(10000)}
      print('Zeit für Erzeugung von', str(bis), 'Zufallszahlen: ',perf_counter()

>>> zZahlenErzeugen(100000)

Zeit für Erzeugung von 100000 Zufallszahlen:  0.0011585690008359961
```

25.3.2. Das Modul `datetime`

Wenn man mit Zeiten rechnen will, ist das Modul `datetime` angesagt. Es enthält die Klasse `datetime`. Hier steht, was alles zu `datetime` gehört. Vorsicht! Das Modul `datetime` enthält eine Klasse `datetime`!!

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__', 'date', 'datetime',
 'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
>>> help(datetime.datetime)
Help on class datetime in module datetime:

class datetime(date)
 |  datetime(year, month, day[, hour[, minute[, second[, microsecond[,tzinfo]]]]]
 |
 |  The year, month and day arguments are required. tzinfo may be None, or an
 |  instance of a tzinfo subclass. The remaining arguments may be ints.
```

Den Rest der Hilfe zu `datetime.datetime` habe ich hier abgeschnitten. Man sieht aber schon: ein Objekt `datetime` hat als Attribute 8 Elemente.

Ein Objekt der Klasse `datetime.datetime` hat die Methode `now`. Der Aufruf dieser Methode tut, was man von dem Namen der Methode erwartet:

```
>>> datetime.datetime.now()
datetime.datetime(2023, 4, 20, 19, 20, 34, 203850)
```

(wobei man sich wieder wundern kann, was ein Mensch mit der Ausgabe von 203850 Millionstel Sekunden anfangen soll; bis das ausgegeben worden ist, sind schon wieder einige Millionstel Sekunden vergangen).

25.3.2.1. Die Klasse `timedelta`

Das Ergebnis von Rechenoperationen mit `datetime`-Objekten ist ein `timedelta`-Objekt.

```
>>> silvester = datetime.datetime(2023,12,31,23,59)
>>> heute = datetime.datetime(2023, 4, 21)
>>> silvester - heute
datetime.timedelta(254, 86340)
>>> heute - silvester
datetime.timedelta(-255, 50038, 772950)
```

Die angezeigten Werte sind in der Reihenfolge Tage, Sekunden, Microsekunden.

25.4. Aufruf von Programmen

Für Windows-Benutzer ist es ungewöhnlich, für alle anderen der Normalfall: ein Programm wird mit Parametern aufgerufen. Wir haben das für den Fall von Unterprogrammen, in Python Funktionen genannt, schon weiter oben im Kapitel 9.6.1 kennengelernt. Auch ein Programm kann mit Parametern aufgerufen werden. Im vorigen Kapitel wurde das Betriebssystem-Programm **ls** aufgerufen. Es zeigt ohne Parameter alle Dateien eines Verzeichnisses an:

Listing 25.17: Dateien eines Verzeichnisses

```
martin@mas-Oberndorf:~/public_html> ls
MBProgrammierung.html
MBWWW.html
MeineBuecher.html
MeineBuecher.txt
phpinfo.php
```

Hier wurde der Unix-Befehl **ls** benutzt. Unter Windows lautet der entsprechende Befehl **dir**.

Wenn ich nur die `HTML`-Dateien, also die, deren Dateiendung `.html` lautet, angezeigt haben will, muss ich dem Programm **ls** diese Information als Parameter mitgeben:

Listing 25.18: HTML-Dateien eines Verzeichnisses

```
martin@mas-Oberndorf:~/public_html> ls *.html
MBProgrammierung.html
MBWWW.html
MeineBuecher.html
```

Das ist natürlich auch bei einem Python-Programm möglich, das man auf der Shell (oder für Windows-Benutzer: in der Eingabeaufforderung) startet. Kurz nachgedacht: da

spielt Python doch zusammen mit dem jeweiligen Betriebssystem! Also benötigt man wieder das Modul `sys`!

Das folgende Programm, von mir **kommandoZeilenArgumente.py** genannt, begrüßt „alle“, wenn es ohne Parameter aufgerufen wird, und eine einzelne Person, wenn der Name einer einzelnen Person als Paramter mitgegeben wird:

Listing 25.19: Programm (mit oder ohne Parameter)

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

if len(sys.argv) > 1:
  print('Hallo '+sys.argv[1])
else:
  print('Hallo alle!!')
```

Ob ein Parameter mitgegeben wird oder nicht, wird mit Hilfe der Länge der Parameterliste überprüft; dabei ist wichtig zu wissen, dass jedes Programm immer einen Parameter erhält, nämlich den eigenen Programmnamen. Erst wenn die Länge der Liste länger als eins ist, wird ein „echter“ Parameter übergeben. Hier kommt als Nachweis, dass das richtig funktioniert, die Ausgabe des Programms bei zwei Aufrufen:

Listing 25.20: Programm-Auruf mit oder ohne Parameter

```
hier:~/bin/python/SystemTools> ./kommandoZeilenArgumente.py
Hallo alle!!
hier:~/bin/python/SystemTools> ./kommandoZeilenArgumente.py Martin
Hallo Martin
```

Nach dem, was wir weiter oben im Kapitel 14.1 über Fehlerbehandlung gesagt haben, geht das auch ein bißchen eleganter:

Listing 25.21: Programm (mit oder ohne Parameter) und Fehlerbehandlung

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

try:
  print('Hallo '+sys.argv[1])
except:
  print('Hallo alle!!')
```

Und die Ausgabe sieht genau wie oben aus.

Beide Module, `sys` und `os`, will ich jetzt einmal benutzen, um ein Tool zu schreiben, das mir die Umgebungsvariablen (das „environment“) eines Programms anzeigt. Das brauche ich dringend, denn ich kann mir das nicht alles merken, vor allem, wenn ich an verschiedenen Rechnern unter verschiedenen Betriebssystemen arbeite. Dabei will ich manchmal auch nicht alle Umgebungsvariablen ausgegeben haben, sondern nur den Wert einer Umgebungsvariablen.

Dazu benutze ich

- die Möglichkeit, Parameter an das Programm zu übergeben
- die Fehlerbehandlung mit `try - except`, falls keine Parameter übergeben werden
- eine Schleife über die Parameter, falls keine Parameter übergeben werden

Das Programm sieht so aus:

Listing 25.22: Umgebungs-Variable

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import os, sys

if len(sys.argv) > 1:
  for einArg in sys.argv[1:]:
    try:
      print(einArg, '\t\t\t==>', os.environ[einArg])
    except:
      print(einArg+' ist keine gültige Umgebungsvariable')
else:
  for k in os.environ:
    print(k,'\t\t\t==>', os.environ[k])
```

25.5. Weiter mit Verzeichnissen: das Modul `glob`

Ein weiteres Modul im Zusammenhang mit Verzeichnissen und den Dateien darin ist `glob`. Der Name leitet sich her von den Jokern, die es in jedem Betriebssystem gibt und die ihre Anwendung beim Durchsuchen von Dateibäumen haben. Sie stehen als Platzhalter nicht nur für eine Datei, sondern „globalisieren“ , das heißt, dass sie ein Muster beschreiben, das auf viele Dateien zutrifft. Ein Beispiel ist ein paar Absätze weiter oben zu sehen.

Das glob-Objekt besitzt eine Methode `glob`, die Datei- und Verzeichnisnamen mit Platzhaltern verarbeiten kann. Lassen wir also mal alle Dateien ausgeben, die als Dateiendung `.txt` haben.

Listing 25.23: globbing einfach

```
>>> glob.glob('*.txt')
['log-Sept.txt', 'nlMySQL.txt', 'Sticks.txt']
```

und schöner:

Listing 25.24: globbing einfach

```
>>> for eineDatei in glob.glob('*.txt'):
...     print(eineDatei)
...
log-Sept.txt
nlMySQL.txt
Sticks.txt
```

25.6. Das Modul `shutil`

Das Modul `shutil` enthält viele Funktionen, die das Arbeiten mit Dateien und Verzeichnissen vereinfachen. Als Beispiele seien genannt:

- `shutil.copy(alt, neu)`, das eine Datei kopiert
- `shutil.copytree(alt, neu)`das einen Dateibaum kopiert
- `shutil.move(alt, neu)`das eine Datei verschiebt
- `shutil.make_archive(base_name, format, root_dir=None, base_dir=None, ...)` das ein Archiv erzeugt

25.7. Das Modul `os.walk`

Wie man am Modulnamen sieht, ist `os.walk` ein Teil von `os. walk` geht durch einen Verzeichnisbaum. Der Aufruf des „Walkers“ geschieht (in der alten Version) durch `os.walk(Startverzeichnis, Auflist-Funktion, Dateiliste)` Dabei kann die Auflist-Funktion eine selbst geschriebene Funktion sein, im einfachsten Fall kann es auch die betriebssystem-interne Funktion (unter Unix / Linux: `ls`) sein. Die Dateiliste ist im Normalfall `None`, also die leere Liste. Das sieht so aus:

Listing 25.25: Walker (alte Version)

```
for dat in os.walk('./2016-17', 'ls ', None):
    print(dat)


('./2016-17', ['BFW2-M', 'IFOE-M', 'BKFH-M'], ['stoffvert.xls'])
('./2016-17/BFW2-M', [], ['mdl03.pdf', 'mdl04.pdf', 'ue_004.pdf',
'ue_001.pdf', 'ue_005.tex', 'ue_003.pdf', 'ka_01.tex', 'ue_004.tex',
'ue_001.tex', 'ue_000.tex', 'ka_03.tex', 'ka_01.pdf', 'ka_02.tex',
'mdl01.pdf', 'mdl03.tex', 'ka_04.tex', 'ka_02.pdf', 'nka_01.tex',
'ue_006.tex', 'nka_04.tex', 'mdl01.tex', 'nka_04.pdf', 'ue_006.pdf',
'ka_05.tex', 'ka_00.tex', 'ka_05.pdf', 'ka_03.pdf', 'ue_003.tex',
'mdl04.tex', 'ue_005.pdf', 'ue_002.tex', 'ue_002.pdf',
'Zweitkorr_2BFW2-1.ods', 'nka_01.pdf', 'ka_04.pdf'])
... und noch einige Zeilen mehr
```

In der neuen Version ist `os.walk` ein Generator. Das bedeutet, dass eine Auflist-Funktion nicht mehr als zweiter Parameter angegeben sein muss. Das sieht so aus:

Listing 25.26: Walker (neue Version)

```
for root, dirs, files in os.walk('./2016-17'):
    for datei in files:
        print(datei)

stoffvert.xls
mdl03.pdf
mdl04.pdf
ue_004.pdf
ue_001.pdf
ue_005.tex
ue_003.pdf
ka_01.tex
ue_004.tex
ue_001.tex
```

Wenn man die vollständigen Angaben, das heißt Pfadnamen plus Dateinamen ausgeben möchte, setzt man des vollständigen Dateinamen mit Hilfe von `os.path` zusammen.

Listing 25.27: Walk mit Ausgabe des vollständigen Dateinamens

```
>>> import os
>>> texte = os.walk('./2017-18')
>>> for dirPfad, dirNamen, datNamen in texte:
...     for datN in datNamen:
...         print(os.path.join(dirPfad,datN))
...
./2017-18/Rueckmeldung_Fehlzeiten_Serienbrief_123.docx
./2017-18/stoffvert.xls
./2017-18/stoffvert.ods
./2017-18/BK1-3INF/ka_01.aux
./2017-18/BK1-3INF/ka_01.tex
... etc.
```

Teil XIII.

Texte für Fortgeschrittene

26. Texte bearbeiten für Fortgeschrittene

26.1. Natural Language Toolkit (NLTK)

26.1.1. NLTK installieren und einrichten

Das NLTK ist in den Repositories der Standard-Linuxe enthalten.

26.1.2. Einfache Bearbeitung von Texten

NLTK wird eingebunden durch

Listing 26.1: Import von NLTK

```
import nltk
```

Zuerst wird jetzt ein klassischer englischer Text untersucht . . . naja, ganz einfach untersucht: The Wind Cries Mary von Jimi Hendrix. Die erste Zeile mag hier in diesem Beispiel genügen, denn den Text kennt ja wohl jeder auswendig ☺ . Danach wird der Text in seine Wörter zerlegt und diese Wörter angezeigt (ausnahmsweise englische Variablennamen).

Listing 26.2: Einfache NLTK-Untersuchung

```
>>> theWindCriesMary = '''
After all the jacks are in the boxes ...'''

words = theWindCriesMary.split()
>>> print(words)

['After', 'all', 'the', 'jacks', ...]
```

Jetzt wird es aber ein bißchen spannender: in welchem Zusammenhang taucht das Wort ‚wind' in diesem Text auf? Dazu müssen die Wörter des Textes wieder zu einem NLTK-compatiblen Text zusammengebaut werden, auf den dann die Funktion ‚concordance' losgelassen wird.

Listing 26.3: Zusammenhang (Concordance)

```
twcm = nltk.Text(words)
>>> twcm.concordance('wind')

Displaying 5 of 5 matches:
et Footprints dressed in red And the wind whispers Mary A broom is drearily sw
Somewhere a king has no wife And the wind it cries Mary The traffic lights the
life that they lived is dead And the wind screams Mary Will the wind ever reme
```

```
d And the wind screams Mary Will the wind ever remember The names it has blown
''No, this will be the last" And the wind cries Mary
```

»> twcm = nltk.Text(words)

»> twcm

<Text: After all the jacks are in the boxes...> »> twcm.concordance('wind')

Displaying 5 of 5 matches: et Footprints dressed in red And the wind whispers Mary A broom is drearily sw Somewhere a king has no wife And the wind it cries Mary The traffic lights the life that they lived is dead And the wind screams Mary Will the wind ever reme d And the wind screams Mary Will the wind ever remember The names it has blown "No, this will be the lastÄnd the wind cries Mary

Trennen eines Textes in Sätze:

Listing 26.4: Text in Sätze aufspalten

```
>>> from nltk.tokenize import sent_tokenize
>>> text = 'Hallo, Welt. Schön, dass Du auch da bist!
Wenigstens ist heute mit dem 12.12.2016 ein schönes Datum'
>>> sent_tokenize(text, language='german')
['Hallo, Welt.', 'Schön, dass Du auch da bist!',
'Wenigstens ist heute mit dem 12.12.2016 ein schönes Datum']
```

sent_tokenize steht für sentence tokenize

Trennen eines Textes in einzelne Wörter:

Listing 26.5: Text in einzelne Wörter aufspalten

```
>>> from nltk.tokenize import word_tokenize
>>> word_tokenize(text, language='german')
['Hallo', ',', 'Welt', '.', 'Schön', ',', 'dass', 'Du', 'auch',
'da', 'bist', '!','Wenigstens', 'ist', 'heute', 'mit', 'dem',
'12.12.2016', 'ein', 'schönes', 'Datum']
```

Beachte:

1. Auch ein Satzzeichen ist ein Wort.
2. Ein Datum ist ein Wort, Punkte nach Tag und Jahr sind keine Satzzeichen.

Stopwörter (stopwords) sind Füllwörter, d.h. Wörter, die sehr häufig vorkommen, aber für den Inhalt des Textes von untergeordneter Bedeutung sind, im Deutschen z.B. die Wörter ist, sind, hat, ein, der, die, das. Diese Wörter bei der Untersuchung auszuschließen geht so:

Listing 26.6: Stopwörter ausschließen

```
>>> from nltk.corpus import stopwords
>>> dt_Stopwoerter = set(stopwords.words('german'))
>>> [wort for wort in word_tokenize(text, language='german')
        if wort not in dt_Stopwoerter]

['Hallo', ',', 'Welt', '.', 'Schön', ',', 'dass', 'Du', '!',
    'Wenigstens', 'heute', '12.12.2016', 'schönes', 'Datum']
```

Separieren des Wortstammes eines Wortes (Stemming). Für die deutsche Sprache benutze

Listing 26.7: Import des deutschen NLTK-Stemmers

```
>>> from nltk.stem.snowball import GermanStemmer
```

Aufruf:

Listing 26.8: Suche den Wortstamm einiger Wörter

```
>>> meinStemmer = GermanStemmer()
>>> meinStemmer.stem('Untersuchung')
'untersuch'
>>> meinStemmer.stem('untersuchen')
'untersuch'
>>> meinStemmer.stem('Ableitung')
'ableit'
```

Teil XIV.

Mathematik (Forts.)

27. Mathematik (Forts.)

27.1. Das Paket numpy

numpy ist ein Paket für numerische Mathematik. Speziell beherrscht numpy Matrizen, und das soll hier als erstes demonstriert werden. Vieles davon ist (vor allem für Mathematiker) selbsterklärend.

Die Komponenten von numpy sind in der Sprache C geschrieben. C ist eine kompilierte Sprache, und damit sind die Komponenten von numpy sehr effektiv und von der Laufzeit her sehr schnell.

Ein kleiner Tip am Rande: auch wenn ich empfehle, immer zuerst die Hilfefunktion eines Moduls aufzurufen, wenn man ihn zum ersten Mal benutzt: bei numpy ist das ein zweifelhaftes Vergnügen: der Hilfetext ist 91000 Zeilen lang!

27.1.1. Matrizen

Eine große Stärke von numpy ist die Bearbeitung von Matrizen. Für den Mathematiker ist der Begriff der Matrix bekannt. Für den Python-Programmierer ist eine Matrix eine Liste von Listen - solange, bis er numpy kennenlernt. Und danach denkt er sich: Aha, eine Liste ist also eine (1 x n)-Matrix, eine Matrix mit einer Zeile und n Spalten.

Weil ich gerade so schön schwitze, nehme ich mir als erstes Beispiel einmal die Temperaturen (in Grad Celsius) in Tübingen der letzten 15 Tage vor (Juli 2019). Also rein damit in eine Liste (denn noch kenne ich numpy nicht): `tempCels = [18, 20, 21, 21, 23, 25, 26, 28, 27, 32, 27, 30, 33, 35, 37]` Jetzt sollen diese Temperaturen in Grad Fahrenheit umgerechnet werden. Dazu schreibe ich eine Funktion `cels2fahr`.

Beispiel 27.1 Funktion Celsius nach Fahrenheit

```
1 >>> def cels2fahr(g):
2     f = g * 9 / 5 + 32
3     return f
```

Und jetzt muss ich in einer Schleife jede Temperatur aus der Liste nehmen, die Funktion darauf loslassen und das zurückgelieferte Ergebnis in eine neue Liste schreiben.

Beispiel 27.2 Anwendung Funktion Celsius nach Fahrenheit

```
>>> tempFahr = []
>>> for temp in tempCels:
    tempFahr.append(cels2fahr(temp))

>>> print(tempFahr)
[64.4, 68.0, 69.8, 69.8, 73.4, 77.0, 78.8, 82.4,
80.6, 89.6, 80.6, 86.0, 91.4, 95.0, 98.6]
```

Das, was ich im letzten Satz des vorigen Absatzes geschrieben habe, kann durch die Verwendung von `numpy` vereinfacht werden; das geschieht dadurch, dass die Liste in eine eindimensionale Matrix umgewandelt wird. Diese eindimensionale Matrix, eine Instanz der Klasse `ndarray` (das steht für n-dimensionales Array, also n-dimensionale Matrix), wird erzeugt durch die Methode `array` und überlädt die Standard-Operatoren und ermöglicht es damit auch, dass an eine Funktion ein Objekt dieser Klasse übergeben wird.

Beispiel 27.3 das selbe mit NumPy

```
>>> import numpy as np
>>> tempNP = numpy.array(tempCels)
>>> cels2fahr(tempNP)
array([64.4, 68. , 69.8, 69.8, 73.4, 77. , 78.8, 82.4, 80.6, 89.6, 80.6,
    86. , 91.4, 95. , 98.6])
```

Es ist ein Standard in der Python-Welt, `numpy` wie in der ersten Zeile zu importieren.

27.1.1.1. Erzeugung von Matrizen

Matrizen kann man mit `numpy` auf verschiedene Weisen erstellen; das umfasst auch das Erzeugen von speziellen Matrizen.

1. Indem man ein `array` anlegt durch Angabe der Matrix-Werte. Hier wird eine (3 x 4)-Matrix und eine (4 x 2)-Matrix angelegt und danach angezeigt.

Beispiel 27.4 Matrizen anlegen mit array

```
import numpy as np
a1 = np.array([[1,2,1,2],[0,1,0,-1],[2,3,-3,-2]])
a2 = np.array([[0,2],[3,1],[2,0],[1,1]])
## Darstellung
>>> a1
array([[ 1,  2,  1,  2],
[ 0,  1,  0, -1],
[ 2,  3, -3, -2]])
>>> a2
array([[0, 2],
[3, 1],
[2, 0],
[1, 1]])
```

Das Format einer Matrix sollte natürlich auch ausgegeben werden können.

Beispiel 27.5 Format einer Matrix

```
>>> a1.shape
(3, 4)
>>> a2.shape
(4, 2)
```

Die Matrix `a1` hat 3 Zeilen und 4 Spalten.

2. `matrix` ist eine Unterklasse von `array`, mit der nur maximal zweidimensionale Felder erzeugt werden können.

Beispiel 27.6 Matrixen mit `matrix`

```
>>> m1 = np.matrix([[1,0],[0,1]])
>>> m1
matrix([[1, 0],
        [0, 1]])
>>> m2 = np.matrix([[1,0]])
>>> m2
matrix([[1, 0]])
```

Für Objekte der Klasse `matrix` sind alle Rechenoperationen definiert, die man aus der Oberstufe des Gymnasiums oder aus der Vorlesung „Lineare Algebra 1“ kennt.

3. Mit `array` können auch drei- (oder höher-)dimensionale Felder angelegt werden. Ein zweidimensionales Feld besitzt eine Länge und eine Breite; ein dreidimensionales Feld besitzt eine Länge, eine Breite und eine Höhe. Gut darstellen ließe sich das durch einen Quader, aber nicht hier auf Papier

Beispiel 27.7 Dreidimensionales Feld

```
>>> dreiDimMatrix = np.array([[[1,1,1],[2,2,2],[3,3,3]],
                [[4,4,4],[5,5,5],[6,6,6]],
                [[7,7,7],[8,8,8],[9,9,9]]]
          )
>>> print(dreiDimMatrix)
[[[1 1 1]
  [2 2 2]
  [3 3 3]]

 [[4 4 4]
  [5 5 5]
  [6 6 6]]

 [[7 7 7]
  [8 8 8]
  [9 9 9]]]
>>> print(dreiDimMatrix.shape)
(3, 3, 3)
```

XML-Datei anschauen, was hier fehlt

4. Im Gegensatz zu `array` erlaubt die Klasse `Matrix`
5. Dabei kann man den Datentyp der Elemente angeben.

Beispiel 27.8 Matrizen anlegen mit vorgegebenem Datentyp

```
import numpy as np
a1 = np.array([[1,2,1,2],[0,1,0,-1],[2,3,-3,-2]], dtype = float)
a2 = np.array([[0,2],[3,1],[2,0],[1,1]], dtype = complex)
## Darstellung
>>> a1

array([[ 1.,  2.,  1.,  2.],
    [ 0.,  1.,  0., -1.],
    [ 2.,  3., -3., -2.]])
>>> a2

array([[0.+0.j, 2.+0.j],
    [3.+0.j, 1.+0.j],
    [2.+0.j, 0.+0.j],
    [1.+0.j, 1.+0.j]])
```

Alles klar, oder?

6. Eine Matrix kann erzeugt werden, indem man einen Datenbereich angibt. Auch das ist schon von `range` bei der Arbeit mit Zählschleifen bekannt.

Beispiel 27.9 Bereiche bei der Erzeugung von Matrizen

```
>>> m1 = np.arange(16)
>>> m1

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])

>>> m2 = np.arange(4,20)
>>> m2

array([ 4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

>>> m3 = np.arange(1,23, 3)
>>> m3

array([ 1,  4,  7, 10, 13, 16, 19, 22])
```

7. Eine Einheitsmatrix ist eine quadratische Matrix, in der die Elemente der Hauptdiagonalen (der Diagonalen von links oben nach rechts unten) den Wert 1 haben, alle anderen Elemente den Wert 0. Eine Möglichkeit, sie zu erzeugen, ist die Methode `identity`. Sie benötigt als Parameter die Anzahl der Zeilen.

XML-Datei anschauen, was hier fehlt

Beispiel 27.10 Einheitsmatrix der Größe 5

```
>>> einhMatr5 = np.identity(5)

>>> print(einhMatr5)

[[1.  0.  0.  0.  0.]
[0.  1.  0.  0.  0.]
[0.  0.  1.  0.  0.]
[0.  0.  0.  1.  0.]
[0.  0.  0.  0.  1.]]
```

`identity` kann auch einen Datentyp als Parameter erhalten. Eine Einheitsmatrix mit ganzen Zahlen folgt:

Beispiel 27.11 Einheitsmatrix der Größe 5 mit ganzen Zahlen

```
>>> einhMatr5 = np.identity(5, dtype=int)

>>> print(einhMatr5)

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

Eine zweite Möglichkeit, Einheitsmatrizen zu erzeugen, ist die Methode `eye`. Auch sie benötigt als Parameter die Anzahl der Zeilen:

Beispiel 27.12 Einheitsmatrix mit `eye`

```
>>> einhMatr5 = np.eye(5, dtype=int)

>>> print(einhMatr5)

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

Und `eye` beherrscht auch, etwas ähnliches wie eine Einheitsmatrix zu bauen, die nicht quadratisch ist. In diesem Fall benötigt `eye` zwei Parameter, nämlich die Anzahl der Zeilen und die Anzahl der Spalten. Dabei muss angegeben werden, welche „Diagonale“ die Einsen enthalten soll. Die Beschreibung ist lästig und sprachlich unschön, darum sollen 3 Beispiele genügen; in eine 3x7 - Matrix wird die 0., die 2. und die (-2). Diagonale mit Einsen gefüllt:

Beispiel 27.13 Etwas ähnliches wie Einheitsmatrix

```
>>> matrix_3x7 = np.eye(3,7, k=0)

>>> print(matrix_3x7)

 [[1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0.]]
>>> matrix_3x7 = np.eye(3,7, k=2)

>>> print(matrix_3x7)

 [[0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0.]]
>>> matrix_3x7 = np.eye(3,7, k=-2)

>>> print(matrix_3x7)

 [[0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0.]]
```

8. Eine Nullmatrix ist eine Matrix, die nur Nullen enthält, eine Einsmatrix enthält nur Einsen.

Beispiel 27.14 Nullmatrix und Einsmatrix

```
>>> nullM = np.zeros((2,4), dtype=int)

>>> einsM = np.ones((2,5), dtype=int)

>>> print(nullM)

 [[0 0 0 0]
 [0 0 0 0]]
>>> print(einsM)

 [[1 1 1 1 1]
 [1 1 1 1 1]]
```

9. Eine Diagonalmatrix ist eine Matrix, die in der Hauptdiagonalen von 0 verschiedene Werte hat, alle restlichen Elemente sind 0.

Beispiel 27.15 Diagonalmatrix

```
## Diagonalmatrix
>>> diagonalmatrix = np.diag(np.array([1,-1,1,-1]))
>>> diagonalmatrix
array([[ 1,  0,  0,  0],
[ 0, -1,  0,  0],
[ 0,  0,  1,  0],
[ 0,  0,  0, -1]])
```

27.1.1.2. Matrizen umformen

1. Arrays können teilweise ausgegeben werden. Dabei funktioniert alles, was auch bei Listen funktioniert. Speziell kann man durch Slicing beliebige Untermatrizen auswählen.

Beispiel 27.16 Matrizen slicen

```
>>> dm = np.diag(np.array([1,-2,3,-4]))

>>> print(dm)

[[ 1  0  0  0]
[ 0 -2  0  0]
[ 0  0  3  0]
[ 0  0  0 -4]]
>>> print(dm[:2,:2])

[[ 1  0]
[ 0 -2]]
>>> print(dm[:2,2:])

[[0 0]
[0 0]]
>>> print(dm[1:3,1:3])

[[-2  0]
[ 0  3]]
>>> print(dm[:2,3:])

[[0]
[0]]
```

2. Wie schon oben bei den Listen gezeigt kann auch eine Matrix umgekehrt werden.

Beispiel 27.17 Matrizenzeilen umkehren

```
import numpy as np
a1 = np.array([[1,2,1,2],[0,1,0,-1],[2,3,-3,-2]])
>>> a1[::-1]
array([[ 2,  3, -3, -2],
[ 0,  1,  0, -1],
[ 1,  2,  1,  2]])
```

3. Das Format einer Matrix kann mit Hilfe der Methode `arange` umgekehrt werden.

Beispiel 27.18 Format ändern

```
>>> m2 = np.arange(16)
>>> m2

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> m2

array([ 4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
>>> m2.reshape(2,8)

array([[ 4,  5,  6,  7,  8,  9, 10, 11],
[12, 13, 14, 15, 16, 17, 18, 19]])
>>> m2

array([ 4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
>>> m2.reshape(4,4)

array([[ 4,  5,  6,  7],
[ 8,  9, 10, 11],
[12, 13, 14, 15],
[16, 17, 18, 19]])
```

4. Zwei oder mehr Matrizen können zusammengefasst werden durch die Methode `concatenate`. Dabei müssen die entsprechenden Formate der Einzelmatrizen passen. Das heißt genau: wenn ich die Matrizen in der Horizontalen zusammenfassen will, müssen sie in der Anzahl der Zeilen übereinstimmen. Sollen die Matrizen in der Vertikalen zusammengefasst werden, müssen sie in der Anzahl der Spalten übereinstimmen.

Beispiel 27.19 Zusammenfassung von Matrizen

```
>>> m1 = np.array(range(1,7))
>>> m1
array([1, 2, 3, 4, 5, 6])
    >>> m1_2x3 = m1.reshape(2,3)
>>> print(m1_2x3)
[[1 2 3]
 [4 5 6]]

 >>> m2 = np.array(range(11,17))
 >>> m2_2x3 = m2.reshape(2,3)
 >>> print(m2_2x3)
 [[11 12 13]
  [14 15 16]]

>>> m3_hintereinander = np.concatenate((m1_2x3, m2_2x3), axis=1)
>>> print(m3_hintereinander)
[[ 1  2  3 11 12 13]
 [ 4  5  6 14 15 16]]

>>> m4_untereinander = np.concatenate((m1_2x3, m2_2x3), axis=0)
>>> print(m4_untereinander)
[[ 1  2  3]
 [ 4  5  6]
 [11 12 13]
 [14 15 16]]

>>> m5 = np.array(range(21,25))
>>> m5_2x2 = m5.reshape(2,2)
>>> print(m5_2x2)
[[21 22]
 [23 24]]

>>> m6_hintereinander = np.concatenate((m1_2x3,m5_2x2), axis = 1)
>>> print(m6_hintereinander)
[[ 1  2  3 21 22]
 [ 4  5  6 23 24]]
```

27.1.1.3. Elementare Zeilenumformungen

In der linearen Algebra wird eine Matrix oft umgeformt, um sie einer besser handhabbaren Form zu bearbeiten. Das bedeutet oft, sie mit dem Gauss-Verfahren auf Zeilenstufenform zu bringen. Zulässige Operationen dazu sind

1. Multiplikation einer Zeile mit einer Zahl
2. Addition der j-ten Zeile zur i-ten Zeile
3. Vertauschen von j-ter und i-ter Zeile

4. Addition des Vielfachen der j-ten Zeile zur i-ten Zeile

Beispiel 27.20 Elementare Zeilenumformungen

```
## an dieser Matrix werden die obigen Umformungen vorgenommen
>>> a1 = np.array([[1,2,1,2],[0,1,0,-1],[2,3,-3,-2]])
>>> print(a1)

[[ 1  2  1  2]
 [ 0  1  0 -1]
 [ 2  3 -3 -2]]
## 1.) Multiplikation von Zeile 1 mit 5
>>> a_umf1 = np.array([5*a1[0],a1[1],a1[2]])

>>> print(a_umf1)

[[ 5 10  5 10]
 [ 0  1  0 -1]
## 2.) Addition von Zeile 2 zu Zeile 1 (Zeilen 2 und 3 bleiben unverändert)
>>> a_umf2 = np.array([a1[0]+a1[1], a1[1], a1[2]])

>>> print(a_umf2)

[[ 1  3  1  1]
 [ 0  1  0 -1]
 [ 2  3 -3 -2]]
## 3.) Vertauschen von Zeile 2 und Zeile 3
>>> a_umf3 = np.array([[a1[0], a1[2], a1[1]]])

>>> print(a_umf3)

[[[ 1  2  1  2]
 [ 2  3 -3 -2]
 [ 0  1  0 -1]]]
## 4.) Addition der negativen Hälfte von Zeile 3 zu Zeile 1
>>> a_umf4 = np.array([a1[0]+(-0.5)*a1[2],a1[1],a1[2]])

>>> print(a_umf4)

[[ 0.   0.5  2.5  3. ]
 [ 0.   1.   0.  -1. ]
 [ 2.   3.  -3.  -2. ]]
```

27.1.1.4. Rechnen mit Matrizen

1. Die Addition von 2 Matrizen ist nur definiert für Matrizen des selben Formats. In diesem Fall werden die Matrizen elementeweise addiert.

Beispiel 27.22 Elementare Zeilenumformungen

```
>>> m1 = np.matrix([[1, 2, 3],
       [4, 5, 6]])
>>> m1
matrix([[1, 2, 3],
        [4, 5, 6]])
>>> m2 = np.matrix([[11, 12, 13],
       [14, 15, 16]])
>>> m2
matrix([[11, 12, 13],
        [14, 15, 16]])
>>> print(m1 + m2)
[[12 14 16]
 [18 20 22]]
```

2. Bei der Multiplikation von Matrizen wird zuerst das Skalarprodukt betrachtet. Das Skalarprodukt zweier Vektoren (das sind (n x 1)-Matrizen) ist eine Zahl (ein Skalar). Voraussetzung dafür ist, dass beide Vektoren die selbe Anzahl Zeilen haben.

   In Python, genauer in `numpy`, wird das Skalarprodukt durch die Methode `dot` berechnet und ist auch auf Matrizen anwendbar. Anders als in der Analytischen Geometrie in der Schule ist hier die Voraussetzung, dass beide Vektoren Zeilenvektoren mit der selben Anzahl Elemente sind, oder dass der erste Vektor ein Zeilenvektor und der zweite Vektor ein Spaltenvektor ist, wobei der erste Vektor soviele Spalten haben muss wie der zweite Vektor Zeilen hat. Umständliche Erklärung, oder? Also schau auf das Beispiel:

Beispiel 27.23 Skalarprodukt

```
### 1. Fall
>>> v1 = np.array([1,-2,4])
>>> v1
array([ 1, -2,  4])

>>> v2 = np.array([[1],[2],[0]])
>>> v2
array([[1],
       [2],
       [0]])
>>> print(np.dot(v1,v2))
[-3]

### 2. Fall
>>> v1 = np.array([1,-2,4])
>>> v1
array([ 1, -2,  4])

>>> v2 = np.array([1,2,0])
>>> v2
array([1, 2, 0])
>>> print(np.dot(v1,v2))
-3

### 3. Fall
>>> v1 = np.array([[1],[-2],[4]])
>>> v1
array([[ 1],
       [-2],
       [ 4]])

>>> v2 = np.array([1,2,0])
>>> v2
array([1, 2, 0])
>>> print(np.dot(v1,v2))

Traceback (most recent call last):
  File "<pyshell#63>", line 1, in <module>
    print(np.dot(v1,v2))
ValueError: shapes (3,1) and (3,) not aligned: 1 (dim 1) != 3 (dim 0)

### 4. Fall
>>> v1 = np.array([[1],[-2],[4]])
>>> v1
array([[ 1],
       [-2],
       [ 4]])

>>> v2 = np.array([[1],[2],[0]])
>>> v2
array([[1],
       [2],
       [0]])
>>> print(np.dot(v1,v2))

Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    print(np.dot(v1,v2))
ValueError: shapes (3,1) and (3,1) not aligned: 1 (dim 1) != 3 (dim 0)
```

Wenn man die Vektoren als `matrix` definiert, sieht das ein wenig anders aus:

Beispiel 27.24 Skalarprodukt(?) mittels `matrix`

```
### 1. Fall: 2 Zeilenvektoren
>>> m3 = np.matrix([2,-1,3])
>>> m3
matrix([[ 2, -1,  3]])
>>> m4 = np.matrix([3,4,-1])
>>> m4
matrix([[ 3,  4, -1]])
>>> print(np.dot(m3,m4))
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print(np.dot(m3,m4))
ValueError: shapes (1,3) and (1,3) not aligned: 3 (dim 1) != 1 (dim 0)

### 2. Fall: Zeilenvektor mal Spaltenvektor
>>> m4 = np.matrix([[3],[4],[-1]])
>>>
>>> m4
matrix([[ 3],
        [ 4],
        [-1]])
>>> print(np.dot(m3,m4))
[[-1]]

### 3. Fall: Spaltenvektor mal Spaltenvektor
>>> m3 = np.matrix([[2],[-1],[3]])
>>> print(np.dot(m3,m4))
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    print(np.dot(m3,m4))
ValueError: shapes (3,1) and (3,1) not aligned: 1 (dim 1) != 3 (dim 0)
```

2. (Forts.) Aus der Schule kennt man nur die dritte Variante: Spaltenvektor mal Spaltenvektor ergibt Zahl. Das funktioniert hier nicht, sondern hier muss die Bedingung für die Multiplikation von Matrizen gelten: Anzahl der Spalten der 1. Matrix muss gleich sein der Anzahl der Zeilen der 2. Matrix. Die geliebte „Kippregel“ !!

   Das ist im obigen Beispiel die zweite Variante. Aber Vorsicht! Hier ist das Ergebnis kein Skalar (keine Zahl), sondern eine 1x1-Matrix. Das Skalarprodukt wird richtig nur berechnet wie im 2. Fall bei der Multiplikation mit `dot` von `arrays`.

3. Zwei Matrizen können miteinander multipliziert werden, wenn die Anzahl der Spalten der ersten Matrix gleich ist der Anzahl der Zeilen der zweiten Matrix. Das Ergebnis der Multiplikation ist eine Matrix des Formats ((Anzahl Zeilen der 1. Matrix) x (Anzahl der Spalten der 2. Matrix)), als Formel $(m \text{ x } n) * (n \text{ x } p) = (m \text{ x } p)$

Beispiel 27.25 Matrizen multiplizieren

```
import numpy as np
a1 = np.array([[1,2,1,2],[0,1,0,-1],[2,3,-3,-2]])
a2 = np.array([[0,2],[3,1],[2,0],[1,1]])
## Darstellung
>>> a1
array([[ 1,  2,  1,  2],
   [ 0,  1,  0, -1],
   [ 2,  3, -3, -2]])
>>> a2
array([[0, 2],
   [3, 1],
   [2, 0],
   [1, 1]])
## Matrizen-Multiplikation
>>> np.dot(a1,a2)
array([[10,  6],
   [ 2,  0],
   [ 1,  5]])
```

4. Die Länge eines Vektors berechnet sich als Wurzel aus dem Skalarprodukt des Vektors mit sich selbst, also

Beispiel 27.26 Länge eines Vektors

```
>>> vektor1 = np.array([3,0,4])
>>> laengeVektor1 = np.sqrt((vektor1**2).sum())
>>> print(laengeVektor1)
5.0
```

5. Ein nettes Beispiel zur Matrizenmultiplikation: Bei einem Skatturnier wird an 5 Tischen gespielt. Die Ergebnisse werden am Ende in einer Liste festgehalten.

Spieler	Tische				
	1	2	3	4	5
A	128	-34	-276	179	304
B	12	-420	123	-589	78
C	75	-374	248	-318	-792

Tabelle 27.1.: Daten Skatturnier

Diese Ergebnisse kommen jetzt in eine (3 x 5)-Matrix. Um die Verluste bzw. Gewinne je Tisch zu berechnen, wird die Gewinn-Verlust-Matrix gv_Matr mit dieser

Matrix multipliziert. Das ergibt wieder eine (3 x 5)-Matrix, in der dann in der jeweiligen Spalten steht, was die einzelnen Spieler eines Tisches erhalten bzw. bezahlen müssen.

Beispiel 27.27 Skatturnier

```
>>> gv_Matr = np.matrix([[2,-1,-1],[-1,2,-1],[-1,-1,2]])
>>> gv_Matr
matrix([[ 2, -1, -1],
        [-1,  2, -1],
        [-1, -1,  2]])

>>> ergMatr = np.matrix([[128,-34,-276,179,305],
                    [12,-420,123,-589,78],
                    [75,-374,248,-318,-792]])
>>> ergMatr
matrix([[ 128,  -34, -276,  179,  305],
        [  12, -420,  123, -589,   78],
        [  75, -374,  248, -318, -792]])

>>> zuZahlen = np.dot(gv_Matr, ergMatr)
>>> zuZahlen
matrix([[  169,   726,  -923,  1265,  1324],
        [ -179,  -432,   274, -1039,   643],
        [   10,  -294,   649,  -226, -1967]])
```

Das bedeutet, dass z.B. am Tisch 1 Spieler A 169 Geldeinheiten(GE) bekommt, Spieler B 179 GE zahlt und Spieler C 10 GE bekommt. (Die Summe in jeder Spalte muss 0 betragen!!)

27.1.1.5. Matrizengleichungen

Auch Matrizengleichungen, also lineare Gleichungssysteme (LGS), löst `numpy`.

Beispiel 27.28 LGS lösen

```
import numpy as np
## Gleichungen lösen
>>> a3 = np.array([[1,0],[2,1]])
>>> a3
array([[1, 0],
    [2, 1]])
>>> b3 = np.array([[3],[5]])
>>> b3
array([[3],
    [5]])

# a * x = b <=> np.linalg.solve(a,b)
>>> np.linalg.solve(a3,b3)
array([[ 3.],
    [-1.]])
```

27.2. Symbolische Mathematik (und andere schöne Dinge)

In diesem Kapitel soll ein schönes Paket für Python ein wenig untersucht werden: das Paket `sympy`. sympy steht für symbolische Mathematik in Python.

Mit sympy ist es möglich, Variable nicht als Platzhalter für einen Wert zu behandeln, sondern mit diesen Variablen zu rechnen. Eine einfache Rechnung ist etwa $x^4 : x^3 = x$

Ein anderes Beispiel für symbolisches Rechnen ist das Bilden einer Ableitungsfunktion. So gilt, wie jeder weiß,

$$f(x) = 2x^3 - x \Longrightarrow f'(x) = 6x^2 - 1$$

Also schauen wir mal, was Python mit Hilfe von sympy daraus macht:

Beispiel 27.29 sympy kann auch Ableiten

```
>>> import sympy
>>> x = sympy.Symbol('x')
>>> print(sympy.diff(3*x**4),x)
12*x**3 x
>>> print(sympy.diff(1/2*x**5+ 3*x**4-2*x),x)
2.5*x**4 + 12*x**3 - 2 x
```

Aber auch so eine Kleinigkeit wie Kürzen von Termen oder Polynomdivision oder das lösen von quadratischen Gleichungen beherrscht sympy:

Beispiel 27.30 sympy:Terme kürzen etc.

```
>>> import sympy
>>> x = sympy.Symbol('x')
>>> print(sympy.simplify((x**4 + x**2)/(x**2)) )
x**2 + 1
>>> print(sympy.simplify((x**3 - x**2 + 4*x - 4)/(x-1)))
x**2 + 4
>>> print(sympy.solve(x**2 - 5*x + 6,x))
[2, 3]
```

27.3. Pandas

Pandas ist ein weiteres Python-Modul, das auf `numpy` basiert und dazu dient, Daten aufzubereiten, zu analysieren und darzustellen. `Pandas` wird mit `import pandas as pd` importiert.

27.3.1. Series

Das erste Objekt aus Pandas, das ich hier vorstelle, ist die `Series`. (Vorsicht: Großbuchstabe!!). Eine `Series` ist Struktur, die eine andere Struktur indiziert (also mit einem Index versieht). Eine `Series` ist manchmal so ähnlich wie eine Python-Liste, und manchmal so ähnlich wie ein Python-Dictionary; sie kann aber mehr!!

Gibt man `Series` ein einzelnes Objekt mit, so werden die Elemente des Objektes mit den natürlichen Zahlen 0 bis (Länge des Objektes - 1) indiziert. Hier wird eine `Series` erstellt, die eine Liste enthält. Diese Liste wird ausgegeben.

Beispiel 27.31 `Series einer Liste`

```
>>> import pandas as pd
>>> pandaSerie = pd.Series([9,17,29,8,13,21])
>>> print(pandaSerie)
0     9
1    17
2    29
3     8
4    13
5    21
dtype: int64
```

Wollte man das mit Python-Bordmitteln erledigen, sähe das so aus:

Beispiel 27.32 Liste indiziert ausgeben

```
>>> liste = [9,17,29,8,13,21]
>>> for i in range(len(liste)):
        print(i,'\t',liste[i])

0       9
1       17
2       29
3       8
4       13
5       21
```

`Series` werden defaultmäß0ig mit dem „natürlichen“ Index ausgegeben: es wird einfach gezählt! [1] Das kann man dadurch ändern, dass man einen Index ausdrücklich angibt:[2]

Beispiel 27.33 `Series` mit einem Index

```
>>> getraenkeMasse = pd.Series(['Viertele','Halbe', 'Maß'],
                               index=[0.25, 0.5, 1.0])
>>> getraenkeMasse
0.25      Viertele
0.50         Halbe
1.00           Maß

```

Als Dictionary, ohne Verwendung von Pandas, sähe das wie im folgenden Beispiel aus. Hier kann man auch Schlüssel und Werte separat anzeigen lassen, zudem die Einträge als Liste von Tupeln.

[1]Nicht vergessen: man fängt bei 0 an zu zählen.

[2]In dem Beispiel geht es um Maße. Da aber in Python-Bezeichnern keine Sonderzeichen benutzt werden sollen (Python akzeptiert sie zwar, aber das geht nur so lange gut, wie man sich in einem Sprachraum bewegt.), mussten die Maße zu Massen werden. Pardon.

Während die beiden ersten Maßeinheiten vor allem in Baden-Württemberg keine Verständnisprobleme aufwerfen, ist die letzte Maßeinheit für jeden Wiesn- oder Wasnbesucher wohl bekannt. Und auch das ist keine Masse.

Beispiel 27.34 Getränkemaße als Dictionary

```
>>> massDic = {0.25:'Viertele',0.5:'Halbe',1.0:'Maß'}
>>> massDic
{0.25: 'Viertele', 0.5: 'Halbe', 1.0: 'Maß'}
>>> massDic.keys()
dict_keys([0.25, 0.5, 1.0])
>>> massDic.values()
dict_values(['Viertele', 'Halbe', 'Maß'])
>>> massDic.items()
dict_items([(0.25, 'Viertele'), (0.5, 'Halbe'), (1.0, 'Maß')])
```

Das geht auch mit `Series`, allerdings muss man ein bißchen aufpassen:

Beispiel 27.35 Schlüssel und Werte des Dictionary

```
>>> getraenkeMasse.keys
<bound method Series.keys of 0.25     Viertele
0.50        Halbe
1.00          Maß
dtype: object>

>>> getraenkeMasse.keys()
Float64Index([0.25, 0.5, 1.0], dtype='float64')

>>> getraenkeMasse.values
array(['Viertele', 'Halbe', 'Maß'], dtype=object)

>>> getraenkeMasse.items
<bound method Series.iteritems of 0.25     Viertele
0.50        Halbe
1.00          Maß
dtype: object>
```

`Series` „vereinheitlichen“ die Datentypen der enthaltenen Daten.

Beispiel 27.36 Verschiedene Datentypen

```
>>> pandaSerie2 = pd.Series([9,17.5,29,8,13,21])
>>> pandaSerie
0     9.0
1    17.5
2    29.0
3     8.0
4    13.0
5    21.0
dtype: float64
>>> pandaSerie3 = pd.Series([9,17.5,29,'Martin',13,21])
>>> pandaSerie3
0         9
1      17.5
2        29
3    Martin
4        13
5        21
dtype: object
>>> pandaSerie4 = pd.Series([9,17.5,29,[17,12],13,21])
>>> pandaSerie4
0           9
1        17.5
2          29
3    [17, 12]
4          13
5          21
dtype: object
```

In der `pandaSerie2` werden alle Zahlen in `float` umgewandelt, weil 17.5 eine Fließkommazahl ist (siehe die dtype-Angabe float). In `pandaSerie3` ist der `object`, da das die Oberklasse von String und Float ist, ebenso in der `pandaSerie4`.

Im nächsten Beispiel ist das Pandas-Objekt eine von Hand erzeugte Matrix. Die Elemente, die indiziert werden, sind folglich die Zeilen der Matrix.

Beispiel 27.37 `Series` einer Matrix

```
>>> pandaMatrix2 = pd.Series([[9,17],[29,8],[13,21]])
>>> print(pandaMatrix2)
0     [9, 17]
1     [29, 8]
2    [13, 21]
dtype: object
```

`Series` kann auch eine list comprehension übergeben werden.

Beispiel 27.38 `Series` durch `list comprehenion`

```
>> brueche = pd.Series([1/z for z in range(1,11)])

>>> print(brueche)

0     1.000000
1     0.500000
2     0.333333
3     0.250000
4     0.200000
5     0.166667
6     0.142857
7     0.125000
8     0.111111
9     0.100000
dtype: float64
```

Das ganze kann natürlich noch schöner ausgegeben werden, indem man als zweiten Parameter von Series durch das Schlüsselwort `index` eingeleitet eine Liste von Schlüsselwerten mitgibt.

Beispiel 27.39 `Series` durch `list comprehension` (Schöne Ausgabe)

```
>>> brueche = pd.Series([1/z for z in range(1,11)],
                    index=['1/'+str(z) for z in range(1,11)])

>>> print(brueche)

1/1      1.000000
1/2      0.500000
1/3      0.333333
1/4      0.250000
1/5      0.200000
1/6      0.166667
1/7      0.142857
1/8      0.125000
1/9      0.111111
1/10     0.100000
dtype: float64
```

Schlüssel und Werte können auch separat ausgegeben werden.

Beispiel 27.40 Schlüssel und Werte von `Series`

```
>>> brueche.index

Index(['1/1', '1/2', '1/3', '1/4', '1/5',
       '1/6', '1/7', '1/8', '1/9', '1/10'], dtype='object')
>>> brueche.values

array([1.        , 0.5       , 0.33333333, 0.25      , 0.2       ,
       0.16666667, 0.14285714, 0.125     , 0.11111111, 0.1       ])
```

Wird der `Series` als Daten ein Dictionary übergeben, werden die Schlüssel des Dictionary die Indizes, die sortiert werden.

Beispiel 27.41 `Series` eines Dictionary

```
>>> noten = pd.Series({'Karl':3.5,'Ina':2.8,'Lisa':2.1,
                       'Sepp':3.2, 'Lena':1.4,'Adam':2.0})

>>> print(noten)

Adam    2.0
Ina     2.8
Karl    3.5
Lena    1.4
Lisa    2.1
Sepp    3.2
dtype: float64
```

Aber Vorsicht: man muss hierbei darauf achten, bei Dictionaries habe ich das schon einmal geschrieben, dass die Schlüssel eindeutig sind. Wird ein Schlüssel doppelt verwendet, wird der erst zu diesem Schlüssel gehörende Wert durch den zweiten Wert überschrieben:

Beispiel 27.42 `Series` eines Dictionary mit doppeltem Schlüssel

```
>>> noten = pd.Series({'Karl':3.5,'Ina':2.8,'Lisa':2.1,
                       'Sepp':3.2, 'Lena':1.4,'Adam':2.0, 'Ina':5.0})
>>> noten
Adam    2.0
Ina     5.0
Karl    3.5
Lena    1.4
Lisa    2.1
Sepp    3.2
dtype: float64

```

Mit `apply` kann man eine Funktion auf die `Series` loslassen. Hier wird das gezeigt mit einer Serie von Temperaturen.

Beispiel 27.43 Umrechnung von `Series` von Temperaturen

```
>>> def celsius2fahrenheit(c):
        return 9.0/5.0 * c + 32

>>> temp = {'Wessling':18, 'Rottenburg':21, 'Berlin':22,
            'Paris':21, 'Colmar':20}

>>> tempSeries = pd.Series(temp)

>>> print(tempSeries)

Berlin        22
Colmar        20
Paris         21
Rottenburg    21
Wessling      18
dtype: int64

>>> tempSeries.apply(celsius2fahrenheit)

Berlin        71.6
Colmar        68.0
Paris         69.8
Rottenburg    69.8
Wessling      64.4
dtype: float64
```

Weil es so schön ist, folgt hier die Berechnung des Bruttopreises (also inklusive Mehrwertsteuer) einer Liste von Waren.

Beispiel 27.44 Berechnung von MwSt und Bruttopreis

```
>>> def mwstBerechnen(p):
        return p*0.19

>>> def bruttoPreisBerechnen(p):
        return p+mwstBerechnen(p)

>>> warenDic = {'D530':58.80, 'D830':63.86, 'D532':65.54, 'D555':67.22}

>>> warenSeries = pd.Series(warenDic)

>>> warenSeries.apply(mwstBerechnen)

D530    11.1720
D532    12.4526
D555    12.7718
D830    12.1334
dtype: float64

>>> warenSeries.apply(bruttoPreisBerechnen)

D530    69.9720
D532    77.9926
D555    79.9918
D830    75.9934
dtype: float64
```

Aber Vorsicht! Das ist noch keine Veränderung der Werte in der `Series`!!

Das nächste Beispiel: ein Metallwarenhändler verkauft Schrauben in den Größen M2 bis M6. Wenn der Bestand in einer Größe unter 150 sinkt, sollen 100 Stück nachbestellt werden.

Beispiel 27.45 Schrauben-Nachbestellung

```
>>> import pandas as pd

>>> def nachbestellen(z):
        if z < 150:
            return z + 100
         else:
            return z

>>> schrauben = pd.Series([412, 83, 312, 532, 130],
        index = ['M'+str(i) for i in range(2,7)])
>>> schrauben
M2     412
M3      83
M4     312
M5     532
M6     130
dtype: int64

>>> schrauben.apply(nachbestellen)
M2     412
M3     183
M4     312
M5     532
M6     230
dtype: int64
```

Das selbe Problem soll jetzt objektorientiert angegangen werden. Hier kommt zuerst die Klassendatei mit dem Manko, dass die Waren des Unternehmens vorläufig fest in der Klassendatei festgeschrieben sind. Das muss also noch geändert werden.

Beispiel 27.46 Nachbestellung objektorientiert

```
#!/usr/bin/python

import pandas as pd

class Warenlager():
    def __init__(self):
        self.waren = pd.Series([412, 83, 312, 532, 130],
            index = ['M'+str(i) for i in range(2,7)])

    def verkaufen(self):
        self.eineWare = input('Bezeichnung der Ware: ')
        self.menge = int(input('Menge: '))
        self.waren[self.eineWare] -= self.menge

    def nachbestellen(self, z):
        if z < 150:
            return z + 100
        else:
            return z

    def bestandAnzeigen(self):
        print('==============\n', self.waren, '==============\n')
```

Und so sieht das Aufrufprogramm aus:

Beispiel 27.47 Aufruf Nachbestellung

```
#!/usr/bin/python

from cl_Warenlager import Warenlager

meineWaren = Warenlager()
print('vor Aktionen')
meineWaren.bestandAnzeigen()
for i in range(2):
    meineWaren.verkaufen()
    print('nach Verkauf')
    meineWaren.bestandAnzeigen()
    meineWaren.waren = meineWaren.waren.apply(meineWaren.nachbestellen)
    print(' nach Nachbestellung')
    meineWaren.bestandAnzeigen()
```

Und es funktioniert:

Beispiel 27.48 Ausgabe der Nachbestellung

```
vor Aktionen
==============
M2      412
M3       83
M4      312
M5      532
M6      130

Bezeichnung der Ware: M4
Menge: 200

nach Verkauf
==============
M2      412
M3       83
M4      112
M5      532
M6      130

nach Nachbestellung
==============
M2      412
M3      183
M4      212
M5      532
M6      230

Bezeichnung der Ware: M5
Menge: 450
nach Verkauf
==============
M2      412
M3      183
M4      212
M5       82
M6      230

nach Nachbestellung
==============
M2      412
M3      183
M4      212
M5      182
M6      230
```

27.3.2. Dateien, die Pandas lesen kann

Pandas beherrscht folgende Dateitypen:

- csv-Dateien (comma separated values) (Dateiendung: .csv)
- xml-Dateien (extended markup language) (Dateiendung: .xml)
- Stata-Dateien (Dateiendung: .dta)
- Von Tabellenkalkulationsprogrammen erzeugte Dateien (Dateiendung: z.B. .xls)

Hier wurde eine csv-Datei vom Statistischen Bundesamt heruntergeladen, der monatlicher Verbraucherpreisindex. Die Datei enthält die monatlichen Daten der Jahre 2017 bis 2019. Hier wurde eingeschränkt auf den Januar der angegebenen Jahre; diese Daten stehen in den Spalten 2, 14 und 26. Außerdem wurden die ersten 7 Zeilen ignoriert, in denen nur eine Beschreibung der Datei mit zusätzlichen Informationen steht. Aus der gelesenen csv-Datei wurde dann ein `DataFrame` gemacht und dieser ausgegeben. Bei der Ausgabe sind nur die ersten 18 Datenzeilen hierhin kopiert.

Beispiel 27.49 Verbraucherpreisindex

```
>>> verbrInd = pd.read_csv('/home/fmartin/bin/python3/DatenStatBundesamt/
                    VerbraucherPreisIndex2017-18UTF8.csv',
                    sep=';',header=7, usecols=[1,2,14,26])
>>> verbrIndDataFrame = pd.DataFrame(verbrInd)
>>> print(verbrIndDataFrame)
                                       Unnamed: 1     2017     2018     2019
0                                             NaN   Januar   Januar   Januar
1          Nahrungsmittel und alkoholfreie Getränke    103,2    105,9    106,6
2                                  Nahrungsmittel    103,5    106,3    107,2
3                     Brot und Getreideerzeugnisse    100,7    101,7    103,4
4             Reis, einschließlich Reiszubereitungen     99,4     99,9    103,8
5                Mehl und andere Getreideerzeugnisse    100,6    101,4    105,8
6                               Brot und Brötchen    102,1    103,4    105,3
7                                 Andere Backwaren     99,7    101,2    103,7
8                      Pizza, Quiches und Ähnliches     97,9     99,5     98,8
9                                        Teigwaren     98,7     97,8     98,0
10                         Frühstückszubereitungen    102,0    102,6    102,9
11                        Andere Getreideprodukte     98,9     96,7     96,0
12                          Fleisch und Fleischwaren    101,3    103,8    105,1
13                            Rind- und Kalbfleisch    101,8    103,5    105,2
14                                  Schweinefleisch    101,8    104,4    105,0
15              Lamm- und Schaffleisch, Ziegenfleisch    107,4    110,2    115,5
16                                  Geflügelfleisch    100,6    102,2    104,8
17                          Andere Fleischprodukte    102,9    106,8    110,2
18        Innereien und andere Schlachtnebenprodukte    101,7    105,4    108,4
```

Dann habe ich hier noch das Literaturverzeichnis dieses Skriptes mit Pandas bearbeitet. Sämtliche Dateien dieses Skriptes sind `docbook-xml` Dateien. Ein Eintrag im Literaturverzeichnis sieht so aus:

Beispiel 27.50 Eintrag im Literaturverzeichnis XML

```
<biblioentry id="biblio.grass02">
    <abbrev>Freie Software</abbrev>
    <author>
        <firstname>Volker</firstname>
        <surname>Grassmuck</surname>
    </author>
    <copyright>
        <year>2002</year>
        <holder>Bundeszentrale für politische Bildung Bonn</holder>
    </copyright>
    <isbn>3-89331-432-6</isbn>
    <title>Freie Software</title>
    <subtitle>Zwischen Privat- und Gemeineigentum</subtitle>
</biblioentry>
```

Diese Datei wird jetzt mit folgendem Programm gelesen und bearbeitet:

Beispiel 27.51 XML-Datei lesen

```
#!/usr/bin/python

import xml.etree.cElementTree as et
import pandas as pd

def knotenWertHolen(knoten):
    if knoten is not None:
        return knoten.text
    else:
        return None
        # if knoten is not None else None

geparsteDatei = et.parse("/home/fmartin/texte/Python3Buch/XML/biblioPUR.xml")
ausgabeSpalten = ['Name', 'Vorname', 'Titel']
biblioDataFrame = pd.DataFrame(columns=ausgabeSpalten)

for node in geparsteDatei.getroot():
    nn = node.find('author/surname')
    vn = node.find('author/firstname')
    titel = node.find('title')

    biblioDataFrame = biblioDataFrame.append(
        pd.Series([knotenWertHolen(nn), knotenWertHolen(vn),
                   str(knotenWertHolen(titel)).strip()], index=ausgabeSpalten),
        ignore_index=True)

print(biblioDataFrame)
```

Und so sieht die Ausgabe aus:

Beispiel 27.52 Ausgabe der Literatur

```
1          Name          Vorname                              Titel
2  0   Grassmuck          Volker                     Freie Software
3  1       Jones  Christopher A.                    Python and XML
4  2     Swaroop            C.H.                   A Byte of Python
5  3        None            None                               None
6  4   Henscheid          Eckard                        Dummdeutsch
7  5        Lutz            Mark               Einführung in Python
8  6        Lutz            Mark                 Programming Python
9  7        Lutz            Mark                 Programming Python
10 8       Eckel           Bruce                   Thinking in Java
11 9   Schraitle          Thomas                        DocBook–XML
12 10       None            None
13 11       None            None
14 12 Walerowski           Peter                             Python
15 13      Lusth         John C.  The Alchemy of Programming: Python
16 14      Friedl     Jeffrey E.F.        Mastering Regular Expressions
17 15      Evans           David          Introduction to Computing
18 16     Briggs           Jason        Schlangengerangel für Kinder
19 17  Severance         Charles            Python for Informatics
20 18  Lubanovic            Bill                Introducing Python
```

27.3.3. Data Frames

Mehrdimensionale Datenfelder (oft Matrizen) beherrscht bereits das Standard-Python. Weiter oben wurde dann das `ndarray` von `numpy` angesprochen. Die `DataFrame` von `pandas` sind auch mehrdimensionale Felder, bei denen noch zusätzlich Zeilen- und Spaltenbezeichnungen angegeben werden können. Außerdem sind die `DataFrame` in der Lage, mit fehlenden Einträgen in den Feldern umzugehen.

27.3.4. Indizierung und andere Möglichkeiten der Auswahl

27.3.4.1. `Series`

Wenn die `Series` als Dictionary angelegt wurde, kann man auf einzelne Elemente und auf eine Teilmenge der Elemente auf 2 Arten zugreifen:

1. über den / die Schlüssel
2. über den Index

Zu beachten dabei ist, dass beim Zugriff über den Index die altbekannte Regel von Listen gilt: Anfangselement inklusive, Endelement exklusive. Beim Slicen über die (explizit angegebenen) Schlüssel werden sowohl Anfangs- als auch Endelement ausgegeben.

Beispiel 27.53 `Series` als Dictionary: Auswahl

```
>>> noten = pd.Series({'Karl':3.5,'Ina':2.8,'Lisa':2.1,
                       'Sepp':3.2, 'Lena':1.4,'Adam':2.0})
>>> noten
Adam    2.0
Ina     2.8
Karl    3.5
Lena    1.4
Lisa    2.1
Sepp    3.2
dtype: float64

>>> noten['Karl']
3.5

>>> noten['Ina':'Lisa']
Ina     5.0
Karl    3.5
Lena    1.4
Lisa    2.1
dtype: float64

>> noten[2]
3.5

>>> noten[1:3]
Ina     5.0
Karl    3.5
dtype: float64
```

Der Wirrwarr mit den verschiedenen Arten der Indizierung wird im folgenden Beispiel hoffentlich klar:

Beispiel 27.54 Wirrwarr bei der Indizierung

```
>>> datenWirrwarr = pd.Series({1:3.5,2:2.8,3:2.1})
>>> datenWirrwarr
1     3.5
2     2.8
3     2.1
dtype: float64
>>> datenWirrwarr[1]
3.5

>>> datenWirrwarr2 = pd.Series([3.5,2.8,2.1], index=[1,2,3])
>>> datenWirrwarr2
1     3.5
2     2.8
3     2.1
dtype: float64
>>> datenWirrwarr2[1]
3.5

>>> datenWirrwarr3 = pd.Series([3.5,2.8,2.1])
>>> datenWirrwarr3
0     3.5
1     2.8
2     2.1
dtype: float64
>>> datenWirrwarr3[1]
2.8
```

Um diese Widersprüchlichkeiten zu umgehen[3]
sind in Pandas weitere Möglichkeiten eingeführt, die Daten indizieren:

1. `loc`
2. `iloc`
3. `ix`

`loc` benutzt immer den explizit angegebene Index ... wenn er existiert. Das kann also auch noch zu Verwirrung führen.

[3] Ich weiß nicht, ob das pythonisch ist. In Python liegt ja vieles in der Verantwortung des Programmierers, von dem man annimmt, dass er weiß, was er tut.

Beispiel 27.55 `loc`

```
>>> datenWirrwarr2 = pd.Series([3.5,2.8,2.1], index=[1,2,3])
>>> datenWirrwarr2.loc[1]
3.5

>>> datenWirrwarr3 = pd.Series([3.5,2.8,2.1])
>>> datenWirrwarr3.loc[1]
2.8
```

Im `datenWirrwarr3`, in dem kein expliziter Schlüssel angegeben ist, wird der natürliche Schlüssel genommen, und das erwartet man nicht.

`iloc` hingegen benutzt immer den natürlichen Schlüssel, auch wenn explizit ein Schlüssel angegeben wird.

Teil XV.

Verzeichnisse

28. Glossar

A

Algorithmus Ein Algorithmus ist eine Vorschrift, die aus einzelnen Anweisungen besteht. Diese Vorschrift dient dazu, ein gegebenes Problem zu lösen. Das Wort Algorithmus ist eine Verballhornung des Namens Muhammed al-Chwarizmi, eines arabischen Gelehrten aus dem 9. Jahrhundert n. Chr.

In der Programmierung dient ein Algorithmus dazu, ein Programm zu schreiben, das ein Problem löst. Dabei werden die Anweisungen des Algorithmus in Befehle einer Programmiersprache übersetzt.

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:

```
1    a = 3
2    b = 5
3    c = a + b
```

und jeder wird verstehen, dass c danach den Wert 8 hat. Jetzt das selbe in COBOL:

```
1        MOVE 3 TO a.
2        MOVE 5 TO b.
3        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.

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.

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.)

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.

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 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.

29. Lösungen zu Aufgaben

29.1. 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 = int(input("Tag eingeben: "))
monat = int(input("Monat eingeben(als Zahl): "))
jahr = int(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])
```

29.2. 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: utf-8 -*-

endZahl = int(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 = int(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

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  # berechnet ggT und kgV zweier Zahlen mit dem Euklidschen Algorithmus
4
5  zz1 = int(input("erste Zahl eingeben "))
6  zz2 = int(input("zweite Zahl eingeben "))
7  z1 = zz1
8  z2 = zz2
9
10 while (z1 != z2):
11     while (z1 > z2):
12         z1 = z1 - z2
13 while (z2 > z1):
14     z2 = z2 - z1
15
16 print("ggT von ",zz1," und ",zz2," ist ",z1)
17
18 print("kgV von ",zz1," und ",zz2," ist ",zz1 * zz2 / z1)
```

Stellenwertsysteme

```
1  #!/usr/bin/python
2
3  zahl = int(input("Zahl im Dezimalsystem eingeben: "))
4  modus = int(input("Basis des Stellenwertsystems,
5                  in das die Zahl umgerechnet werden soll: "))
6  if modus > 10:
7      mehrAlsHex = ['0','1','2','3','4','5','6','7','8','9',
8                    'A','B','C','D','E','F','G','H','I','J','K','L']
9
10 restliste = []
11 while(zahl > 0):
12 (zahl,rest) = divmod(zahl, modus)
13 if modus > 10:
14     restAlpha = mehrAlsHex[rest]
15     restliste.append(restAlpha)
16 else:
17     restliste.append(rest)
18
19 restliste.reverse()
20
21 for i in restliste:
22     print(i,end=',')
```

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: utf-8 -*-

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: utf-8 -*-

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 = int(input('Fakultät bis ... '))
    print(fak(z))

    print('n über k')
    n = int(input('n = '))
    k = int(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 = 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 "Tippfehler" ein)'

            frageLText =  'von welchem Land ist '+stadt+' die Hauptstadt?'
                                + tippfText

            neuesLand = input(frageLText)
            if neuesLand == 'Tippfehler':
                pass
            else:
                neuLaHa[neuesLand] = stadt
            fehler += 1
```

```
48
49      print("Du hast ", punkte, " von ",punkte+fehler," richtige Antworten")
50      if fehler > 0:
51          print("also nochmal!!!")
52      LaHa.update(neuLaHa)
53      return (fehler)
```

Und hier das aufrufende Programm:

```
1 #!/usr/bin/python
2 import LandHauptst
3
4 print(LandHauptst.__doc__)
5 LaHa = {}
6 neuLaHa = {}
7 LandHauptst.initialisiere()
8 while LandHauptst.raetsel() > 0:
9     1
```

Caesar-Verschlüsselung

```
1 #!/usr/bin/python
2 # -*- coding: utf-8 -*-
3
4 dateiName = input('Welche Datei soll verschlüsselt werden? ')
5 schlBuchst = input('Schlüssel-Buchstabe, um den verschoben werden soll? ')
6 ausDatName = dateiName+'ver'
7 leseDat = open(dateiName,'r')
8 schreibDat = open(ausDatName,'w')
9
10 for zeile in leseDat.readlines():
11     ausZeile = ''
12     for zeichen in zeile:
13         if zeichen == ' ':
14             ausZeile += '#'
15         elif zeichen == '\n':
16             ausZeile += zeichen
17         else:
18             ausZeile += chr(((ord(zeichen) - ord('A')
19                            + ord(schlBuchst))%26)+ord('A'))
20     schreibDat.write(ausZeile)
21 schreibDat.close()
22 leseDat.close()
```

. . . und auch Entschlüsseln

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

dateiName = input('Welche Datei soll entschlüsselt werden? ')
schlBuchst = ('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: utf-8 -*-

dateiName = input('Welche Datei soll verschlüsselt werden? ')
schlWort = 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()
```

29.3. 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: utf-8 -*-

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 = int(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)
```

Beispiele

8.1 aus Englisch-Deutsch mache Deutsch-Englisch 158
8.2 Zugriff auf das obige Wörterbuch 159
8.3 erfolgloser Zugriff auf ein Dictionary 159
8.4 Abfangen eines erfolglosen Zugriffs auf ein Dictionary 159
8.5 Was fehlt, wird ergänzt 159
8.6 Buchstabenhäufigkeit bestimmen und alphabetisch ausgeben 161
8.7 Drei Chinesen 162
8.8 Drei Chinesen . . . mit richtiger Ersetzung der Diphtonge 162
8.9 Caesar-Verschlüsselung 166

25.1 Programm für zeilenweises Lesen einer Datei mit `readline` 454
25.2 Zeitmessung 460

27.1 Funktion Celsius nach Fahrenheit 475
27.2 Anwendung Funktion Celsius nach Fahrenheit 476
27.3 das selbe mit NumPy 476
27.4 Matrizen anlegen mit array 477
27.5 Format einer Matrix 477
27.6 Matrixen mit `matrix` 477
27.7 Dreidimensionales Feld 478
27.8 Matrizen anlegen mit vorgegebenem Datentyp 478
27.9 Bereiche bei der Erzeugung von Matrizen 479
27.10 Einheitsmatrix der Größe 5 479
27.11 Einheitsmatrix der Größe 5 mit ganzen Zahlen 480
27.12 Einheitsmatrix mit `eye` 480
27.13 Etwas ähnliches wie Einheitsmatrix 481
27.14 Nullmatrix und Einsmatrix 481
27.15 Diagonalmatrix 482
27.16 Matrizen slicen 482
27.17 Matrizenzeilen umkehren 483
27.18 Format ändern 483
27.19 Zusammenfassung von Matrizen 484
27.20 Elementare Zeilenumformungen 485
27.21 Addition von Matrizen 486
27.22 Elementare Zeilenumformungen 486
27.23 Skalarprodukt 487
27.24 Skalarprodukt(?) mittels `matrix` 488

27.25 Matrizen multiplizieren . . . 489
27.26 Länge eines Vektors . . . 489
27.27 Skatturnier . . . 490
27.28 LGS lösen . . . 491
27.29 sympy kann auch Ableiten . . . 491
27.30 sympy:Terme kürzen etc. . . . 492
27.31 `Series einer Liste` . . . 492
27.32 Liste indiziert ausgeben . . . 493
27.33 `Series` mit einem Index . . . 493
27.34 Getränkemaße als Dictionary . . . 494
27.35 Schlüssel und Werte des Dictionary . . . 494
27.36 Verschiedene Datentypen . . . 495
27.37 `Series` einer Matrix . . . 495
27.38 `Series` durch `list comprehenion` . . . 496
27.39 `Series` durch `list comprehension` (Schöne Ausgabe) . . . 496
27.40 Schlüssel und Werte von `Series` . . . 497
27.41 `Series` eines Dictionary . . . 497
27.42 `Series` eines Dictionary mit doppeltem Schlüssel . . . 497
27.43 Umrechnung von `Series` von Temperaturen . . . 498
27.44 Berechnung von MwSt und Bruttopreis . . . 499
27.45 Schrauben-Nachbestellung . . . 500
27.46 Nachbestellung objektorientiert . . . 501
27.47 Aufruf Nachbestellung . . . 501
27.48 Ausgabe der Nachbestellung . . . 502
27.49 Verbraucherpreisindex . . . 503
27.50 Eintrag im Literaturverzeichnis XML . . . 504
27.51 XML-Datei lesen . . . 504
27.52 Ausgabe der Literatur . . . 505
27.53 `Series` als Dictionary: Auswahl . . . 506
27.54 Wirrwarr bei der Indizierung . . . 507
27.55 `loc` . . . 508

Index

überschreiben
  von Methoden, 315

Algorithmus, 54, 200
Alternative, 64, 201, 208
Anweisung, 54, 79
Anweisungsblock, 210
append, 147
Array, *siehe* Liste
Attribut, 291
Aufgaben
  zu Alternative, 220
  zu Auswahl, 220
  zu Dateien, 361
  zu Dictionaries, 193
  zu doctest, 282
  zu Funktionen, 271
  zu Klassen, 331
  zu Listen, 193
  zu Programmierung allgemein, 74
  zu Schleifen, 248
  zu Sequenzen, 200
  zu Texten, 137
  zu Variablen, 94
Ausdrücke
  reguläre, 127
Ausdruck, 85
Ausgabe
  mit print, 85
Ausnahme, 340
Auswahl, 201

Bedingung, 211
Bedingungen, 203
  Kurzschluß-Verfahren, 206
  verknüpfte, 205
Befehl, 54, 79
Betriebssystem, 62
  Befehle, 62
binäre Suche, 146
Block, 210
  Einrückung, 210
Boole'sche Variable
  False, 202
  True, 202

C, 57
Caesar-Verschlüsselung, 177
capitalize, 391
CC BY-NC-SA, 35
Cleese, John, 45
COBOL, 57
collections, 173
  defaultdic, 173
Compiler, 56
Creative Commons, 35

Dateien, 349
Dateisystem, 62
Datenbank, 363
  MySQL, 363
Datenstrukturen
  Dictionary, 168
  Liste, 139
  Tupel, 179
Datum, 468
Debugger, 73
Dekorator, 308
Dezimalpunkt, 83
Dictionary, 168

- Comprehension, 177
- defaultdic, 173
- get, 169
- set, 169
- sortiert ausgeben, 175

divmod, 85
docstring, 67

Editor, 59
- Syntax highlighting, 59
- Wortvervollständigung, 59

eingebaute Funktionen, 253
Entscheidung, 64
Entscheidungen
- verschachtelte, 218

enumerate, 233
environment, *siehe* Umgebungsvariable
exception, 340
expression, 85
- regular, 127

extend, 151

False, 202
Fehler, 72
- Debugger, 73

Fehlerarten
- Laufzeitfehler, 73
- semantischer Fehler, 73
- Syntaxfehler, 72

Feld, *siehe* Liste
Filtern von Listen, 147
float, 83
for-Schleife, 225
Formatierung
- im C-Stil, 118
- im Python-Stil, 120
- mit format, 120

Fortran, 57
Freie Software, 58
Funktionen, 252
- eingebaute, 253
- lambda, 272
- rekursive, 269

Gültigkeitsbereich, 262
glob, 473
globale Variable, 263
GUI
- Tkinter, 451

help, *siehe auch* Hilfe, 285, *siehe auch* Hilfe, 286
Hilfe, 67, 285, 286
- dir, 286
- help, 285

Hilfetext, 67
HTML, 417
- Überschrift, 418
- Absatz, 418
- Formular, 420
- Geordnete Liste, 419
- Hervorhebungen, 419
- Leerzeichen, 418
- Link, 419
- Liste, 419
- Spiegelstrich-Liste, 419
- Strukturierung, 418
- tag, 418

IDE, 60
idle, 78
Idle, Eric, 45, 78
Import, 284
input, 101
int, 83
Interpreter, 56
Iteration, 64, 201
Iterator, 129, 186

Kamel-Schreibweise, 292
Klasse
- Attribut, 291
- Funktion super, 314
- Methode, 291
- new style, 313

Kodierung, 116
Kommentar, 66
Konstruktor, 296
Kontrollstruktur, 208
Konventionen, 69

Kryptographie
  Caesar-Verschlüsselung, 177

lambda-Funktionen, *siehe* Funktionen
Leerzeichen, 115
LGS, 500
Lineares Gleichungssystem, 500
list, *siehe* Liste
list comprehension, 237
Liste, 110
Listen, 139
  comprehension, 141
  Einträge mit Namen, 160
  Element löschen, 155
  erzeugen, 140
  fester Typ der Daten, 164
  Filterfunktion, 147
  filtern, 147
  flache Kopie, 163
  Generator, 141
  Hausnummer, 140
  Index, 140, 146
  kopieren, 163
  Listenelement, 140
  Matrix, 157
  pop, 155
  Position eines Elements, 146
  Shortcut, 148
  slicen, 156
  sortieren, 156
  sortieren (groß oder klein), 157
  sortierte Kopie, 157
  umkehren, 156
  Verkettung, 146
  verlängern, 147, 151
  Verlängerung, 152
  zip, 162
lokale Variable, 262

Maskieren, 105
Mathematik
  LGS, 500
  lineares Gleichungssystem, 500
  numpy, 485
  sympy, 501
Matrix, 157
  Nullmatrix, 158
Matrizen, 485
  Addition, 495
  Diagonal-, 491
  Einheits-, 489
  Eins-, 491
  Erzeugung von, 486
  invertieren, 492
  Multiplikation von, 499
  Null-, 489, 491
  Skalarprodukt, 496
  umkehren, 492
Mehrfach-Entscheidungen, 215
Menge, 187
  Operationen, 187
Methode, 291
  Aufruf, 113
  statisch, 308
Methoden überschreiben, 315
Modul, 283
Module
  glob, 473
  os, 464
  os.walk, 474
  pickle, 358
  shutil, 474
  sys, 461
Modulus, 84
Monty Python, 45
MySQL, 363

Namensraum, 262
natürliche Sprache, 479
new style, 313
NLTK, 479
numpy, 485

objektorientierte Programmierung
  Vererbung, 293
OOP
  Vererbung, 293
Operatoren

arithmetische, 85
os, 464
os.walk, 474

packages, 288
Paket
numpy, 485
sympy, 501
Pakete, 288
Palin, Michael, 45
Parameter, 252
PEP 8, 71
pickle, 358
print, 85
private, 303
Programmaufruf, 99
aus Python-Programm, 470
Programmierrichtlinien, 71
Programmiersprachen
C, 57
COBOL, 57
Fortran, 57
Programmierung
strukturierte, 200
protected, 303
Pseudocode, 207
public, 303

Quellcode, 50, 98
Quelltext, 56

range, 140, 227
raw strings, 115
Regeln für Bezeichner
Klassen und Objekte, 292
Variablen, 91
reguläre Ausdrücke, 127
regular expression, 127
Rekursion, 269
Reservierte Wörter, 71
Rest, 84
Rundungsfehler, 86

Schlüsselwörter, 71
Schleife, 64, *siehe auch* strukturierte Programmierung, 225
Zähl-, 225
Schleifen
Überspringen eines Elements, 245
Abbruch, 245
weiter mit dem nächsten Element, 245
Schleifen-Beispiele, 236
Selbstreferenz, 296
self, 296
Sequenz, 64, 200
sha-bang, 99
Shortcut
Listen, 148
shutil, 474
Sichtbarkeit
private, 303
protected, 303
public, 303
Skalarprodukt, 496
Slicing, 111
Sprache
natürliche, 479
Statement, 79
statement, 54
statische Methode, 308
Steuerzeichen, 107
Stil, 69
String-Methoden
str.center, 114
str.index, 114
str.lower, 114
str.upper, 114
strings, 105
strukturierte Programmierung, 200
Alternative, 64, 201, 208
Entscheidung, 64
For-Schleife, 225
Iteration, 64, 201
Schleife, 64
Sequenz, 64, 200, 201
While-Schleife, 234
Suche

binäre, 146
super, 314
sympy, 501
Syntax highlighting, 59
sys, 461

Text
capitalize, 391
Tkinter, 451
True, 202
Tupel, 179
benannte, 182
type, 83
Typisierung
dynamisch, 89
statisch, 89

Umgebungsvariable, 472
Umkehrung
Liste, 156
Umkehrung (beliebige Objekte), 156
UML, 296
Umbrello, 297

Variable, 88
globale, 263
lokale, 262
Wertzuweisung, 93
Variablenname, 91
Variablennamen, 91
Vererbung, 293

Wahrheit, 202
Wahrheitswerte, 212
False, 204
True, 204
Wertzuweisung, 93
while-Schleife, 225
whitespace, *siehe* Leerzeichen
Wiederholung, 201
Wortvervollständigung, 59

Zählschleife, 225
mitzählen, 233
Zahlen, 79
Zeichenketten, 105, *siehe auch* Text, *siehe* String, 114
Teilbereiche von, 111
Zeilenvorschub, 107
Zeit, 468
Zeitmessung, 469
zip, 168
Zuweisungsmuster, 95
Zuweisungsoperator, 79, 88, 93

Literatur

[1] Daniel Arbuckle. *Python testing.* English. E-Book. Packt Pub., 2010. ISBN: 978-1-84719-884-6.

[2] Guido Arnold. *Freie Software. Was ist das?* 2019. URL: http://www.gnu.org/philosophy/free-sw.de.html.

[3] Bastian Ballmann. *Network Hacks - Intensivkurs.* E-Book. Springer, 2020, S. 174. ISBN: 978-3-642-24305-9.

[4] David M. Beazley. *Python Essential Reference.* English. E-Book. Sams, 2006, S. 625. ISBN: 0-672-32862-3.

[5] Stevan Bird, Ewan Klein und Edward Loper. *Natural Language Processing With Python.* English. E-Book. O'Reilly, 2009, S. 512. ISBN: 978-0-596-51649-9.

[6] Anthony D. Briggs. *Hello! Python.* English. E-Book. Manning, 2012, S. 398. ISBN: 1-935182-08-0.

[7] Martin C. Brown. *XML processing with Perl, Python, and PHP.* English. E-Book. Sybex, 2001, S. 422. ISBN: 0-7821-4021-1.

[8] Dave Brueck und Stephen Tanner. *Python 2.1 Bible.* English. E-Book. Wiley, 2001, S. 731. ISBN: 0-7645-4807-7.

[9] Allen B. Downey. *Think Python.* English. SoHo Books, 2009, S. 216. ISBN: 1-44141-916-0.

[10] Bruce Eckel. *Thinking in Java.* Deutsch. 3. A. Markt Und Technik, 2005, S. 1119. ISBN: 3-8272-6896-6.

[11] David Evans. *Introduction to Computing.* English. Version: August 19, 2011. Self Publishing, 2011, S. 251. ISBN: 1-46368-747-8.

[12] Jeffrey E. F. Friedl. *Mastering Regular Expressions.* Englisch. 2. A. O'Reilly Media, 2002, S. 484. ISBN: 0-596-00289-0.

[13] Tony. Gaddis. *Starting out with Python.* English. Pearson Addison Wesley, 2009. ISBN: 0-321-53711-4.

[14] Gray Girling. *Das Raspberry Pi Praxishandbuch.* Deutsch. 1. Aufl. Franzis, 2013, S. 299. ISBN: 3-645-60262-3.

[15] Michael H. Goldwasser. *Object-oriented programming in Python.* English. Pearson Prentice Hall, 2008. ISBN: 0-13-615031-4.

[16] Volker Grassmuck. *Freie Software.* Deutsch. 2002. ISBN: 3-89331-432-6.

[17] John E. Grayson. *Python and Tkinter programming*. English. Manning, 2000. ISBN: 1-884777-81-3.

[18] Rashi Gupta. *Making use of Python*. English. John Wiley & Sons, 2002. ISBN: 0-471-21975-4.

[19] Tim Hall und J-P Stacey. *Python 3 for Absolute Beginners*. Apress, 2009, S. 314. ISBN: 978-1-43021-633-9.

[20] Daryl Harms und Kenneth McDonald. *Go To Python* . Deutsch. 1. Aufl. Addison-Wesley, 2001, S. 640. ISBN: 3-8273-1800-9.

[21] Alan Harris. *Pro IronPython*. English. Springer London, Limited, 2009. ISBN: 978-1-43021-963-7.

[22] *Head First Python*. E-Book. O'Reilly Media, 2010. ISBN: 978-1-44938-267-4.

[23] Doug. Hellmann. *The Python standard library by example*. English. Addison-Wesley, 2011. ISBN: 0-321-76734-9.

[24] Eckhard Henscheid. *Dummdeutsch*. Deutsch. Philipp Reclam Jun. Verlag GmbH, 1999, S. 294. ISBN: 3-15-008865-8.

[25] Lars Heppert. *Coding for Fun mit Python*. Hrsg. von Galileo Press. 1. Aufl. Bonn, 2010, 325 S. ISBN: 978-3-8362-1513-8.

[26] Magnus Lie Hetland. *Beginning Python*. English. 2nd Ed. Apress, 2008, S. 656. ISBN: 1-59059-982-9.

[27] Christopher A. Jones und Fred L. Drake. *Python & XML*. Deutsch. O'Reilly, 2002. ISBN: 3-89721-175-0.

[28] Michael Lauer. *Python und GUI-Toolkits*. Hrsg. von Mitp. 1. Aufl. Bonn, 2002, 525 S. ISBN: 978-3-8266-0844-5.

[29] Martin von Löwis und Nils Fischbeck. *Das Python-Buch*. Deutsch. Addison-Wesley, 1997, S. 483. ISBN: 3-8273-1110-1.

[30] Mark Lutz. *Programming Python*. English. 2nd Edition. O'Reilly, 2001, S. 880. ISBN: 1-56592-197-6.

[31] Mark Lutz. *Programming Python*. Englisch. 3rd Ed. O'Reilly Media, 2006, S. 1596. ISBN: 0-596-00925-9.

[32] Mark Lutz und David Ascher. *Einführung in Python*. Deutsch. 2. Auflage. O'Reilly, 2007, S. 624. ISBN: 3-89721-488-1.

[33] Andreas Müller und Sarah Guido. *Einführung in Machine Learning mit Python*. Hrsg. von Dpunkt. Verlag GmbH. ISBN: 978-3-9600904-9-6.

[34] *Python cookbook*. English. E-Book. O'Reilly, 2005, S. 807. ISBN: 0-596-00797-3.

[35] *Python for Unix and Linux System Administration*. O'REILLY, 2008, S. 458. ISBN: 978-0-596-51582-9.

[36] Kenneth Reitz. *PEP 8 — the Style Guide for Python Code*. ohne Jahr. URL: https://pep8.org.

[37] Guido van Rossum, Berry Warsaw und Nick Coghlan. *PEP 8 – Style Guide for Python Code*. 2022. URL: peps.python.org/pep-0008/.

[38] Peter Walerowski. *Python*. Deutsch. 1., Aufl. Addison-Wesley, München, 2007, S. 336. ISBN: 3-8273-2517-X.