Hochschule Bochum
WS 2023/2024

Bauinformatik mit Python

Dipl.-Ing. Martin Vogel

Pythonbrug Amsterdam, Alain Rouiller, CC BY-SA 2.0

Version vom 23. November 2023
Auf martinvogel.de/python erhalten Sie stets die aktuellste Ausgabe..

Inhaltsverzeichnis

1 Einleitung 10

1.1 Bedeutung der Bauinformatik ... 12

1.2 Ermutigung ... 13

1.3 Lerntipps ... 14

1.4 Suchmaschinentipps ... 16

2 PC-Grundkenntnisse 18

2.1 Tastatur ... 19

2.2 Betriebssystem ... 24

2.3 Dateien und Verzeichnisse ... 26

2.3.1 Verzeichnisbäume ... 27

2.3.2 Versteckte Dateinamenerweiterungen ... 28

2.3.3 Desktop, Ordner und Verzeichnisse ... 30

2.3.4 Archivdateien ... 33

2.4 Zwischenablage ... 36

2.5 Bildschirmkopien ... 38

2.6 Sonderzeichen ... 42

2.7 Texteditoren ... 44

2.8 Textverarbeitungen ... 47

2.8.1 Formatvorlagen ... 47

2.8.2 Schriftarten ... 48

2.8.3 Zeichenformatierung ... 48

2.8.4 PDF-Dateien ... 50

2.8.5 Grafiken ... 51

2.8.6 Verzeichnisse ... 52

2.8.7 Erzwungene neue Seite ... 52

2.8.8 Kopf- und Fußzeilen ... 52

2.9 Tabellenkalkulationen ... 53

2.9.1 Formeln ... 54

2.9.2 Variablennamen 54
2.9.3 Relative und absolute Zellbezüge 55
2.9.4 Funktionen 56
2.9.5 Zellbereiche 57
2.9.6 Fallunterscheidungen mit WENN 57
2.9.7 VERWEIS, SVERWEIS und WVERWEIS 60
2.9.8 Zielwertsuche und Solver 62
2.9.9 Matrixformeln 65
2.9.10 Diagramme 66
2.9.11 CSV-Dateien und Tabellenkalkulationen 67
2.9.12 Anwendungsgrenzen 71

3 Hypertext 72

3.1 HTML-Tags 73
3.2 Hierarchische Ordnung 74
3.3 Attribute 76
3.4 HTML-Entitäten 77
3.5 CSS 78

4 Algorithmen und ihre Darstellung 79

4.1 Flussdiagramm 80
4.2 Struktogramm 81
4.2.1 Reihenfolge der Arbeitsschritte 81
4.2.2 Fallunterscheidung 82
4.2.3 Mehrfachauswahl 83
4.2.4 Abweisende Schleife 83
4.2.5 Nichtabweisende Schleife 84
4.2.6 Endlosschleife 84
Ausbruch aus der Endlosschleife.....85
4.2.7 Beispiel für ein vollständiges Struktogramm 86
4.2.8 Struktogramm-Editor 87

5 Python **88**

5.1 Download und Installation **90**

5.1.1 Module für wissenschaftliches Arbeiten 94

5.1.2 Virtuelle Umgebungen 97

5.2 Erste Schritte in der IDLE-Shell **98**

5.3 Fehlermeldungen **101**

5.4 Konstanten **103**

5.5 Variablen **104**

5.5.1 Variablennamen 106

5.6 Rechenoperationen **109**

5.7 Funktionen und Module **111**

5.7.1 Funktionsweiser Import 113

5.7.2 Modulweiser Import 114

5.7.3 Das Mathematik-Modul: math 115

5.7.4 Funktionszuweisungen 117

5.8 Eingabe mit input() **118**

5.8.1 Lesen aus Textdateien 118

5.9 Ausgabe mit print() **120**

5.9.1 Ausgabe in Textdateien 121

Warnung!.....121

5.9.2 Alternatives Trennzeichen: sep 122

5.9.3 Alternatives Zeilenende: end 123

5.10 Typumwandlung **124**

5.10.1 Evaluation von Ausdrücken 125

5.11 Das erste richtige Programm **128**

5.12 Quelltextformatierung **130**

5.12.1 Kommentarzeilen 130

5.12.2 Zeilenlänge 131

5.12.3 Groß- und Kleinschreibung 132

5.12.4 Shebang und Zeichenkodierung 133

5.13 Verzweigungen **135**

5.13.1 Fallunterscheidungen: if ... elif ... else 135
5.13.2 Fehlerbehandlung 139
5.14 Programmschleifen 142
5.14.1 Bedingte Schleifen mit „while“ 142
Aussprung mit break.....144
Unstrukturierte Programmierung.....146
5.14.2 Verkürzte Arithmetiknotation 146
5.14.3 Iterationsschleifen mit „for“ 147
5.14.4 Die Funktion range() 147
5.14.5 Generatoren 149
Generatorausdrücke.....150
5.14.6 Else und die Schleifen 151
5.14.7 Verschachtelte Schleifen 153
5.15 Sequenzen 155
5.15.1 Listen 155
Listen aus Listen.....156
5.15.2 Tupel 157
5.15.3 Mengen (Sets) 157
5.15.4 Dictionarys 158
5.15.5 Indizes 159
5.15.6 Schleifen über Sequenzen 160
5.15.7 Sequenzabschnitte (Slices) 161
5.15.8 Kopieren einer Sequenz 162
Kopien verschachtelter Sequenzen.....164
5.15.9 Umwandlung eines Generator-Objektes in eine Liste 165
5.15.10 Sequenzen sprengen 165
5.15.11 Das enumerate-Objekt 166
5.15.12 Reißverschlussverfahren: das Zip-Objekt 167
5.15.13 Funktionen für Sequenzen 169
5.15.14 Löschen von Sequenzen 170
5.15.15 Methoden von Listen 171

5.16 Anwendung von Listen: Vektoren 175
5.16.1 Vektoraddition 175
5.16.2 Skalarprodukt 176
5.16.3 Formatierte Ausgabe eines Vektors 177
5.17 Eigene Funktionen definieren 179
5.17.1 Eingangswerte (Argumente) 180
5.17.2 Vorbelegte Eingangswerte 181
5.17.3 Beliebig viele Argumente 181
5.17.4 Reihenfolge von Funktionsargumenten 182
5.18 Sichtbarkeit von Variablen 184
5.19 Klassen und Objekte 185
5.19.1 Attribute von Objekten 186
5.19.2 Methoden von Objekten 188
5.19.3 Die Methode __init__ 190
5.19.4 Vererbung 191
Wir bauen uns eine Durchreiche.....191
5.20 Eigene Module 194
5.20.1 Modulpfade 196
5.20.2 Funktionsüberschreibungen 197
5.21 Zeichenketten 200
5.21.1 Anführungszeichen in Zeichenketten 200
5.21.2 Der Rückwärtsschrägstrich 200
5.21.3 Mehrzeilige Ausgabe 201
5.21.4 Zeichenketten-Methoden 202
.count(Suchtext).....203
.encode(Kodierung, Fehlerbehandlung).....203
.endswith(Suchtext).....204
.find(Suchtext).....204
.isalnum().....204
.isalpha().....205
.isascii().....205

.isdecimal()....206
.join(iterierbares Objekt)....206
.lower()....206
.replace(alt, neu)....207
.split(Trennzeichen)....207
.startswith(Suchtext)....208
.strip(abzustreifende Zeichen)....208
.upper()....208
5.21.5 Formatierung mit Platzhaltern 209
5.21.6 F-Strings 210
5.21.7 Die Methode .format() 211
5.21.8 Die Formatierungs-Mini-Sprache 212
Einige Beispiele....212
5.21.9 Die alte printf-kompatible Formatierung 216
Vergleich mit C....217
Vergleich mit Java....218
Übersicht....218
5.21.10 Kodierung und Dekodierung 219
5.21.11 Komprimierung und Verschlüsselung 220
Simple Verschlüsselung....223
5.21.12 Sonderformen von Zeichenketten 223
B-Strings....223
U-Strings....224
R-Strings....224
F-Strings....225
5.22 Dateien lesen und schreiben 226
5.22.1 Textdateien lesen 226
5.22.2 Textdateien schreiben 229
5.22.3 Textdateien fortsetzen 229
5.22.4 Binärdateien 229
5.22.5 Pickle 230

5.22.6 Das aktuelle Arbeitsverzeichnis 232
5.23 Diagramme mit Matplotlib 235
5.23.1 Ein schnelles x-y-Diagramm 235
5.23.2 Ein schönes x-y-Diagramm 238
5.23.3 Streudiagramme 242
5.23.4 Text 244
5.23.5 gefüllte Flächen 245
5.23.6 Zeichenreihenfolge 246
5.23.7 3D-Diagramme 248
5.24 Grafik mit Tkinter 251
5.24.1 Das Hauptfenster 252
5.24.2 untergeordnete Fenster 254
5.24.3 Canvas – die Leinwand 257
5.24.4 Koordinaten der Canvas 258
5.24.5 Koordinatentransformationen 259
5.24.6 Linien und Linienzüge 260
Die Canvas-ID.....261
5.24.7 Pfeilspitzen 262
5.24.8 Gestrichelte Linien 263
5.24.9 Splines (Kurvenlinien) 264
5.24.10 Geschlossene Polygone 264
5.24.11 Rechtecke und Ellipsen 266
5.24.12 Kreise 267
5.24.13 Text 267
Schriftart, Auszeichnung und Schriftgröße.....268
5.25 GUI – Grafische Benutzungsoberflächen 269
5.25.1 EVA und die Events 269
Beispiel für einen Eventhandler.....269
5.25.2 Anordnung der GUI-Elemente 275
5.25.3 Die drei Geometriemanager 276
Pack.....276

Place....278
Grid....279
5.25.4 GUI-Widgets 281
Taste: Button....282
Beschriftung: Label....282
Eingabefeld: Entry....284
Schieberegler: Scale....286
Rahmen: Frame....289
Beschrifteter Rahmen: LabelFrame....290
Schiebefenster: PanedWindow....294
Ankreuzkästchen: Checkbutton....296
Radiobutton....299
Menubutton....301
5.26 Webserver 303
5.26.1 Zeichenkodierung 304
5.27 Logische Aussagen 305
5.27.1 Wahrheitswerte anderer Datentypen 305
5.27.2 Vergleichsoperatoren 306
5.27.3 Logische Aussagen über Gleitkommazahlen 307
5.27.4 Boolesche Algebra 308
Die Konjunktion: and....308
Die Disjunktion: or....309
Die Negation: not....310
Die Kontravalenz: ^....310
Prioritäten....311
Umkehrung logischer Aussagen....311
Boolesche Variablen....311
5.27.5 Venn-Diagramme 312

6 Datenspeicherung und Zahlensysteme 314
6.1 Bits und Bytes 314

6.1.1 Das Bit 314
6.1.2 Das Byte 315
6.1.3 Das Hexadezimalsystem 317
6.2 Zeichenkodierung – von ASCII bis Unicode 319

7 Anhang 324

7.1 Häufige Fehlermeldungen 324
7.2 Farben und Farbnamen (Auswahl) 326
7.3 Der Windows-Paketmanager WinGet 332
7.4 Abbildungsverzeichnis 337
7.5 Links und Literaturhinweise 341
7.6 Lizenz 344
7.7 Download und Feedback 345

1 Einleitung

I wish I understood what this was. But it's kind of nice.
Eric Idle[1]

Dieses Buch ist eine Ergänzung zu der seit 2009 von mir gehaltenen Vorlesung „Bauinformatik“ im Fachbereich Bau- und Umweltingenieurwesen der Hochschule Bochum. Sie finden hier außer vielen Inhalten der Vorlesung auch einige Zusatzinformationen, die in ihrer Breite nicht in Hörsaal-Lehrveranstaltungen passen.

In den Vorlesungen und Übungen an der Hochschule lernen wir wichtige Konzepte der Informatik kennen, indem wir uns selbst sprachliche Werkzeuge schaffen, mit denen wir ingenieurmäßige Probleme lösen werden. Wir bauen diese Werkzeuge, indem wir Handlungsanweisungen in einer Sprache formulieren, die ein Computer interpretieren und ausführen kann – einer Programmiersprache.

Das vorliegende Werk ist kein Vorlesungsskript im klassischen Sinne. Die Reihenfolge der Kapitel im Buch ist nicht dieselbe wie in der Vorlesung, da ich diesen Text zum leichteren Nachschlagen nach Sachthemen gegliedert habe. Außerdem werden wir in den Praktika und Übungen manche Techniken gelegentlich schon kurz kennenlernen, die erst später im Semester ausführlich behandelt werden. Diese Vorgriffe tauchen im Buch nicht auf.

Im Wintersemester 2022/23 werden erstmals seit Beginn der Covid-19-Pandemie wieder Präsenzvorlesungen im Hörsaal stattfinden. Beibehalten wird jedoch das bewährte Verfahren, über unser E-Learning-System Moodle interaktive Texte und wöchentliche Aufgaben anzubieten, zu denen die Studierenden zur Kontrolle des Lernstandes Rückmeldungen erhalten. Auf dem zum Kurs gehörenden Peertube-Kanal[2] finden Sie eine ständig wachsende Zahl von Videos zum Kurs.

1 https://twitter.com/EricIdle/status/1355227039146467329
2 https://tube.tchncs.de/c/python/videos

In Präsenz werden auch Übungen und Tutorien in kleinen Gruppen durchgeführt. Hierzu sind im Moodle-Kursbereich für eingeschriebene Studierende nähere Informationen verfügbar.

Im Gegensatz zu einem Papierbuch verändert sich der Inhalt in dieser PDF-Datei gelegentlich. Sie sollten daher mit Zitaten vorsichtig sein, wenn Sie einen wissenschaftlichen Anspruch an Ihre Arbeit haben. Zum Wintersemester 2018/19 verschwand beispielsweise das komplette Kapitel über Matrizenrechnung mithilfe verschachtelter Schleifen aus diesem Werk, weil es sich gezeigt hatte, dass dieser Themenbereich erhebliche Schwierigkeiten hatte, die „Mathe-Hirn-Schranke“ zu passieren.

Falls Sie diese PDF-Datei auf einem Mobilgerät ansehen, sollten Sie einen PDF-Betrachter verwenden, der Verknüpfungen (Links) unterstützt[1]. Durch Antippen der Seitenzahlen im Inhaltsverzeichnis können Sie so beispielsweise direkt zum jeweiligen Kapitel springen. Auch externe Links auf Webseiten funktionieren dann.

Trotz des Namens „Pythonbuch“ im Downloadlink ist dieser Text nicht als eigenständiges Selbstlernbuch konzipiert und stellt kein umfassendes Kompendium zur Programmiersprache Python dar. Falls es Ihnen dennoch gelingt, ohne die dazugehörenden Lehrveranstaltungen das Programmieren in Python mit diesem Buch zu lernen, oder wenn Sie einen interessanten Fehler im Text entdecken, schreiben Sie mir! Meine Adresse steht auf der letzten Seite.

1 Ein Beispiel für eine geeignete App ist das kostenlose und werbefreie PDF-Betrachtungs- und Bearbeitungsprogramm XODO: https://www.xodo.com/

1.1 Bedeutung der Bauinformatik

Die Anwendung von Computern, ob in stationärer oder mobiler Form, ist eine Kulturtechnik geworden, in die wir von Kindheit an hineinwachsen. Als Ingenieurinnen und Ingenieure stehen wir aber vor der Aufgabe, nicht nur nach Anleitung fertige Apps und andere Computerprogramme zu bedienen, sondern den Computer auch als individuell formbares Werkzeug zur Lösung von nicht standardisierten Problemen einzusetzen.

Die Grenzen populärer Bürosoftware sind mitunter schneller erreicht, als es uns lieb ist, doch oft können wir scheinbar komplexe Probleme mit wenigen Zeilen Programmcode elegant und schnell lösen. Zwar ließen sich viele dieser Aufgaben mit Ausdauer, Fleiß und Überstunden auch ohne Programmierkenntnisse bewältigen, sie würden dann aber deutlich weniger zur Arbeitsfreude beitragen. Viele scheinbar absurde Phänomene bei der Anwendung von Standardsoftware lassen sich zudem erst dann verstehen, wenn wir wenigstens eine ungefähre Ahnung davon haben, was gerade „unter der Motorhaube“ geschieht.

Informatik ist weit mehr als nur Programmierung, aber das Schreiben von Computerprogrammen wird in diesem Kurs der rote Faden sein, der sich durch unsere fünfmonatige Reise durch die Welt der formalen Sprachen, der Datenverarbeitung und der Algorithmen zieht.

1.2 Ermutigung

Es gibt an vielen Schulen hervorragende Informatiklehrer und -lehrerinnen, die fachlich kompetent und pädagogisch engagiert sind und Wunderbares leisten. Leider gibt es auch andere. Falls Sie seit dem Informatikunterricht in Ihrer Schule der Meinung sind, Computer niemals im Leben verstehen zu werden, dafür aber eine unerklärliche Abneigung gegen Hamster entwickelten: vergessen Sie am besten alles, was sie dort gelernt haben, bevor Sie weiterlesen!

Haben Sie den Mut, Dinge auszuprobieren! Sie lernen nicht, zu programmieren, indem Sie ein Buch durchlesen oder ein Video ansehen. Sie lernen es vor allem, indem Sie eigenhändig Programme schreiben, Fehler machen (das ist wirklich wichtig!), Fehlermeldungen lesen und verstehen sowie die Fehlerursachen finden, begreifen und beseitigen. Immer wieder.

Seien Sie aktiv! Damit Sie wirklich etwas lernen, benötigen Sie außer Ihren Augen und Ohren auch Ihre Hände. Schreiben Sie in der Vorlesung und auch beim Betrachten von Videos mit, machen Sie sich Notizen mit Stift und Papier – vor allem, wenn etwas unklar scheint! Besprechen Sie offen gebliebene Fragen nach der Vorlesung mit ihrer Lerngruppe, suchen Sie die Antworten in diesem Text, im Internet oder in der Literatur!

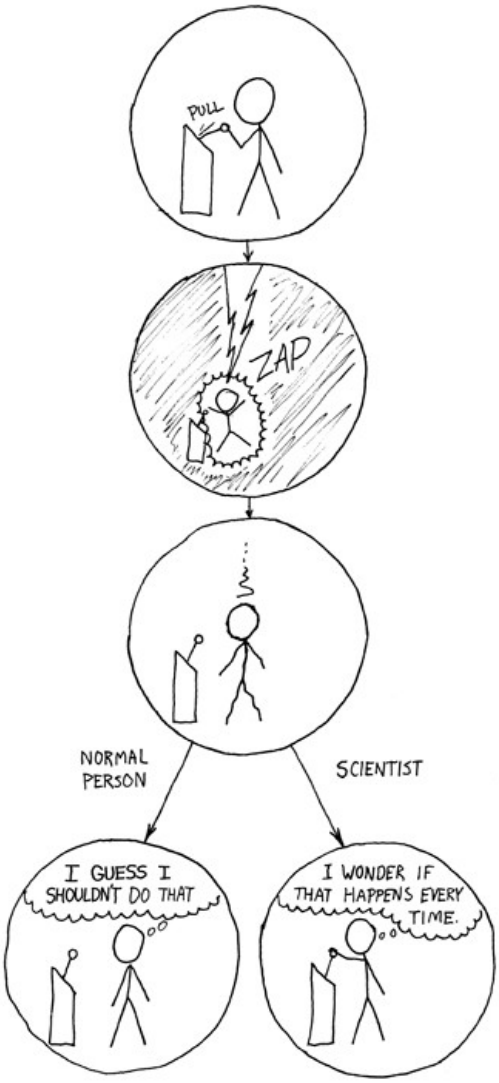


Abb. 1: The Difference (Randall Munroe)

https://xkcd.com/242/ Creative Commons Attribution-NonCommercial 2.5 License

1.3 Lerntipps

Bringen Sie kein Notebook oder Tablet mit in die Vorlesung! Es mag verführerisch erscheinen, vorgestellte Codeschnipsel gleich auszuprobieren oder die Vorlesungsmitschrift gleich in leserlicher Druckschrift zu erfassen. Ihnen entgeht jedoch durch das Tippen zu viel vom eigentlichen Sinn des Vorlesungsstoffs.[1][2][3][4][5][6]

Bearbeiten Sie vor allem die Übungsaufgaben selbst und besorgen Sie sich keine fertigen Lösungen! Sie geben die Übungen nicht ab, um zu beweisen, dass Sie in der Lage sind, fristgerecht eine Ware abzuliefern, sondern um bei der eigenen Arbeit daran die Lerninhalte zu vertiefen. Außerdem finden Sie nur so heraus, ob Sie das, was Sie zu wissen glauben, tatsächlich verstanden haben.

Fast jeder Mensch, der sich eine Weile mit einer guten Programmiersprache auseinandersetzt, entwickelt früher oder später eine gewisse Begeisterung fürs Programmieren und möchte gern noch interessantere und umfangreichere Probleme lösen, als wir im Rahmen dieser Veranstaltung behandeln können. Am Ende dieses Buches finden Sie im Kapitel 7.5

---

1 Besser lernen mit Stift statt Tastatur. heise online [online]. 4 Mai 2014. [Zugriff am: 8 September 2023]. Verfügbar unter: https://www.heise.de/news/Besser-lernen-mit-Stift-statt-Tastatur-2182057.html

2 For better learning in college lectures, lay down the laptop and pick up a pen. Brookings [online]. [Zugriff am: 8 September 2023]. Verfügbar unter: https://www.brookings.edu/articles/for-better-learning-in-college-lectures-lay-down-the-laptop-and-pick-up-a-pen/

3 PATTERSON, Richard W. und Robert M. PATTERSON, 2017. Computers and productivity: Evidence from laptop use in the college classroom. Economics of Education Review. 1 April 2017. Bd. 57, S. 66–79. Verfügbar unter: https://doi.org/10.1016/j.econedurev.2017.02.004

4 FISHER, Beth, 2015. Laptop Use in Class: Effects on Learning and Attention. Center for Teaching and Learning [online]. 22 August 2015. [Zugriff am: 8 September 2023]. Verfügbar unter: https://ctl.wustl.edu/laptop-use-effects-learning-attention/

5 The Impact of Computer Usage on Academic Performance: Evidence from a Randomized Trial at the United States Military Academy,. Blueprint Labs [online]. [Zugriff am: 8 September 2023]. Verfügbar unter: https://blueprintlabs.mit.edu/research/the-impact-of-computer-usage-on-academic-performance-evidence-from-a-randomized-trial-at-the-united-states-military-academy/

6 Using laptops in class harms academic performance, study warns, 2017. Times Higher Education (THE) [online]. [Zugriff am: 8 September 2023]. Verfügbar unter: https://www.timeshighereducation.com/news/using-laptops-in-class-harms-academic-performance-study-warns

(Links und Literaturhinweise) ein paar Empfehlungen, um Hilfen und Anregungen jenseits des Tellerrands der Erstsemestervorlesungen zu erhalten.

1.4 Suchmaschinentipps

Suchmaschinen wie Google, Bing oder Duckduckgo sind unentbehrliche Werkzeuge zum schnellen Auffinden von Informationen im WWW. Um in der Masse der Fundstellen gezielt die relevanten Seiten zu finden, lassen sich die Suchen verfeinern. Die folgenden Tipps beziehen sich auf die Suche mit Google:

Anführungszeichen fassen mehrere Wörter zu einer Phrase zusammen, die als Ganzes im zu findenden Text vorkommen muss. Die Suche nach **hochschule bochum** findet Texte, in denen sowohl das Wort „Hochschule“ als auch das Wort „Bochum“ vorkommen. Die Suche nach **"hochschule bochum"** dagegen beschränkt die Fundstellen auf Texte zur Hochschule Bochum.

Minuszeichen kennzeichnen auszuschließende Begriffe. Wer Bilder von Jaguaren sucht, sollte die Suche entweder mit **jaguar -cat -katze** oder mit **jaguar -car -auto** durchführen, je nachdem, welche Ergebnisse nicht gewünscht sind. Ein Minuszeichen vor einer Phrase schließt diese als Ganzes von der Suche aus.

Die Suche nach Dateien eines bestimmten Typs oder Formats lässt sich mit dem Schlüsselwort „filetype“ beeinflussen. Die Suche nach **"python 3" bauinformatik filetype:pdf** sollte die PDF-Datei des Textes, den Sie gerade vor sich haben, recht weit oben auf der ersten Suchergebnisseite aufführen.

Sie können die Suche auf Seiten eines bestimmten Webauftritts begrenzen. **stundenplan site:hs-bochum.de** beschränkt die Suche nach Stundenplänen auf die Website der Hochschule Bochum. Der Zusatz **site:.de** findet nur Seiten, die für Deutschlands Top-Level-Domain DE registriert wurden.

Wenn Sie nur einen Teil der Adresse (URL) einer Seite kennen, können Sie auch danach suchen **stundenplan inurl:fbb**.

Zahlenbereiche können mit zwei Punkten festgelegt werden. Die Suche nach **Notebook "500..800 €"** findet Seiten über Notebooks mit Preisangaben zwischen 500 und 800 Euro.

Weitere Möglichkeiten, um im Idealfall genau das eine für Sie passende Suchergebnis zu finden, bietet das Formular zur erweiterten Suche auf der Website https://www.google.com/advanced_search.

2 PC-Grundkenntnisse

In den Vorlesungen und Übungen des Kurses „Bauinformatik“ setze ich gewisse im Umgang mit persönlichen Computern (PC) alltägliche Grundkenntnisse voraus. In diesem Kapitel des Skriptes erhalten Sie einen kurzen Überblick über einige Begriffe und Techniken, die sie verstanden haben sollten, wenn Sie sich mit den eigentlichen Themen dieses Semesters auseinandersetzen.

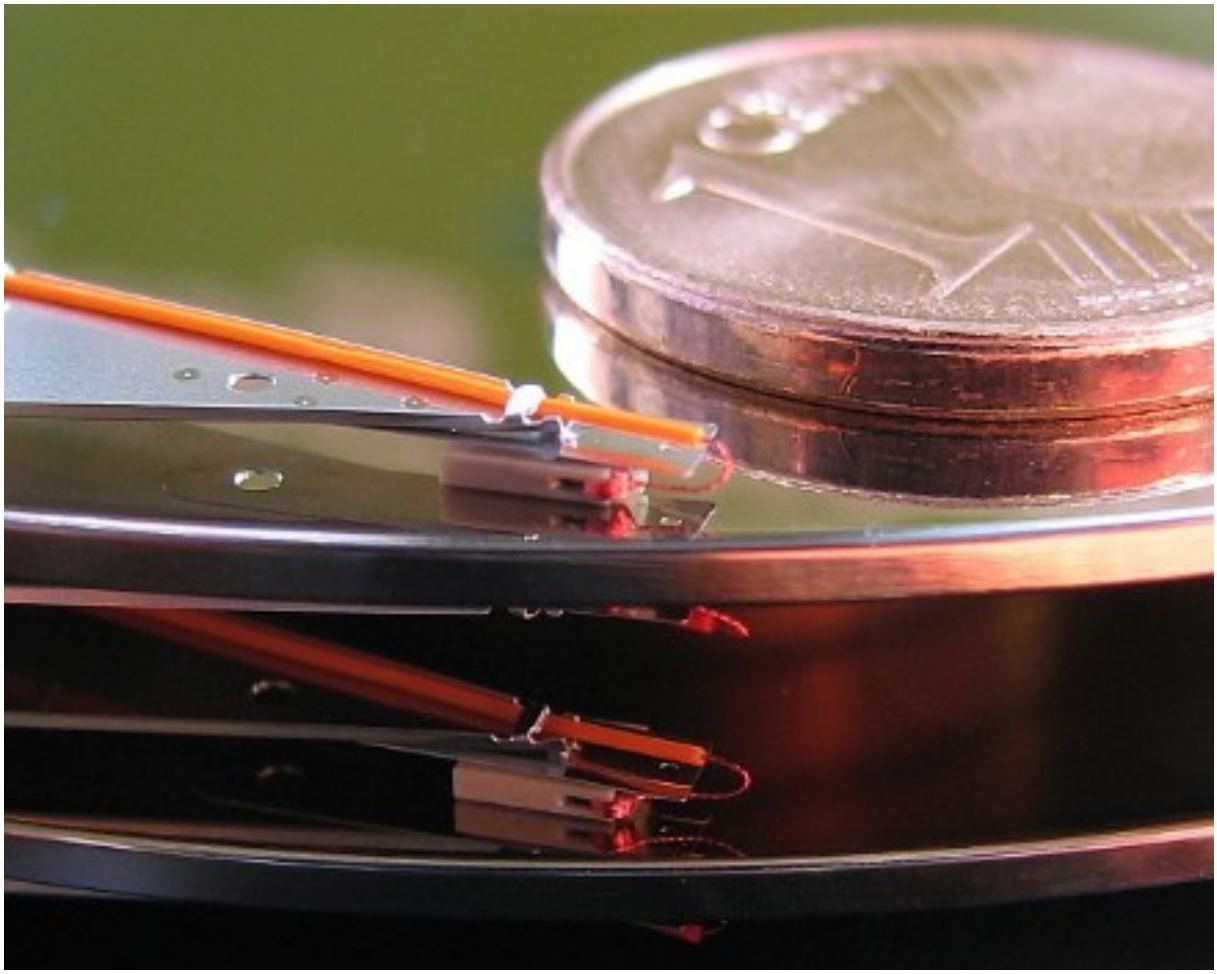


Abb. 2: Schreib-/Leseköpfe einer Festplatte im Größenvergleich

2.1 Tastatur

Die Tastatur eines PC orientiert sich an den im 19. Jahrhundert aufgekommenen Schreibmaschinentastaturen. Sie enthält auf mehreren Belegungsebenen die häufigsten in Texten verwendeten Buchstaben, Ziffern und Sonderzeichen sowie einige Steuer- und Funktionstasten.

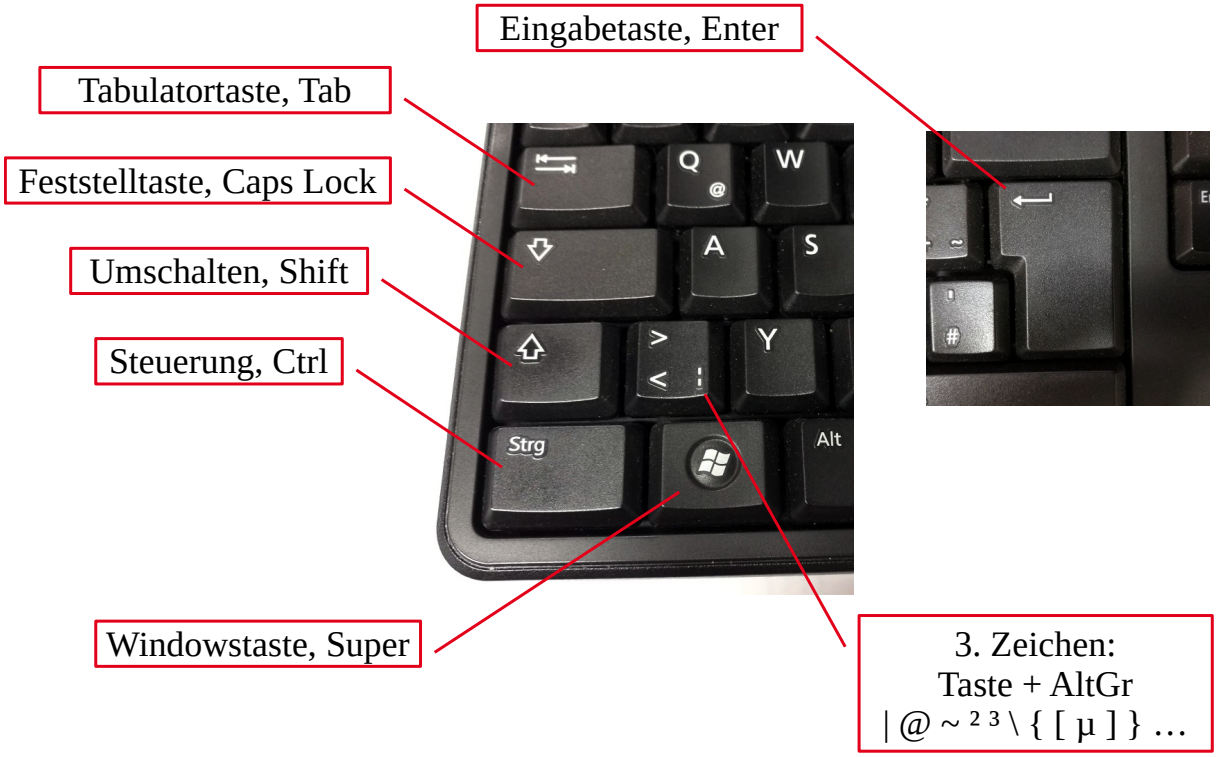


Abb. 3: Tastenbezeichnungen unter Linux und Windows

In Abb. 3 sind einige wichtige Tastenbezeichnungen aufgeführt, die in diesem Buch verwendet werden.

PCs mit den Betriebssystemen Linux und Windows verwenden in der Regel die gleichen Standardtastaturen; in seltenen Fällen ersetzt der Pinguin Tux das „Windows-Fähnchen“ auf der Super-Taste.

PCs der Firma Apple[1] verwenden abweichende Bezeichnungen und Symbole für einige Tasten.

1 Die Marketingabteilungen von Apple und Intel versuchen weiszumachen, es gebe zwei Sorten von schreibtischtauglichen Computern: den PC (mit Windows) und den Mac (mit macOS). Das ist Unfug.

Da aus gestalterischen Gründen insbesondere auf Apple-Tastaturen viele mit der Tastatur eingebbare Sonderzeichen nicht auf den Tastenkappen aufgedruckt sind, hier eine Eingabehilfe:

Zeichen	Name	Windows	Linux	macOS
@	at	AltGr Q	AltGr Q	⌥ L
[	eckige Klammer auf	AltGr 8	AltGr 8	⌥ 5
]	eckige Klammer zu	AltGr 9	AltGr 9	⌥ 6
{	geschweifte Klammer auf	AltGr 7	AltGr 7	⌥ 8
}	geschweifte Klammer zu	AltGr 0	AltGr 0	⌥ 9
/	Schrägstrich	⇧ 7	⇧ 7	⇧ 7
\	Rückwärtsschrägstrich	AltGr ß	AltGr ß	⌥ ⇧ 7
|	senkrechter Strich	AltGr <	AltGr <	⌥ 7
„	Anführungszeichen auf	Alt 0132	AltGr V	⌥ ⇧ W
“	Anführungszeichen zu	Alt 0147	AltGr B	⌥ 2
‚	einfaches Anf.-zeichen auf	Alt 0130	AltGr⇧V	⌥ S
‘	einfaches Anf.-zeichen zu	Alt 0145	AltGr⇧B	⌥ #
»	franz. Anfz. nach rechts	Alt 0187	AltGr Y	⌥ ⇧ Q
«	franz. Anfz. nach links	Alt 0171	AltGr X	⌥ Q
›	einf. franz. Anfz. nach rechts	Alt 0155	AltGr⇧Y	⌥ ⇧ N
‹	einf. franz. Anfz. nach links	Alt 0139	AltGr⇧X	⌥ ⇧ B
µ	My (Mikro-)	AltGr M	AltGr M	⌥ M
·	Multiplikationspunkt	Alt 0183	AltGr ,	⌥ ⇧ 9
…	Auslassungspunkte	Alt 0133	AltGr .	⌥ :
’	Apostroph	Alt 0146	AltGr #	⌥ ⇧ #
←	Pfeil nach links	Alt 8592	AltGr Z	
↓	Pfeil nach unten	Alt 8595	AltGr U	
↑	Pfeil nach oben	Alt 8593	AltGr⇧U	
→	Pfeil nach rechts	Alt 8594	AltGr I	
¹	hochgestellte 1	Alt 0185	AltGr 1	
²	hochgestellte 2	AltGr 2	AltGr 2	
³	hochgestellte 3	AltGr 3	AltGr 3	
≤	kleiner oder gleich	Alt 8804	⊡ < =	⌥ >
≥	größer oder gleich	Alt 8805	⊡ > =	⌥ ⇧ >
≠	ungleich	Alt 8800	⊡ / =	⌥ 0

Zeichen	Name	Windows	Linux	macOS
±	Plus/Minus	Alt 0177	◌ + -	⌥ +
−	Minuszeichen	Alt 8722	u2212	
–	Gedankenstrich	Alt 0150	AltGr -	⌥ _
-	Bindestrich	−	−	−
∞	Unendlich-Symbol	Alt 8734	◌ 8 8	⌥ ,
×	Multiplikationskreuz	Alt 0215	AltGr⇧,	
÷	Divisionsoperator	Alt 0247	AltGr⇧.	
π	Pi	Alt 0960	u3C0	
⌀	Durchmesserzeichen	Alt 8960	u2300	
√	Wurzelzeichen	Alt 8730	u221a	⌥ V
∑	Summenzeichen Sigma	Alt 8721	u2211	⌥ W
‰	Promillezeichen	Alt 0137	◌ % O	⌥ ⇧ R
	schmales festes Leerzeichen	Alt 8201	u2009	
	festes Leerzeichen	Alt 0160	Strg ⇧ ␣	⌥ ␣

In den rechten Tabellenspalten werden mehrere ungewöhnliche Symbole verwendet. Das Symbol ◌ steht hier für die unter Linux standardmäßig verfügbare Compose-Taste, mit der jeweils mehrere leicht zu merkende Tastendrücke zur Eingabe von Sonderzeichen kombiniert werden. Das Symbol ␣ bezeichnet die Leertaste. Das unterstrichene u symbolisiert die Tastenkombination Strg ⇧ U, welche unter Linux die Unicode-Eingabe aktiviert.

Feste Leerzeichen werden zwischen Zahlenwerte und Einheiten gesetzt, um eine automatische Trennung am Zeilenende zu verhindern. Schmale feste Leerzeichen trennen Tausendergruppen großer Zahlen.

MacOS besitzt eine systemweite Funktion zur Eingabe hoch- und tiefgestellter Zeichen. Dazu ist zunächst die gewünschte Zeichenfolge zu markieren. Die Tastenkombination ^⌘+ (Control-Command-Plus) wandelt die Zeichen dann in die entsprechenden hochgestellten Unicode-Zeichen um und ^⌘− bewirkt dasselbe zum Tiefstellen.

Unter macOS lässt sich in der Systemeinstellung unter „Sprache und Text → Eingabequellen“ bzw. „Tastatur → Eingabequellen“ die „Unicode-Hex-Eingabe“ aktivieren. Damit können die Zeichencodes bei gedrückter Taste ⌥ (Alt, Option) eingetippt werden.

Die Windows-Tastenkombinationen mit dem Schema „Alt Zahlencode“ werden eingegeben, indem Sie die Taste Alt mit der linken Hand gedrückt halten, während Sie auf dem numerischen Ziffernblock rechts nacheinander die entsprechenden Ziffern tippen. Beim Loslassen der Alt-Taste erscheint das gewünschte Zeichen. Wenn Ihre Tastatur keinen numerischen Ziffernblock hat, können Sie diese Eingabemethode nicht nutzen. Die Zifferntasten der oberen Tastenreihe werden von Windows zur Codeeingabe nicht akzeptiert.

Etwas ungewöhnlich bei der Code-Eingabe unter Windows ist die Verwendung einer Dezimalzahl. Üblicherweise werden Unicode-Symbole über ihre Hexadezimalcodes[1] adressiert.

Um auch unter Windows den Luxus einer Compose-Taste und einer Unicode-Eingabe über Hexadezimalcodes nachzurüsten, lässt sich das Programm „WinCompose“ installieren[2]. Es erlaubt zudem, die selten willentlich verwendete Großbuchstabenfeststelltaste zur gut erreichbaren Compose-Taste umzudefinieren.

1 Mehr dazu in Kapitel 6.1.3

2 **`CMD`** → **`winget install wincompose`** – siehe Kapitel 7.3

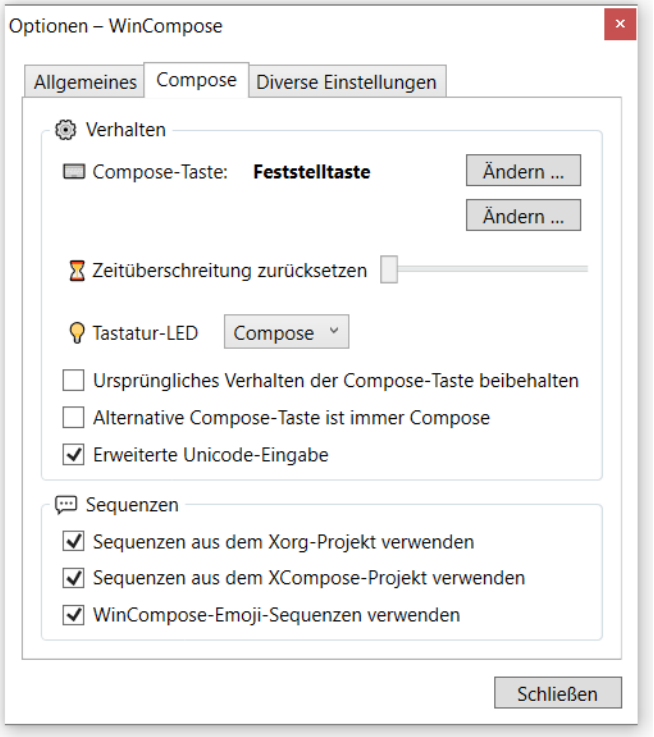


Abb. 4: WinCompose rüstet auch eine Unicode-Eingabe nach

2.2 Betriebssystem

Ein Betriebssystem verbindet die Anwendungssoftware (das können beispielsweise kommerzielle oder freie Officepakete, heruntergeladene Apps oder beliebige andere ausführbare Programme sein) mit der Hardware (dem Gerät, auf dem die Programme laufen sollen).

Etwas umfassender ausgedrückt bezeichnet man als Betriebssystem eine Programmsammlung, die Anwendungsprogrammen standardisierte Werkzeuge zum Zugriff auf interne und externe Geräte, gespeicherte Daten und Datenträger sowie die Kommunikationskanäle eines Rechnersystems zur Verfügung stellt.

Häufig werden Betriebssysteme noch mit einer Vielzahl mehr oder weniger nützlicher Programme ausgeliefert. Wir nennen so ein Paket aus Betriebssystemkern und Anwendungsprogrammen eine Distribution. Während zu Linux-Distributionen in der Regel ein oder mehrere Officepakete, ein breites Sortiment an Programmiersprachen und tausende problemspezifischer Anwendungsprogramme aus dem wissenschaftlich-technischen Bereich gehören, wird das oft als unvermeidlich angesehene Windows in der Regel nur mit dem Allernötigsten zum Betrieb des Rechners sowie einem Haufen Bloatware in Form von laufzeitbeschränkten Demoversionen kommerzieller Softwareprodukte verkauft. Erst seit dem Jahr 2020 wird mit Windows der einfache Paketmanager „WinGet“ ausgeliefert, mit dem sich über das Textterminal eine Vielzahl von Softwarepaketen installieren lässt, ohne diese zuvor im Handel erwerben zu müssen oder unter dem Risiko, sich stattdessen Schadsoftware einzufangen, auf Downloadseiten suchen zu müssen. Eine Kurzanleitung zu WinGet befindet sich im Anhang dieses Textes (Kapitel 7.3).

Das auf Desktop-PCs und Notebooks im Ingenieurbereich hierzulande meistverbreitete Betriebssystem ist derzeit Microsoft Windows, gefolgt von Linux und dem damit entfernt verwandten macOS. Auf mobilen Kleingeräten wie Smartphones und Tablets sorgen in der Mehrzahl die Betriebssysteme Android (ebenfalls Linux) und iOS für die Kommunikation zwischen den einzelnen Programmen und der Gerätetechnik.

Auf größeren Rechnern, zum Beispiel den Servern publikumsintensiver Webauftritte, den Supercomputern in Rechenzentren und den Datenzentren von Cloudspeicherdiensten kommt nahezu ausschließlich Linux zum Einsatz.

2.3 Dateien und Verzeichnisse

Eine Datei ist eine zusammengehörige Gruppe von Informationen, die in einem Verzeichnis eines Dateisystems durch einen eindeutigen Dateinamen identifizierbar ist.

Abhängig von ihrem Inhalt unterscheiden wir zwischen Grafikdateien, Textdateien, Videodateien, Programmdateien und zahlreichen anderen Dateitypen.

Im Zusammenhang mit Dateien hören wir oft, dass diese „geöffnet“ oder „geschlossen“ werden. Die Anmeldung eines Programms beim Betriebssystem zum lesenden oder schreibenden Zugriff auf eine Datei nennt man „Öffnen“. Greift ein Programm nicht mehr auf eine Datei zu, meldet es seine Zugriffserlaubnis wieder ab. Die Datei wird „geschlossen“.

Solange eine Datei nicht geschlossen wurde, können wir uns nicht darauf verlassen, dass von unserem Programm abgeschickte Daten wirklich in der Datei angekommen sind – möglicherweise sind sie noch in einem Zwischenspeicher des Betriebssystems, wo sie auf einen günstigen Augenblick warten, um tatsächlich in die Datei zu gelangen. Bis die Datei dann allerdings vollständig auf ihren Datenträger (Festplatte, USB-Stick, etc.) geschrieben wird, dauert es auch nach ihrem Schließen manchmal noch eine gewisse Weile. Bei langsamen USB-Sticks und großen Dateien müssen wir sogar mehrere Minuten Geduld aufbringen. Das Abziehen eines USB-Sticks direkt nach der Meldung, dass das Schreiben abgeschlossen sei, kann deshalb schlimmstenfalls zum Verlust sämtlicher gerade zu schreibender Inhalte führen.

Besonders unter Microsoft Windows ist das frühzeitige Schließen einer Datei wichtig, da dieses Betriebssystem regelmäßig den lesenden Zugriff auf eine Datei verbietet, solange diese noch von irgendeinem Programm geöffnet ist. Seit Windows 8 führt das beispielsweise zu manch ärgerlicher Situation beim Erstellen von PDF-Dateien, da der PDF-Betrachter der Firma Adobe in die Vorschaufunktion des Windows-Dateimanagers „Explorer“ eingebunden wurde und die unsinnige Eigenart hat, die zuletzt gelesene Datei dauerhaft geöffnet zu halten, selbst wenn das Programm scheinbar geschlossen wurde, tatsächlich jedoch noch unsichtbar im Hintergrund läuft. Erstellt man nun mit einem beliebigen Programm

eine PDF-Datei und entdeckt darin beim Probelesen einen Fehler, so ist es mitunter nicht möglich, eine neue PDF-Datei nach der Korrektur unter demselben Namen wie zuvor zu speichern. Verwenden Sie dann beim Speichern einen anderen Namen für Ihre PDF-Datei oder greifen Sie zu einem benutzungsfreundlicheren Betriebssystem.

2.3.1 Verzeichnisbäume

Dateien werden üblicherweise in einer hierarchischen, baumähnlichen Verzeichnisstruktur organisiert. Jedes einzelne Verzeichnis kann außer Dateien auch wieder andere Verzeichnisse enthalten, diese nennen wir Unterverzeichnisse. Im botanischen Gegenstück entsprechen die Dateien den Blättern und die Verzeichnisse den Ästen und Zweigen.

Kurioserweise wird die Wurzel eines Verzeichnisbaums meistens als obenliegend angesehen, was ihn signifikant von seinen botanischen Verwandten unterscheidet:

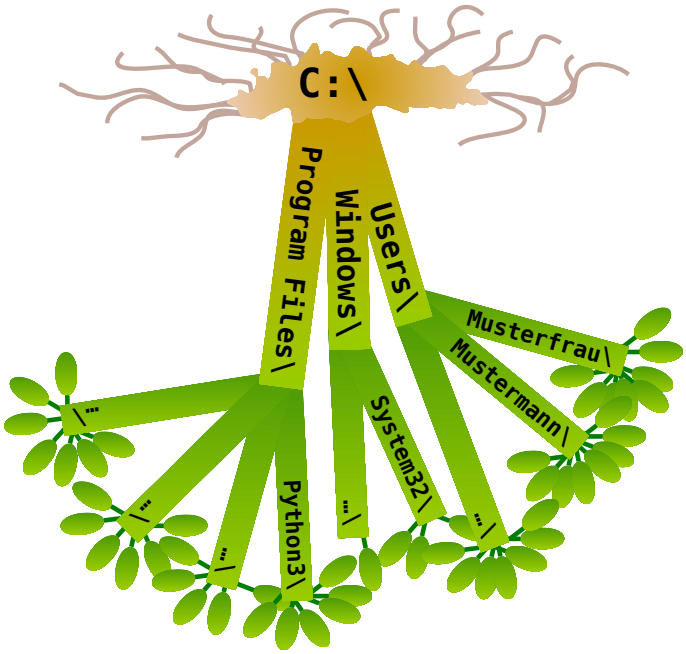


Abb. 5: Teil des Verzeichnisbaums unter Microsoft Windows

2.3.2 Versteckte Dateinamenerweiterungen

Für Programme, die Dateien verarbeiten, ist es erforderlich, den genauen Namen einer Datei und des Verzeichnisses, in dem diese sich befindet, zu kennen. Leider wird beides im Windows-Explorer in der Regel nicht oder sogar falsch angezeigt.

Dateien verfügen seit den Anfängen des Personal Computers über ein Suffix am Ende des Dateinamens, das Windows auch heute noch zur Bestimmung des Dateityps verwendet. Dieses Suffix, die sogenannte Dateinamenerweiterung oder Extension, beginnt mit einem Punkt und ist meistens zwei bis vier Buchstaben lang.

Die Namen ausführbarer Windows-Programme enden beispielsweise in der Regel auf „.exe“, Textverarbeitungsdateien auf „.doc“, „.docx“ oder „.odt“ und Python-Programme auf „.py“ oder „.pyw“.

Unter anderen Betriebssystemen wie macOS oder Linux gibt es diesen Zwang zur Dateinamenerweiterung nicht. Der Dateityp hängt dort auch vom Inhalt der Datei ab, nicht nur von ihrem Namen. Um diese Benutzungsfreundlichkeit vorzutäuschen, versteckt der Windows-Explorer daher seit Windows XP bei manchen bekannten Dateitypen die vorhandenen Dateinamenerweiterungen vor den Anwenderinnen und Anwendern.

Das hat nicht nur den unangenehmen Nebeneffekt, dass Kriminelle immer wieder erfolgreich ausführbare Windows-Programme durch ihre Opfer starten lassen, weil das jenen zugeschobene ausführbare Programm im Explorer wie eine harmlose Bild- oder Textdatei gelistet wird, es bewirkt auch, dass wir Dateien im Windows-Explorer nicht mehr vollständig umbenennen können. Eine neu angelegte Textdatei „Berechnung.txt“ wird im Windows-Explorer beispielsweise nur als „Berechnung“ angezeigt. Handelt es sich bei der Textdatei aber um ein Python-Programm, so sollte es stattdessen auf „.py“ enden, um durch Doppelklick gestartet oder durch Rechtsklick mit IDLE geöffnet zu werden.

Wir können zwar versuchen, die Datei umzubenennen, doch hilft uns das zunächst nicht weiter. Der Windows-Explorer zeigt die umbenannte Textdatei anschließend zwar irreführenderweise als „Berechnung.py“ an, führt das darin enthaltene Programm jedoch beim Doppelklicken immer

noch nicht aus, sondern lädt die Datei weiterhin nur in den Texteditor. In Wirklichkeit heißt sie nun nämlich „Berechnung.py.txt“ und wird von Windows daher immer noch als Textdatei behandelt.

Um diese unnötigen Probleme loszuwerden, sollten Sie im Windows-Explorer jeder Windows-Installation, der Sie begegnen, eine Einstellung vornehmen, die dafür sorgt, dass Dateinamen grundsätzlich unverstümmelt angezeigt werden.

Im Windows-Explorer von Microsoft Windows 7 muss dazu unter „Organisieren → Ordner- und Suchoptionen → Ansicht → Erweiterte Einstellungen“ das Häkchen vor „Erweiterungen bei bekannten Dateitypen ausblenden“ entfernt werden.

Unter Windows 10/11 drücken Sie die Windowstaste ⊞ und tippen das Wort „Ordneroptionen“. Sie erreichen so die hinter der etwas schrägen Bezeichnung „Suchoptionen für Dateien und Ordner ändern“ versteckten „Explorer-Optionen“ der Systemsteuerung.

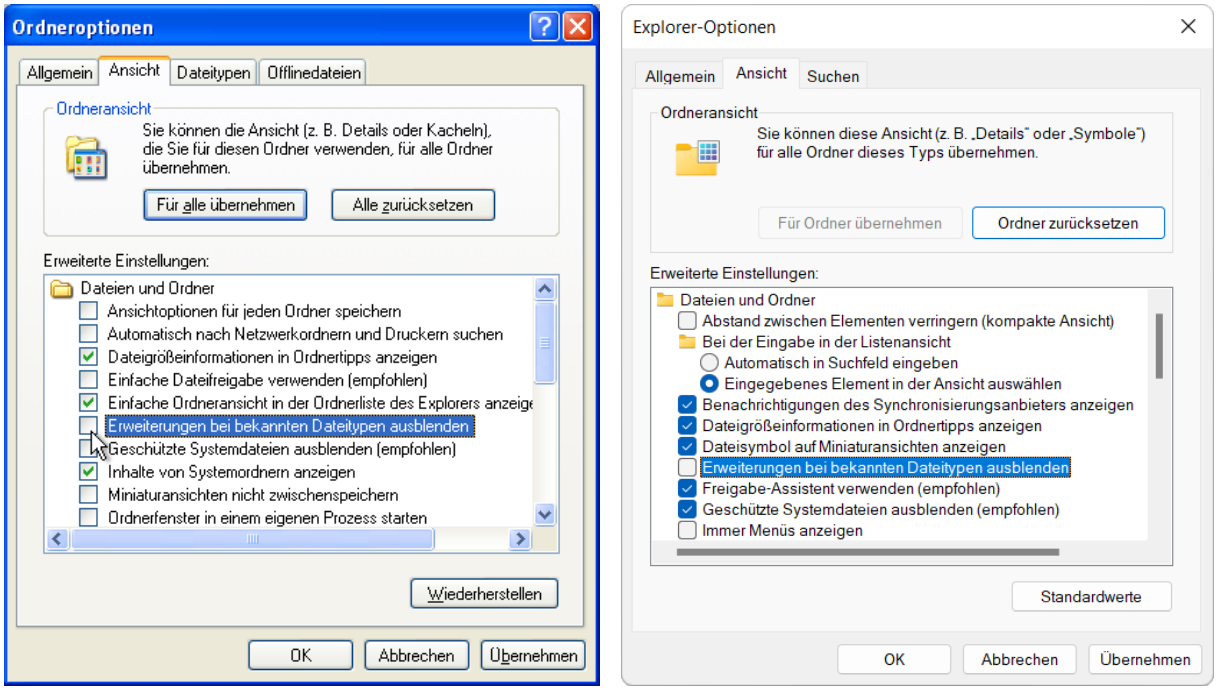


Abb. 6: Ordneroptionen in Windows XP (2001) und Windows 11 (2022)

Seit einiger Zeit blendet auch macOS bei einigen Dateitypen den hinteren Teil von Dateinamen aus. Das kann für jede einzelne Datei in deren Eigenschaftendialog (Cmd-i) deaktiviert werden. In den Einstellungen des

Finders lässt sich dieses Verhalten allerdings auch gleich systemweit abstellen. Dazu muss dort das Häkchen vor „Alle Suffixe einblenden“ gesetzt werden (Abb. 7).

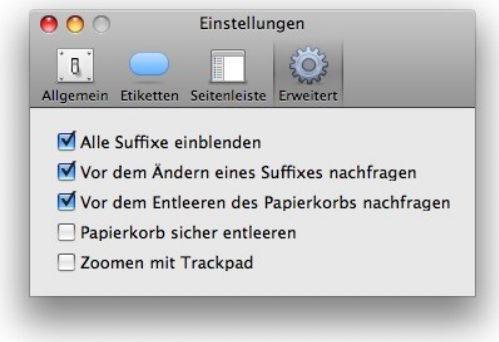


Abb. 7: Dateinamenerweiterungen bei macOS heißen Suffixe

2.3.3 Desktop, Ordner und Verzeichnisse

In fast allen modernen PC-Betriebssystemen gibt es das Konzept einer Arbeitsfläche, auf der Dateien und Verzeichnisse oder Verweise darauf zum schnellen Zugriff abgelegt werden können.

Diese Arbeitsfläche ist üblicherweise bildschirmfüllend und taucht im Dateimanager des Betriebssystems unter Namen wie „Desktop“ oder „Schreibtisch“ auf. Auch aus diesem Grund werden Betriebssysteme wie Microsoft Windows, Ubuntu Linux oder macOS heute als Desktop-Betriebssysteme bezeichnet[1].

Unter Microsoft Windows finden wir weitere besondere Orte zur Dateiablage in der baumartigen Anordnung im linken Panel des Dateimanagers „Windows-Explorer“. Diese Verweise auf bestimmte Verzeichnisse werden als „Ordner“ bezeichnet. Wir sehen im Bereich „Schnellzugriff“ zum Beispiel häufig benötigte Ordner wie den Desktop oder den Ordner „Down-

1 Der andere Grund für diese Benennung ist, dass sich Computer mit Desktop-Betriebssystemen in der Regel auf, neben oder unter Schreibtischen befinden. Das führte zu der Kuriosität, dass Notebooks einige Jahre lang „Laptops“ hießen, da man sie ja auf dem Schoß (engl. *lap*) bedienen konnte. Die Bezeichnung wurde etwa zu dem Zeitpunkt unpopulär, als sich herumsprach, dass die Nähe heißer Gerätelüfter zu männlichen Keimdrüsen unerwünschte Wirkungen nach sich zieht.

loads“ für die vorübergehende Aufbewahrung heruntergeladener Dateien. Diese Liste lässt sich mit einem Rechtsklick auf die Einträge beliebig kürzen oder erweitern.

Unterhalb der Schnellzugriffseinträge gibt es einen Bereich mit dem Namen „Dieser PC“, in dem ebenfalls der Desktop und der Download-Ordner sowie die persönlichen Ordner für Bilder, „Dokumente“, Musik und Videos aufgeführt sind.

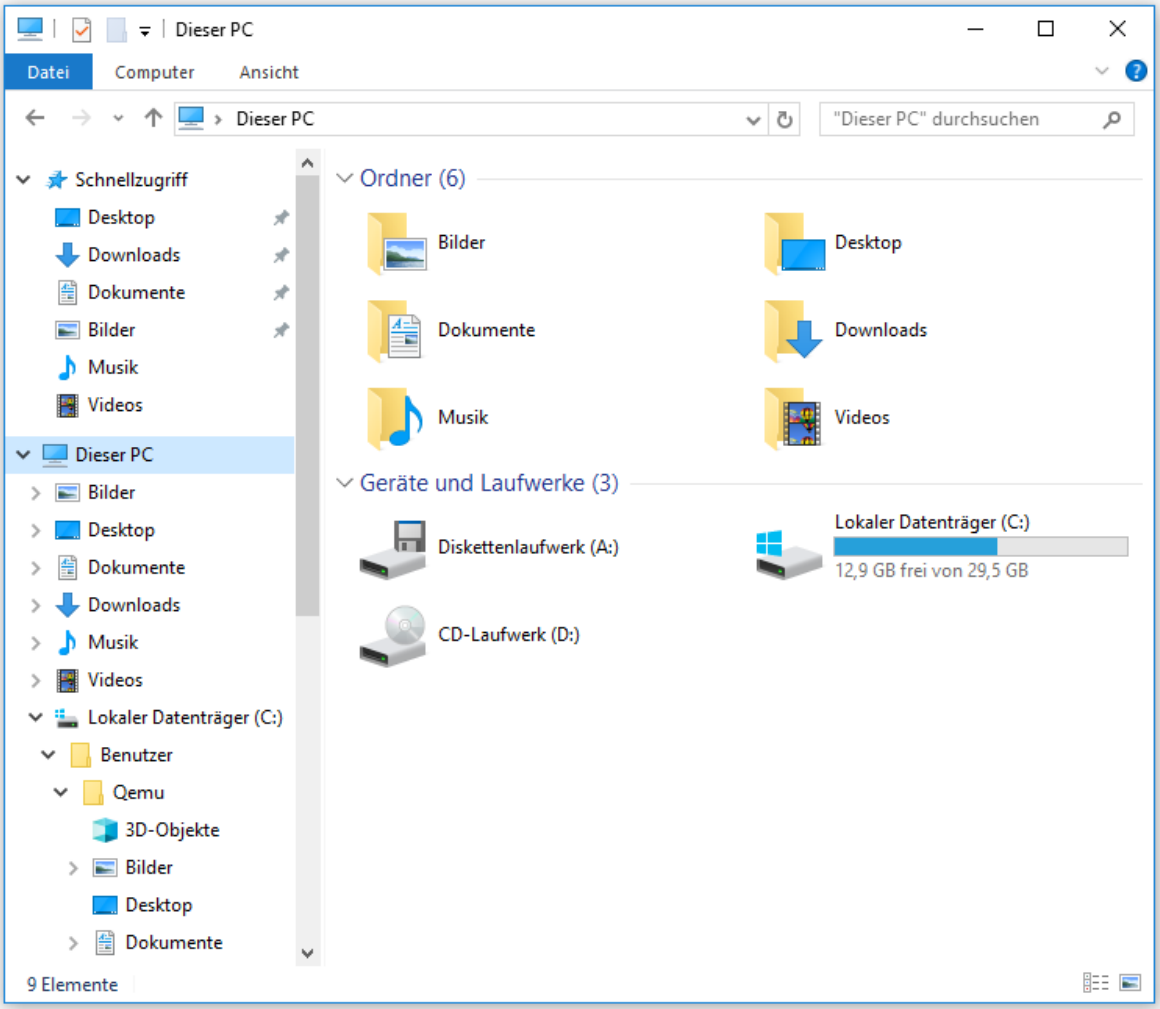


Abb. 8: Bibliotheken als „Dieser PC“ im Windows-10-Explorer

Wo genau sich diese Ordner im Dateisystem der Festplatte oder anderer angeschlossener Datenträger befinden, ist oft gar nicht auf Anhieb erkennbar. Kurioserweise ist die tatsächliche Verzeichnisstruktur unter Microsoft Windows der im Explorer dargestellten Ordnung fast genau entgegengesetzt.

Oberste Ordnungsinstanz sind in Wirklichkeit die Datenträger. Diese sind in eine oder mehrere Partitionen unterteilt. Jeder von Windows lesbaren Partition ist dabei ein Buchstabe zugeordnet. Diese Buchstaben werden in der Regel fortlaufend vergeben. Die Buchstaben „A:“ und „B:“ sind für die im letzten Viertel des vorigen Jahrhunderts in fast jedem PC zu findenden Diskettenlaufwerke reserviert. Die Festplatte mit dem Windows-Betriebssystem, den Anwendungsprogrammen und den persönlichen Daten heißt daher auch heute immer noch nach dem ersten damals freien Buchstaben „C:“. Rechner, die noch über ein optisches Laufwerk für CDs oder DVDs verfügen, sehen dies in der Regel unter Windows als Laufwerk „D:“ und für USB-Sticks und alle weiteren Laufwerke stehen die Buchstaben von „E:“ bis „Z:“ zur Verfügung.

Unterhalb des Stammverzeichnisses „C:\“ befinden sich die Verzeichnisse für das Betriebssystem („C:\Windows“), für Programme („C:\Program Files“) und für die persönlichen Dateien („C:\Users*Anmeldename*“).

Der Ordner „Desktop“ findet sich in Microsoft Windows 7 bis 10 schließlich tief unten als Verzeichnis „C:\Users*Anmeldename*\Desktop“ und die unsinnigerweise[1] als „Dokumente“ bezeichneten eigenen Dateien auf derselben Ebene als „C:\Users*Anmeldename*\Documents“.

1 Bevor die Microsoft Corporation das Wort „Dokument“ durch Anwendung auf alle möglichen Arten beliebiger Informationsgruppen verwässerte, bedeutete es in Deutschland „Urkunde“ oder „beweiskräftiges Schriftstück“.

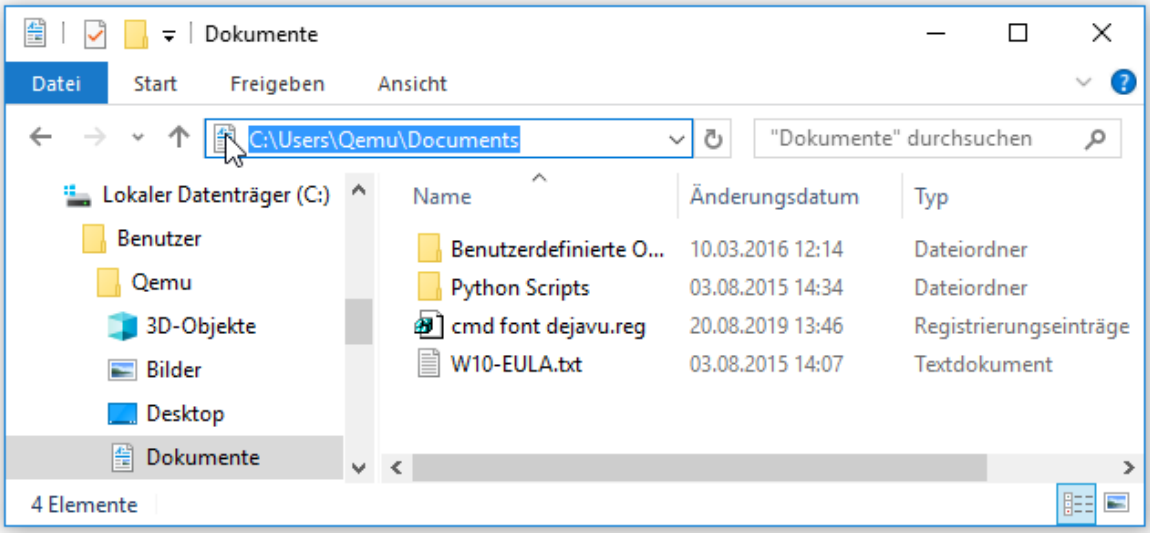


Abb. 9: Eigene Dateien unter Windows 10

Das Verzeichnis „C:\Users“ von Microsoft Windows entspricht somit dem Verzeichnis „/home“ von Linux und Unix oder dem Verzeichnis „/Users“ von macOS.

Der Windows-Explorer zeigt die wahren Verzeichnisnamen seiner Ordner an, wenn man in der ganz oben im Fenster angezeigten Adressleiste auf das Icon links des Ordnernamens klickt. Aus dem Ordnernamen „«Benutzer▸*Anmeldename*▸Eigene Dokumente“ wird dann der Verzeichnisname „C:\Users*Anmeldename*\Documents“.

Wir benötigen diese wahren Datei- und Verzeichnisnamen, wenn wir Programme entwerfen, die selber Dateien anlegen, lesen, verarbeiten und schreiben werden. Bis dahin genügen uns der Desktop oder der Ordner für „Dokumente“.

2.3.4 Archivdateien

Um Gruppen von Dateien einfacher weitergeben zu können und um Platz zu sparen, fassen wir mehrere Dateien und sogar ganze Verzeichnisbäume zu einer komprimierten Archivdatei zusammen.

Unter Microsoft Windows heißen diese Archivdateien seit Windows XP „ZIP-komprimierte Ordner“ und sind auf den ersten Blick von gewöhnlichen Verzeichnissen nur schwer zu unterscheiden. Die Funktion zum An-

legen eines ZIP-Archivs ist etwas versteckt angeordnet und befindet sich beim Windows-Explorer im Kontextmenü der rechten Maustaste unter dem Menüpunkt „Senden an …“.

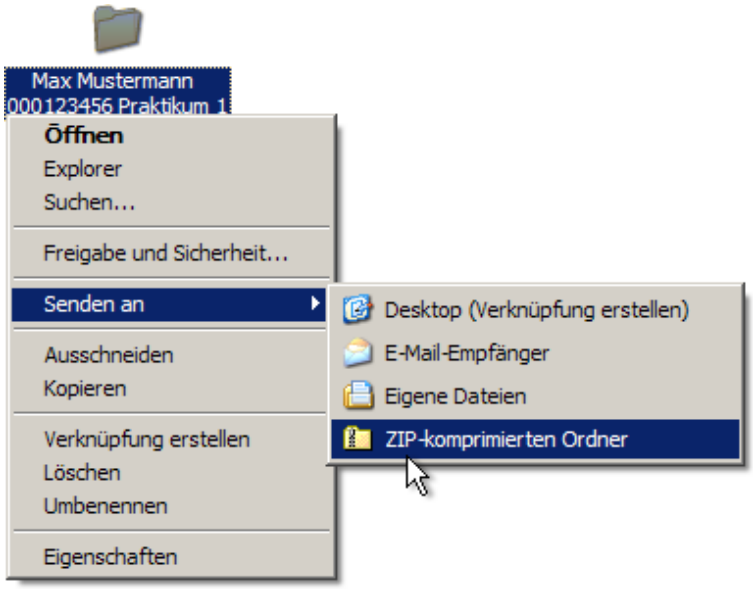


Abb. 10: Anlegen eines ZIP-Archives im Windows-Explorer

Außer dem ZIP-Format von Windows gibt es noch andere Archivformate wie RAR, TAR.BZ2 oder 7Z, die in der Regel effizienter komprimieren und vielseitiger in der Bedienung sind.

Gerade das freie[1] 7Z-Format besitzt gegenüber ZIP Vorteile, da ZIP einen schweren Designfehler aufweist, der dafür verantwortlich ist, dass es Dateinamen mit Umlauten oder Sonderzeichen nicht eindeutig speichert. In lokal begrenzten Betriebssystemmonokulturen fällt der Fehler nicht auf, doch beim Austausch zwischen unterschiedlichen Sprachversionen von Windows oder zwischen Windows und anderen Betriebssystemen, wie Linux oder macOS, kommt es bei ZIP-Dateien immer wieder zu Problemen mit Dateinamen. Falls Sie ZIP-Dateien verwenden, sollten Sie für die Dateinamen nur die Buchstaben „A“ bis „Z“ und „a“ bis „z“ sowie Ziffern, Punkte und Unterstriche „_“ verwenden. Die Namen der verschiedenen Dateien in der ZIP-Datei dürfen sich nicht nur durch Groß- und Klein-

1 Als „frei“ bezeichnen wir Software, wenn ihre Verwendung von Patenten und sonstigen Verwendungsbeschränkungen unbelastet ist. Freie Software ist nicht notwendigerweise kostenlos. Oft wird das Wort „frei“ daher durch Erläuterungen ergänzt: „frei wie freie Rede“ im Gegensatz zu „frei wie Freibier“.
Das Programm 7zip erhalten Sie auf https://www.7-zip.org/download.html

schreibung unterscheiden, da Microsoft Windows sonst Probleme beim Entpacken dieser Dateien verursacht.

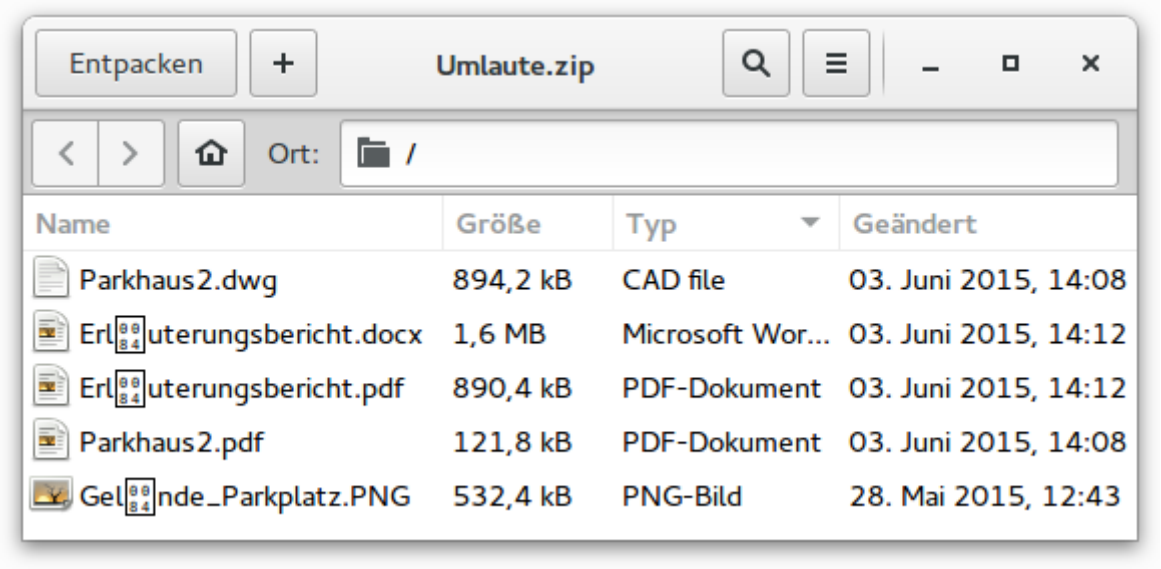


Abb. 11: ZIP-Datei mit Windows-Umlauten unter Linux

Dass Windows ZIP-Archive scheinbar als gewöhnliche Verzeichnisse behandelt, führt in der Praxis häufig zu Problemen, denn die „transparente“ Unterstützung von ZIP-Archiven durch Windows birgt eine Fußangel. Der Windows-Explorer tut zwar so, als sei eine ZIP-Datei ein ganz normales Verzeichnis, tatsächlich wird jedoch jede im Archiv enthaltene Datei, wenn sie doppelgeklickt wird, einzeln in das Windows-Temporärverzeichnis „%tmp%“ entpackt und dort mit dem ihrem jeweiligen Dateityp zugeordneten Programm geöffnet. Die anderen Dateien des Archivs, die von der doppelgeklickten Datei möglicherweise benötigt werden, werden jedoch dort im Temporärverzeichnis nicht gefunden, weil sie sich noch unausgepackt in der Archivdatei befinden.

Viele Programme weigern sich außerdem, geöffnete Dateien aus dem Temporärverzeichnis nach Änderungen wieder dort zu speichern, weil Windows die Änderungen dann nicht wieder in die Datei im ZIP-Archiv überträgt. Wenn Sie bereits Änderungen an einer Datei durchgeführt haben, können Sie diese unter einem anderen Namen in einem regulären Verzeichnis speichern, damit die Änderungen nicht verlorengehen.

Heruntergeladene ZIP-Archive sollten Sie daher immer komplett entpacken, bevor Sie Dateien daraus bearbeiten.

2.4 Zwischenablage

Die Zwischenablage (ZA) ist unter vielen Betriebssystemen ein Konzept, mit dem innerhalb eines Programms oder programmübergreifend Daten ausgetauscht werden können.

Meistens ist die Zwischenablage über das Menü „Bearbeiten“ eines Anwendungsprogramms erreichbar, fast immer über das Kontextmenü nach Drücken der rechten Maustaste und am schnellsten über drei unter Windows und Linux einheitliche Steuerungstastenkombinationen:

Strg C	Kopieren (Markierte Inhalte werden in die ZA kopiert)
Strg X	Ausschneiden (Inhalte werden in die ZA verschoben)
Strg V	Einfügen (Von der Zwischenablage an ein neues Ziel)

Gelegentlich finden wir in Programmen zudem die Funktion „Einfügen als …“, die es uns erlaubt, das Format oder die Art der einzufügenden Daten auszuwählen (zum Beispiel als „normaler Text“ oder „formatierter Text“). Sie ist in der Regel über die Tastenkombination Strg ⇧ V erreichbar.

Auf manchen Tastaturen ist die Steuerungstaste nicht mit Strg (Steuerung[1]), sondern mit Ctrl (Control) beschriftet. Auf Apple-Rechnern werden die Tastaturkommandos für die Zwischenablage nicht mit der Steuerungstaste, sondern mit der Taste ⌘ (Command) ausgelöst.

Im Englischen heißt die Zwischenablage *Clipboard* (Klemmbrett), weshalb die Einfügen-Funktion üblicherweise über ein wie ein Klemmbrett aussehendes Icon erreichbar ist.

1 Falls Ihr Informatiklehrer Ihnen beigebracht hat, diese Taste hieße „String“, „Strange“ oder „Strong“, geben Sie ihm bitte meine Telefonnummer. Ich muss mit ihm reden.

Abb. 12: Icons für „Ausschneiden“, „Kopieren“ und „Einfügen“

In Abbildung 12 befindet sich rechts neben dem Klemmbrett-Icon ein kleines Dreieck, hinter dem sich ein nach unten aufklappendes sogenanntes Drop-Down-Menü verbirgt, das die zur Verfügung stehenden Einfügeformate des Zwischenablageinhalts aufzählt.

2.5 Bildschirmkopien

Zu Dokumentationszwecken ist es oft sinnvoll, Grafiken mit dem aktuellen Bildschirminhalt oder Teilen davon als sogenannte „Screenshots“ in einen Text einzufügen. Smartphonefotos sind als schnelle Bildnotizen sehr beliebt, eignen sich für technische Dokumentationen jedoch fast nie, da sie in der Regel verwackelt, verdreht, verzerrt, unscharf, farbstichig, falsch belichtet und mit einem als Moiré bekannten Streifenmuster überzogen sind. Besser ist es, den Bildschirminhalt unmittelbar zu verwenden.

Als sehr praktisch erweist sich dazu bei PCs die Taste [Druck]. Ende des vergangenen Jahrhunderts sorgte das Drücken dieser Taste noch dafür, dass die gerade angezeigten 80 × 25 Zeichen des Textbildschirms auf dem angeschlossenen Drucker ratternd zu Papier gebracht wurden. Vor über 20 Jahren wurde die Funktion dieser Taste jedoch geändert. Sie kopiert nun unter den meisten Windowsversionen den gesamten Bildschirminhalt in die Zwischenablage und schreibt ihn unter Linux in eine Bilddatei im Bilderordner des angemeldeten Benutzers bzw. der angemeldeten Benutzerin. Die Tastenkombination [Alt Druck] kopiert nur den Inhalt des aktuellen Fensters (einschließlich der Fensterdekorationen wie Rahmen und Titelzeile). Die meisten Bildschirmkopien in diesem Buch sind so entstanden.

Auf Apple-Tastaturen gibt es keine dedizierte Druck-Taste; hier helfen Tastenkombinationen aus [⌘] (Command), [⇧] (Shift oder Umschalttaste) sowie der Ziffern [3] und [4] weiter.

Auch Mobiltelefone verfügen in der Regel über eine Screenshotfunktion. Unter dem Linux-Betriebssystem Android wird diese beispielsweise durch exakt gleichzeitiges Drücken der Tasten [an/aus] und [leiser] ausgelöst und auf iPhones durch exakt gleichzeitiges Drücken der Tasten [an/aus] und [home].

Unter Linux und macOS können wir auch beliebige Ausschnitte von Bildschirminhalten anfertigen. Die folgende Tabelle zeigt die dazu erforderlichen Tastenkombinationen:

Funktion	Linux	macOS	Windows
Den gesamten Bildschirm in die Zwischenablage kopieren	Strg Druck	Ctrl ⌘ ⇧ 3	Druck
Den gesamten Bildschirm als Grafikdatei speichern	Druck	⌘ ⇧ 3	⊞ Druck[1]
Das aktuelle Fenster in die Zwischenablage kopieren	Strg Alt Druck	Ctrl ⌘ ⇧ 4 Leertaste	Alt Druck
Das aktuelle Fenster als Grafikdatei speichern	Alt Druck	⌘ ⇧ 4 Leertaste	
Einen rechteckigen Bereich in die Zwischenablage kopieren	Strg ⇧ Druck	Ctrl ⌘ ⇧ 4	⊞ ⇧ S[2]
Einen rechteckigen Bereich als Grafikdatei speichern	⇧ Druck	⌘ ⇧ 4	
Ein kurzes[3] Video des laufenden Desktops als Datei aufzeichnen/stoppen	Ctrl Alt ⇧ R		

Tabelle 1: Tastenkombinationen für Bildschirmkopien

Mit Ubuntu Linux 22.04 wurden die zahlreichen Tastenkombinationen durch ein grafisches Menü abgelöst, das alle Optionen vereinigt. Bildschirmkopien werden nun automatisch sowohl in die Zwischenablage kopiert als auch im Verzeichnis „~/Bilder/Bildschirmfotos“ abgelegt. Videos werden entsprechend im Verzeichnis „~/Videos/Bildschirmaufzeichnungen“ gesammelt und dürfen nun auch mehrere Minuten lang sein.

1 Erst ab Windows 10

2 Erst ab Windows 10 „Creators Update“, 2017

3 Standardmäßig wird die Aufzeichnung nach 30 Sekunden beendet.

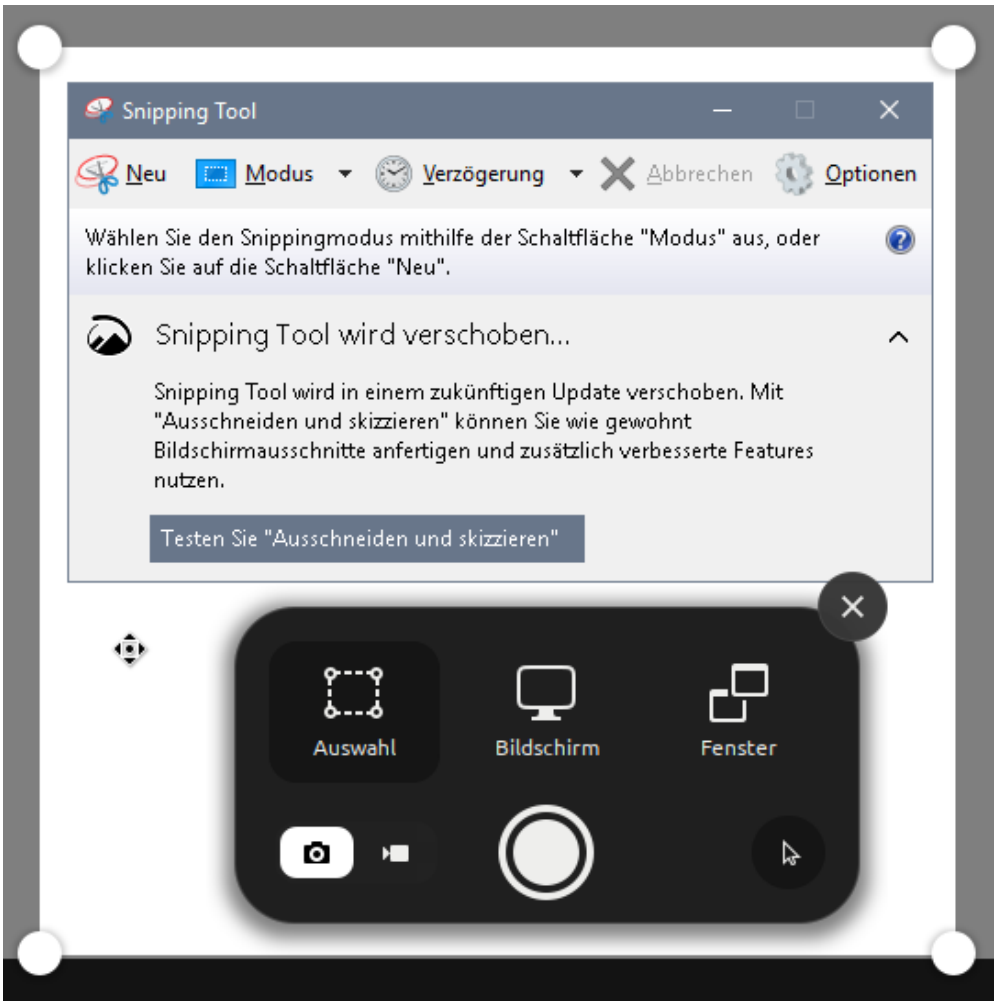


Abb. 13: Bildschirmkopiemenü in Windows (oben) und Linux (unten)

Um auch unter Microsoft Windows vernünftig Bildschirmkopien anfertigen zu können, benötigen wir je nach Windowsversion noch ein separates Programm. Im Lieferumfang von Windows befindet sich dazu seit Version 7 das sogenannte „Snipping Tool“, das die aus anderen Betriebssystemen bekannten Möglichkeiten nachliefert. In Windows 10 wurde zumindest die Tastenkombination ⊞ Druck eingebaut, um Bildschirmkopien als Grafikdatei im Ordner „Eigene Bilder/Screenshots“ zu speichern, doch erst mit dem sogenannten „Creators Update“ wurde 2017 auch für Windows die Möglichkeit geschaffen, beliebige Bildschirmausschnitte durch Drücken der Tastenkombination ⊞ ⇧ S zu kopieren. Tatsächlich macht ⊞ ⇧ S nichts anderes, als das Snipping Tool mit dem Parameter „/clip“ zu starten.

Seit 2019 behauptet eine flächenfüllende Einblendung im Snipping-Tool von Windows 10, dass dieses „in einem zukünftigen Update verschoben“ und durch ein ähnliches Programm namens „Ausschneiden und Skizzieren[1]“ abgelöst werde. In Windows 11 verschwand diese Einblendung jedoch wieder und das Snipping Tool wurde durch eine App ersetzt, die ebenfalls wieder Snipping Tool heißt und nun immerhin mit der Druck-Taste gestartet werden kann.

Noch mehr Möglichkeiten bieten Grafikprogramme wie das kostenlose GIMP[2], das auch zeitversetzte Bildschirmkopien anfertigen kann und sogar in der Lage ist, auf Wunsch den Mauszeiger mit abzubilden[3].

1 Microsofts für ihre umständliche Ausdrucksweise und hirnlosen Wortneuschöpfungen („Posteingangs-Druckertreiber“) berüchtigte Übersetzer hatten anscheinend keine bessere Idee für die Übertragung des im Original recht knackigen Namens „Snip & Sketch“ ins Deutsche als ausgerechnet „Ausschneiden und Skizzieren“.

2 https://docs.gimp.org/de/

3 Abb. 13, die das Ubuntu-Bildschirmkopiemenü zeigt, in dem gerade ein Bildschirmausschnitt des Windows-Snipping Tools in einer virtuellen Windows-Maschine angefertigt wird, entstand auch mithilfe von GIMP.

2.6 Sonderzeichen

Buchstaben, Symbole und andere Zeichen, die nicht über einzelne Tastendrücke eingegeben werden können, nennen wird Sonderzeichen. Um diese Zeichen zu verwenden, können wir sie aus einer Codetabelle heraussuchen.

Die meisten modernen Programme unterstützen die international genormte Zeichenzusammenstellung Unicode. Jedes weltweit verwendete Schriftzeichen und tausende gängige Symbole einschließlich der unentrinnbaren Emojis sind dort mit einer festgelegten Kodierung zu finden. So lauten die Zeichencodes für die beiden griechischen Buchstaben α und β beispielsweise U+03b1 und U+03b2.

Um Sonderzeichen in Python zu verwenden, können wir entweder diesen Zeichencode verwenden, wir schreiben „α und β“ dann wie in **`print("\u03b1 und \u03b2")`**, oder wir suchen das jeweilige Zeichen aus der Zeichentabelle des Betriebssystems heraus und fügen es über die Zwischenablage in den Quelltext ein.

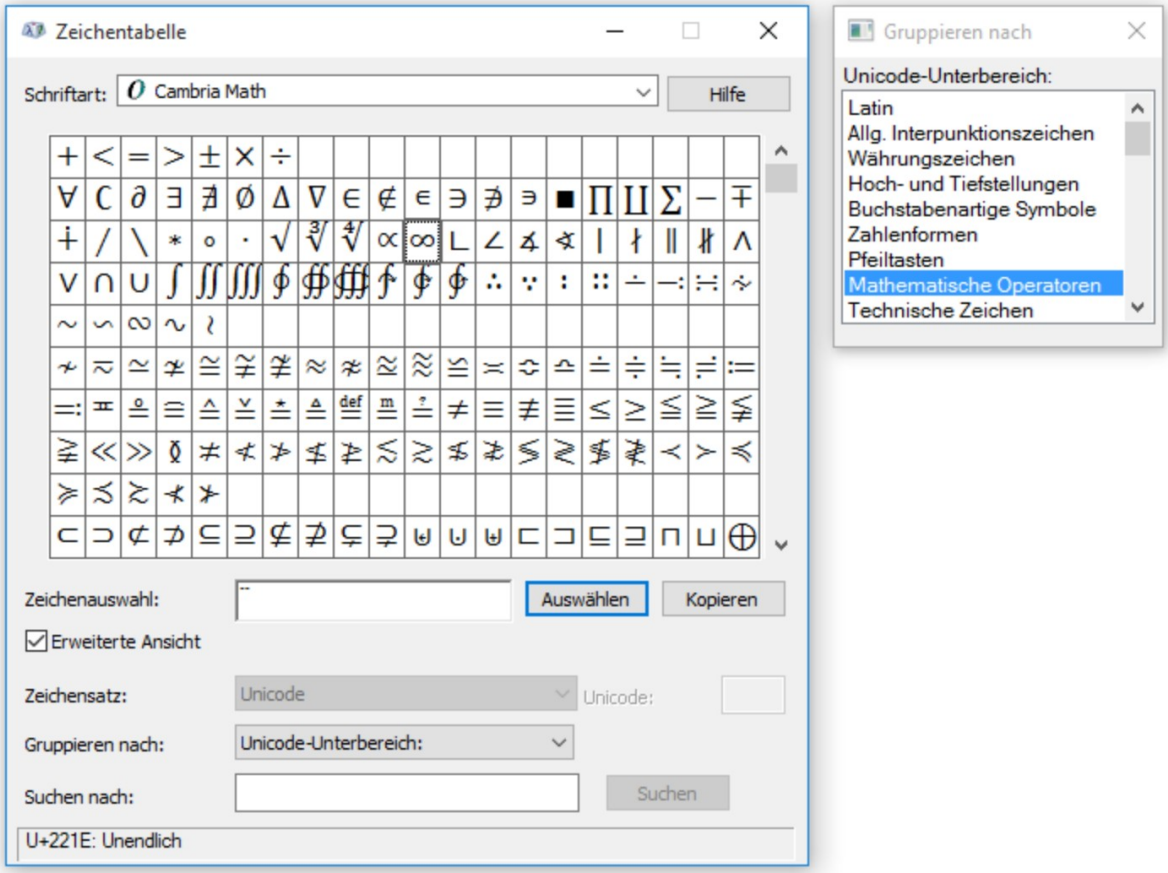


Abb. 14: Die Zeichentabelle von Windows 10

2.7 Texteditoren

Wir benötigen zum Editieren, also zum Verfassen und Bearbeiten von Quelltexten, ein geeignetes Programm. Python bietet mit der integrierten Entwicklungsumgebung „IDLE“ bereits einen einfachen Texteditor an. Dieser ist für kleinere Programme recht praktisch – nicht zuletzt, weil durch simples Drücken der Taste F5 sofort das gerade getippte Programm ausgeführt werden kann. Weitere Software benötigen Sie eigentlich nicht[1]. Es schadet jedoch auch nicht, zum Schreiben ein komfortableres Programm zu verwenden.

Ein Textverarbeitungsprogramm wie Microsoft Office Word oder LibreOffice Writer[2] ist dazu ungeeignet, da beim Speichern in den Standarddateiformaten dieser Programme zusätzliche Metadaten – das sind beispielsweise Gestaltungs- und Verwaltungsinformationen – in die Dateien geschrieben werden, wodurch die eigentlichen Inhalte für unsere Zwecke unbrauchbar werden.

Besser ist es, spezielle Texteditoren „für Programmierer“ zu verwenden. Diese besitzen oft eine besondere Hilfe für Autorinnen und Autoren, indem sie die verschiedenen Elemente eines Quelltextes auf dem Monitor farblich hervorheben. Diese sogenannte Syntaxhervorhebung (engl.: syntax highlighting) hilft uns bei der Fehlersuche, da falsch geschriebene Schlüsselwörter oder fehlende Klammern und Anführungszeichen gleich zu einer farblichen Auffälligkeit der Fehlerstelle führen.

---

1 Es sei denn, Sie schaffen es nicht mehr, Texte zu schreiben, ohne dabei Emojis zu verwenden. 😉
IDLE kann (zumindest bis Python 3.11) keine Emojis anzeigen.

2 Der kostenlose und quellenoffene LibreOffice Writer ist auch das Programm, mit dem ich gerade dieses Pythonbuch für Sie schreibe: http://de.libreoffice.org.

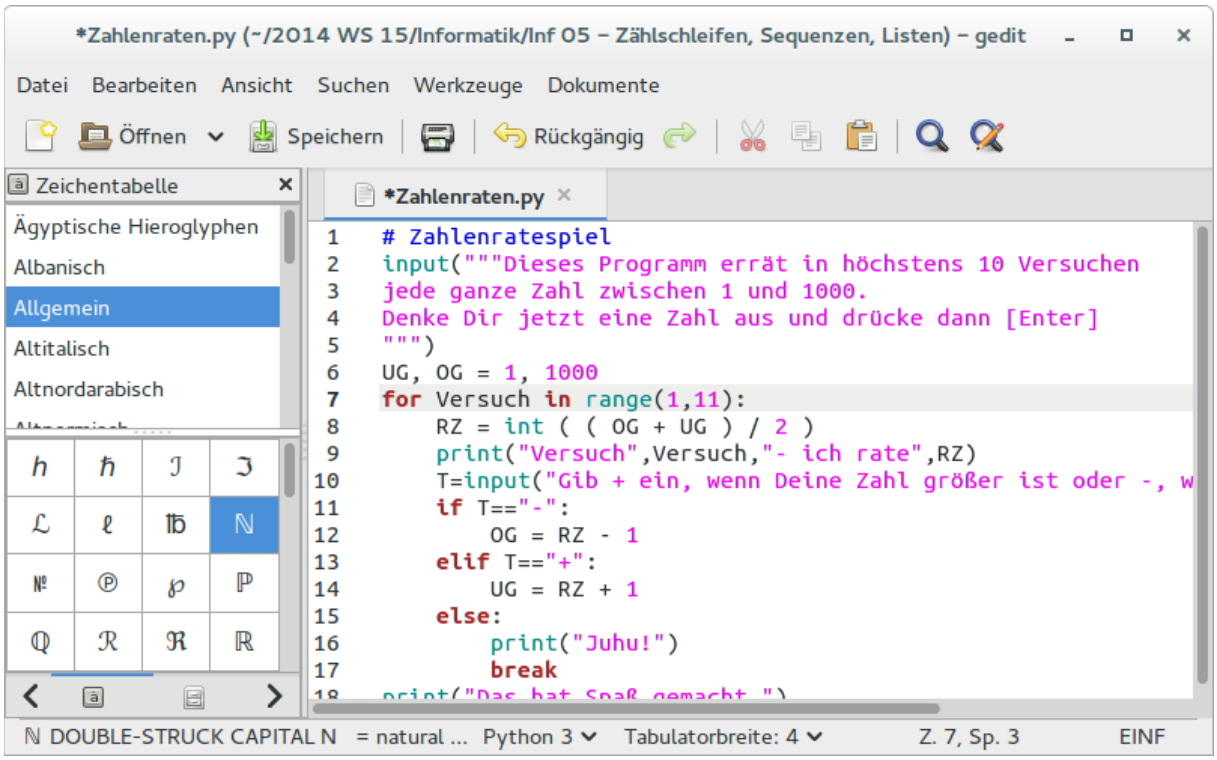


Abb. 15: Gedit unter Ubuntu Linux

Damit das Syntax-Highlighting funktioniert, muss der Editor „wissen“, welche Art von Text wir schreiben, denn ein Python-Programm wird logischerweise ganz anders farbig markiert als ein HTML-Quelltext. In der Regel trifft die Software die Entscheidung anhand der Dateinamenerweiterung. Das bedeutet aber auch: solange ein neuer Text nicht mit der korrekten Dateinamenerweiterung gespeichert wurde, ist das Syntax-Highlighting häufig inaktiv.

Hilfreich ist es auch, wenn der Editor die Nummer der aktuellen Textzeile anzeigt, damit wir diese bei Fehlern im Quelltext schnell auffinden können.

Linux-Distributionen wie Ubuntu Linux enthalten in der Regel eine Vielzahl geeigneter Editoren. Populäre und einfach zu bedienende Editoren mit grafischer Benutzungsoberfläche sind dort beispielsweise „Gedit“ oder „Geany“.

Unter Microsoft Windows gibt es in der Standardausstattung des Betriebssystems keine geeignete Software. Sie können jedoch mehrere kostenlose Windowsprogramme im Internet finden.

Welchen Texteditor Sie zum Verfassen und Bearbeiten von Quelltexten verwenden, ist weitgehend Geschmackssache. Probieren Sie einfach mal ein paar aus!

Gute Erfahrungen haben wir mit den Programmen „PSPad“ des tschechischen Programmierers Jan Fiala[1] und „Notepad++“ von Don Ho aus Frankreich[2] gemacht. Auch der unter Linux sehr beliebte Editor „Gedit“ ist in einer Windowsversion zu haben[3].

Abb. 16: Zwei vermeidenswerte Programme

Den mit Microsoft Windows mitgelieferten primitiven Texteditor „Notepad.exe“ alias „Editor“ sollten Sie möglichst nicht verwenden. Manche Textdateien entstellt dieses Softwareprodukt schon beim Öffnen. Es erkennt Zeilenumbrüche gelegentlich nicht oder zeigt die falschen Nicht-ASCII-Zeichen an. Der Editor verfügt weder über eine Randleiste mit Zeilennummern noch über Syntaxhervorhebung. Beim Speichern ruiniert er zudem zuverlässig die vorhandene Zeichenkodierung oder fügt störende Signaturbytes am Dateianfang ein.

1 Jan Fiala, PSPad, http://www.pspad.com/de/download.php
2 Don Ho, Notepad++, http://notepad-plus-plus.org
3 GNOME text editor, Gedit, https://wiki.gnome.org/Apps/Gedit

2.8 Textverarbeitungen

Textverarbeitungen und Textsatzsysteme erlauben es, längere strukturierte Texte layoutunabhängig zu erfassen und inhaltsunabhängig zu gestalten. Sie erstellen automatisch Inhalts-, Abbildungs- und Literaturverzeichnisse, passen Texte und Abbildungen in den vorgesehenen Bereich ein und führen Querverweise innerhalb eines Textes automatisch nach.

2.8.1 Formatvorlagen

Die Zuordnung von Inhalt und Form geschieht mithilfe sogenannter Formatvorlagen (in manchen Programmen heißen sie auch *Styles* oder *Templates*). So kann die Formatvorlage für eine Überschrift 2. Grades, wie sie beispielsweise über diesem Absatz steht, die Information enthalten, dass der Überschrift eine Kapitelnummer voranstehen soll, dass sie in 16 Punkt hoher Schrift in der Schriftart „DejaVu Sans“ gesetzt und fett gedruckt werden soll. Falls nun alle Überschriften 2. Grades ein anderes Aussehen erhalten sollen, so genügt es, diese Formatvorlage zu ändern und sofort werden sämtliche mit ihr formatierten Überschriften angepasst.

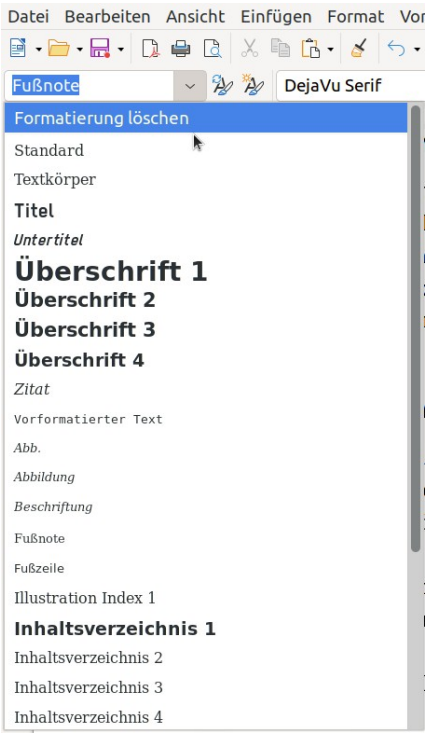


Abb. 17: Formatvorlagen in LibreOffice

Um einem Textabschnitt eine Formatvorlage zuzuweisen, markieren Sie ihn mit der Maus und wählen die gewünschte Vorlage aus der Formatierungsleiste der Textverarbeitung.

2.8.2 Schriftarten

Als Schriftart bezeichnet man die Gestaltung der Buchstaben einer Schrift. Eine Textverarbeitung kann beliebig viele Schriftarten verwenden. Allerdings: die Schnittmenge von Schriftarten, die auf allen Rechnern unter allen Betriebssystemen verfügbar sind, ist leer. Unter anderem deshalb sehen Textverarbeitungsdateien auf unterschiedlichen Rechnern, unter unterschiedlichen Betriebssystemen oder in unterschiedlichen Textverarbeitungen oft ganz anders aus als beim Schreiben. Ein anderer Grund für Abweichungen im Aussehen liegt darin, dass manche Textverarbeitungsdateiformate nicht vollständig dokumentiert sind und daher nur von den Softwareprodukten des jeweiligen Herstellers korrekt gelesen werden. Selbst bei offenen, gut dokumentierten Dateiformaten gibt es Probleme, wenn diese Formate nicht vollständig in die herstellereigenen Strukturen übersetzt werden.

Wenn Sie zu mehreren Personen an einer Datei arbeiten, sollten alle dieselbe Softwareversion unter derselben Betriebssystemfamilie verwenden und alle verwendeten Schriftarten besitzen oder sie in das Dokument einbinden. In LibreOffice geschieht das mit dem Menübefehl „Datei → Eigenschaften → Schriftart → Schriftarten ins Dokument einbetten“. Nicht alle Schriftarten erlauben das. Manche unterliegen sehr restriktiven Nutzungslizenzen.

2.8.3 Zeichenformatierung

Sie können Textstücke direkt formatieren anstatt Formatvorlagen zu verwenden, das sollte aber möglichst die Ausnahme bleiben, weil die Bearbeitbarkeit vor allem längerer Texte dadurch erheblich erschwert wird.

Hervorhebungen von Textstellen werden auch *Auszeichnungen* genannt. Sie sollten äußerst sparsam verwendet werden, weil sie den Lesefluss u n t e r b r e c h e n und dadurch die ***Lesbarkeit*** eines Textes vermindern.

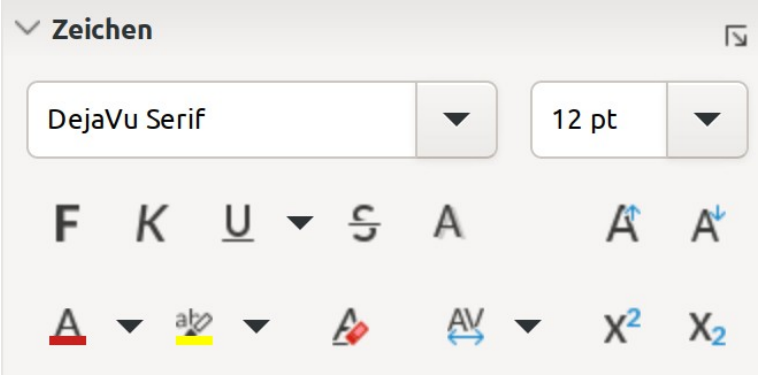


Abb. 18: Zeichenformatierung in LibreOffice

Für **Fettschrift** wird Text mit dem Format „fett“ versehen. Der Button dafür ist in eingedeutschten Textverarbeitungen meistens ein fettgeschriebenes „F“, gelegentlich aber auch ein „B“ (für „bold“) oder sogar nur ein „A“.

Kursivschrift soll ein wenig an geschwungene, leicht schräg gestellte Handschrift erinnern. Kursiv formatierte Buchstaben sind daher nicht einfach parallelogrammartig geschert, sondern in der Regel komplett anders gestaltet als die geraden Buchstaben einer Schriftart.

Die Unterstreichung ist eine Auszeichnungsform, die heute eher selten genutzt wird, da unterstrichene Textstellen zu sehr wie Hyperlinks aussehen und so für Verwirrung sorgen.

S p e r r s c h r i f t mit einzeln geschriebenen Buchstaben ist verpönt, da durch Leerzeichen auseinandergezogene Wörter nicht durch die Suchfunktion gefunden werden können und auch die automatische Silbentrennung hier nicht funktioniert.

2.8.4 PDF-Dateien

Zum Austausch von Texten, bei denen es auf das exakte Aussehen ankommt, verwenden wir das „portable document format“ PDF. Wenn wir in LibreOffice die Option „Gliederung exportieren“ verwenden, erhalten wir ein navigierbares PDF-Dokument, in dem wir alle Kapitel direkt anspringen können.

Zum erstmaligen Erzeugen einer PDF-Datei wählen Sie den Menübefehl „Datei → Exportieren als … → als PDF exportieren“. Dort nehmen Sie alle gewünschten Einstellungen vor, zum Beispiel, dass die Gliederung exportiert werden soll. Alle weiteren PDF-Exportvorgänge können Sie zukünftig aus dem Hauptbildschirm durch Anklicken des PDF-Icons auslösen.

In Microsoft Office Word ist die Einstellung etwas versteckter. Rufen Sie im Datei-Menü den Befehl „Exportieren“ auf und wählen Sie dort „PDF/XPS-Dokument erstellen“ aus. Im nächsten Dialog drücken Sie den Button „Optionen“ und aktivieren im sich öffnenden Dialog im Abschnitt „Nicht druckbare Informationen einschließen“ den Unterpunkt „Textmarken erstellen mithilfe von“. Dort wählen Sie „Word-Textmarken“ oder „Überschriften“ aus. Wenn der Unterpunkt „Textmarken erstellen mithilfe von“ ausgegraut und nicht auswählbar ist, besitzt Ihr Text keine richtigen Überschriften. Verwenden Sie für Überschriften immer die entsprechenden Formatvorlagen und keine direkten Formatierungen!

Auf den meisten Rechnern werden PDF-Dateien standardmäßig mit einem Webbrowser geöffnet. Um darin navigieren zu können, klicken Sie zum Beispiel je nach verwendeter Software entweder auf das „Lesezeichen“-Symbol oben rechts oder auf ein „Hamburger[1]-Icon“ oben links. Wenn Ihr Browser das nicht unterstützt, ist ein zusätzlicher PDF-Betrachter, wie beispielsweise Evince, sinnvoll. Software der Firma Adobe benötigen Sie entgegen häufig verbreiteter Falschinformationen nicht. Der Adobe Reader ist außerdem ziemlich ressourcenhungrig und braucht oft recht lange, um überhaupt zu starten und eine PDF-Datei anzuzeigen.

1 Das Hamburger-Icon ≡ erinnert anscheinend manche Menschen an die Rinderhackbrötchen von Schnellimbissketten.

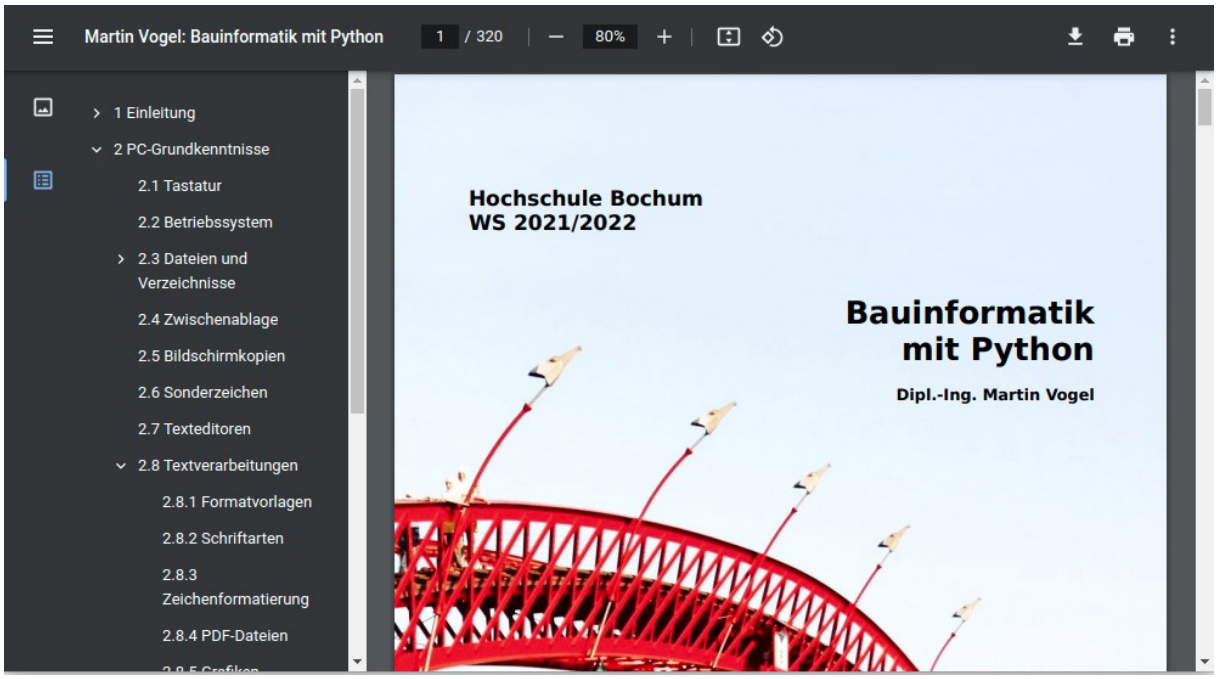


Abb. 19: Navigation durch die Dokumentenstruktur im Browser

2.8.5 Grafiken

Um eine Grafik in einen Text einzufügen, können Sie die gewünschte Grafikdatei direkt in den Text ziehen. Vorhandener Text fließt automatisch um die Grafik herum. Die Höhe und Breite der Grafik passen Sie freihändig mit der Maus an oder Sie bestimmen die Abmessungen numerisch exakt über die Eigenschafteneinstellungen, die Sie durch Rechtsklicken der Grafik aufrufen. Kontrast und Helligkeit lassen sich nachträglich einstellen. Das ist besonders für eingefügte Smartphonefotos wichtig, mit denen Sie besonders einfach auf Papier vorliegende Bildinhalte in Ihren Text übernehmen können.

Die Abstände der Grafik zum umgebenden Text können Sie nach Belieben einstellen. Um einer Grafik eine Beschriftung zuzuordnen, rechtsklicken Sie diese und wählen den Menüeintrag „Beschriftung einfügen …“. Sie können zwischen verschiedene Kategorien auswählen, für die Sie später separate Abbildungsverzeichnisse erstellen können.

2.8.6 Verzeichnisse

Sie können automatisch Inhalts- oder Abbildungsverzeichnisse erstellen lassen, die an einer beliebigen Stelle im Text eingefügt werden können. Achten Sie darauf, alle Verzeichnisse vor dem Drucken oder vor dem Exportieren des Textes als PDF-Datei aktualisieren zu lassen. In LibreOffice geschieht das über die Menüfolge „Extras → Aktualisieren → Alles“.

2.8.7 Erzwungene neue Seite

Mit Strg ↲ erzeugen Sie einen festen Seitenumbruch. Normalerweise ist das niemals notwendig. Versuchen Sie auf keinen Fall, durch wiederholtes Drücken der Eingabetaste auf die nächste Seite zu kommen. Sie würden damit zeigen, dass Sie nicht begriffen haben, was eine Textverarbeitung von einer Schreibmaschine unterscheidet.

Auch zwischen zwei Absätze setzen Sie bitte nicht einfach eine leere Zeile. Arbeiten Sie immer mit den Abstandseinstellungen des verwendeten Absatzformates.

2.8.8 Kopf- und Fußzeilen

Der Text in Kopf- und Fußzeilen erscheint auf jeder Seite Ihres Textes. Eine automatische Seitenzahl fügen Sie dort durch den Menübefehl „Einfügen → Seitennummer“ ein. Andere automatisch aktualisierbare Inhalte, wie den Namen der Textverarbeitungsdatei, das Druckdatum oder die Gesamtbearbeitungszeit erhalten Sie über „Einfügen → Feldbefehl“.

Es ist empfehlenswert, dass Sie sich für die Hausaufgaben in diesem Fach zumindest aneignen, wie man mit dem von Ihnen bevorzugten Programm Überschriften korrekt setzt, ein Inhaltsverzeichnis erzeugt, die Rechtschreibkorrektur verwendet, Grafiken einbindet und PDF-Dateien erzeugt.

Darüber hinaus ist das Thema „Textverarbeitung“ nicht Bestandteil dieses Kurses.

2.9 Tabellenkalkulationen

Tabellenkalkulationen gehören zu den ältesten Anwendungsprogrammen der PC-Geschichte. Sie trugen maßgeblich zur schnellen Verbreitung dieser Geräte in Buchhaltungen und Ingenieurbüros bei.

Abb. 20: Tabellenkalkulation 1979 (Bild: Wikipedia)

Bereits 1979 erschien die Tabellenkalkulation Visicalc für den Apple][, einem der ersten je gebauten Personal Computer. 1983 war Lotus 1-2-3 für das junge Betriebssystem DOS erhältlich, ein Jahr später brachte das frisch gegründete Startup-Unternehmen „Micro-Soft“ für den Apple Macintosh die Tabellenkalkulation Excel auf den Markt. Excel war eine dermaßen originalgetreue Kopie von Lotus 1-2-3, dass es von diesem sogar den Fehler übernahm, das Jahr 1900 als Schaltjahr auszuweisen[1]. 1985 veröffentlichte der Lüneburger Schüler Marco Börries das Programm StarWriter für den 8-Bit-Heimcomputer Schneider CPC, welches 1995 zum ersten plattformübergreifenden Bürosoftwarepaket StarOffice heranwuchs und Grundlage für das heutige LibreOffice (bzw. OpenOffice) wurde. 2006 schließlich machte Google mit der Browserapplikation „Text und Tabellen“ die Tabellenkalkulation unabhängig von einzelnen PCs und ermöglichte es, mit mehreren Personen gleichzeitig über das Internet an derselben Datei zu arbeiten.

1 Konrad Lischka, Das Excel-Phantomschaltjahr 1900, 2008, http://www.spiegel.de/netzwelt/web/technikaergernis-tabellenkalkulation-so-falsch-rechnet-excel-a-563637-4.html

Allen Tabellenkalkulationen gemeinsam ist das Rechengitter oder Arbeitsblatt, in dessen Zellen Zahlenwerte, Texte oder Formeln gespeichert und ausgeführt werden können. Traditionell werden bei den meisten Tabellenkalkulationen die Spalten dieses Arbeitsblattes mit Buchstaben und die Zeilen mit Zahlen bezeichnet. Die Zelle mit der Zellenadresse „B3“ ist also in der dritten Zeile der zweiten Tabellenspalte zu finden.

2.9.1 Formeln

Beginnt der Inhalt einer Zelle mit einem Gleichheitszeichen, so wird der Zellinhalt als Formel interpretiert. Mit Formeln lassen sich neue Zellwerte berechnen. Anstelle von Variablennamen werden in den Formeln einer Tabellenkalkulation oft nur die Zellbezeichner anderer Zellen verwendet.

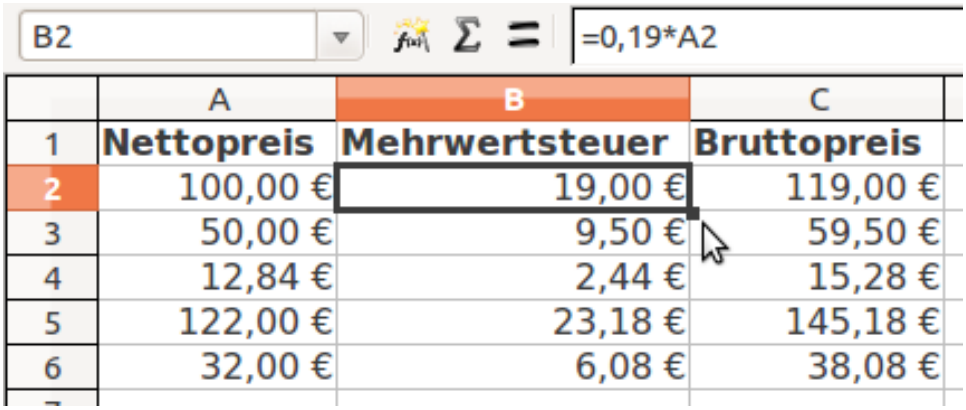


Abb. 21: Formel einer Tabellenkalkulation

In Abb. 21 lesen wir oben rechts ab, dass sich in der mit einem dicken schwarzen Rahmen hervorgehobenen Zelle B2 die Formel **`=0,19*A2`** befindet, welche also 19 % des Wertes von Zelle A2 ausrechnet.

2.9.2 Variablennamen

Um Formeln lesbarer zu gestalten, können wir anstelle von Zellbezeichnern richtige Variablennamen verwenden, die wir einzelnen Zellen oder ganzen Zellbereichen zuweisen. Den gewünschten neuen Namen tragen wir dazu nach dem Markieren der Zelle oder des Zellbereichs in das in Abb. 21 oben links zu sehende Feld ein, in dem momentan noch der Zellbezeichner „B2“ zu sehen ist.

Die Namen sollten nur aus den Buchstaben von „A“ bis „Z“, dem Unterstrich „_“ und Ziffern bestehen und dürfen keine Leerzeichen enthalten. Umlaute und sonstige Sonderzeichen sind in der Regel nicht zulässig. Groß- und Kleinschreibung wird ignoriert. Das erste Zeichen des Namens darf keine Ziffer sein.

Die Namen C und R (sowie c und r) dürfen in Microsoft Office Excel und LibreOffice Calc nicht verwendet werden.

2.9.3 Relative und absolute Zellbezüge

Beim Kopieren von Formeln stellen wir fest, dass die Zellbezeichner darin automatisch verändert werden. Mit jeder Zeile, die die Formel beim Kopieren nach unten wandert, erhöhen sich die Ziffern der Zellbezeichner um den Wert eins und mit jeder Spalte nach rechts wird der Spaltenbuchstabe heraufgesetzt. Aus der Formel **`=A2+B4`** wird, wenn sie drei Spalten nach rechts und vier Zeilen nach unten kopiert oder verschoben wird, **`=D6+E8`**.

Relativ zur neuen Position bleiben die Zellbezüge dabei unverändert. Ein Bezeichner, der an der alten Position auf ein Feld drei Spalten links und zwei Zeilen oberhalb der Formel verwies, bezieht sich auch an der neuen Position auf ein Feld drei Spalten links und zwei Zeilen oberhalb der Formel.

Damit Formeln sich beim Kopieren oder Verschieben nicht unerwünscht verändern, sollten wir bevorzugt mit Variablennamen arbeiten. Diese beziehen sich immer auf absolute Zelladressen und Zellbereiche. Wir können aber auch Zellbezeichner als absolut kennzeichnen, ohne ihnen einen Namen zu geben. Dazu stellen wir den Zeilen- und Spaltenbezeichnern jeweils ein Dollarzeichen voran. Der Zellbezeichner **`$A2`** wird auch auf der neuen Position stets auf Spalte A verweisen, beim Zellbezeichner **`A$2`** bleibt Zeile 2 fest eingestellt und **`$A$2`** bezieht sich immer absolut auf Zelle A2.

2.9.4 Funktionen

Um Formeln innerhalb von Zellen nicht übermäßig lang und kompliziert werden zu lassen, verfügen Tabellenkalkulationen über eine große Zahl eingebauter Funktionen. Leider gibt es hier keinen verbindlichen Standard, sodass in der Dokumentation der jeweils verwendeten Tabellenkalkulation nachgeschaut werden muss, ob die gesuchte Funktion dort existiert und wie sie dort heißt. Zu allem Überfluss haben dieselben Funktionen in den internationalisierten Versionen eines Programms oft unterschiedliche Namen.

Die meisten in Deutschland verwendeten Tabellenkalkulationen verwenden weitgehend dieselben Funktionsnamen, auch wenn diese zum Teil recht ausufernde Längen besitzen (Abb. 22).

Eine Funktion besteht immer aus einem Namen und einem oder mehreren dahinter in Klammern eingeschlossenen Argumenten. Der Funktionsaufruf **`SIN(B3)`** berechnet beispielsweise den Sinus des Wertes in Zelle B3 und **`WURZEL(C7)`** zieht die Quadratwurzel aus Zelle C7.

Wenn eine Funktion mit mehreren Argumenten aufgerufen wird, so sind diese in deutschsprachigen Tabellenkalkulationen mit einem Semikolon zu trennen. In englischsprachigen Tabellenkalkulationen wird anstelle des Semikolons ein Komma zur Trennung von Funktionsargumenten verwendet. Das geht in Deutschland nicht, weil wir das Komma schon als Dezimaltrennzeichen verwenden.

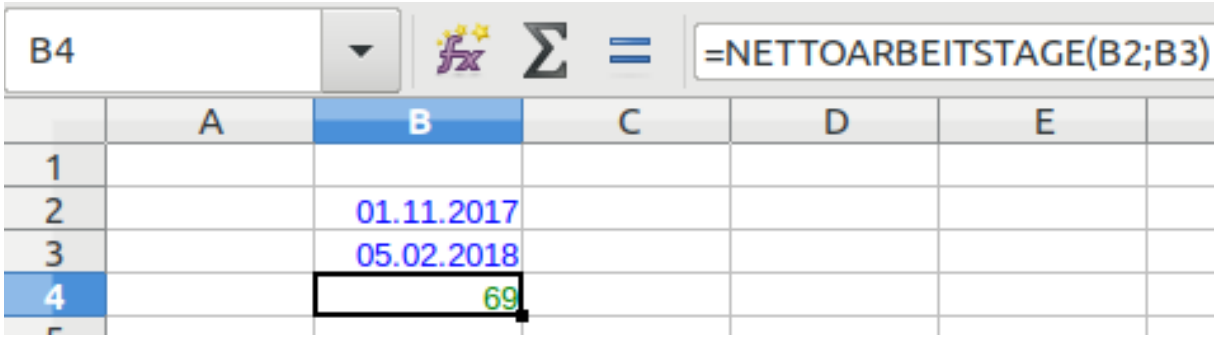


Abb. 22: Funktion mit zwei Parametern

2.9.5 Zellbereiche

Bezieht sich eine Formel nicht nur auf eine einzelne Zelle, sondern auf einen zusammenhängenden Bereich, so kann man diesen Bereich als **`von:bis`** formulieren. Soll in Feld A5 beispielsweise die Summe der Felder A1 bis A4 ausgerechnet werden, so schreiben wir dies als **`=SUMME(A1:A4)`** in A5 (Abb. 23).

	A
1	100
2	12
3	31
4	42
5	=SUMME(A1:A4)

Abb. 23: Bereichschreibweise

2.9.6 Fallunterscheidungen mit WENN

Innerhalb von Formeln können Entscheidungen auf der Grundlage logischer Aussagen getroffen werden. Der Funktion **`WENN()`** sind dazu drei Argumente zu übergeben: Erstens die zu untersuchende logische Aussage, zweitens der Wert, den die Funktion zurückgeben soll, wenn die Aussage wahr ist und drittens der Rückgabewert für den Fall, dass die Aussage nicht wahr ist. Merkhilfe: Wenn(was; dann; sonst). Falls keine Werte für „dann“ und „sonst“ angegeben werden, gibt die Wenn-Funktion den Wahrheitswert der Aussage als **`WAHR`** oder **`FALSCH`** zurück.

Als Beispiel aus dem Bauingenieurwesen seien in dem in Abb. 25 gezeigten Tabellenblatt in Spalte E, von Zelle E4 an abwärts, diverse Zugspannungswerte eingetragen. In Spalte F soll nun von uns untersucht werden, ob diese einen Grenzwert überschreitet. Dieser wurde in Zelle B1 eingetragen, der wir der Lesbarkeit halber den Namen „Grenzwert“ zugewiesen haben.

Die logische Aussage für Zelle F4 lautet also „E4<=Grenzwert“, der darzustellende Text, wenn die Aussage wahr ist, soll „Spannung zulässig“ lauten und der Text für den Fall, dass die Aussage falsch ist, lautet „Grenzwert überschritten!“

Wir können das grafisch übersichtlich darstellen:

E4 <= Grenzwert	
wahr	falsch
"Spannung zulässig"	"Grenzwert überschritten!"

Abb. 24: Fallunterscheidung

In Zelle F4 schreiben wir diese Fallunterscheidung als Formel:
`=WENN(E4 <= Grenzwert; "Spannung zulässig"; "Grenzwert überschritten!")`.

Die Anführungszeichen setzen wir um alle Texte, die wir ausgeben wollen, um sie von Variablen wie **`Grenzwert`** oder Zelladressen wie **`E4`** zu unterscheiden.

Bitte verwenden Sie nur die simplen "Ersatz-Anführungszeichen" der Schreibmaschinentastatur, nicht die ästhetisch ansprechenderen „typographischen Anführungszeichen“![1]

1 Unter Microsoft Windows stellt sich das Problem nicht, weil es dort ohnehin nur das gerade Ersatz-Anführungszeichen " auf der Tastatur gibt, wie es im 19. Jahrhundert als Notlösung anstelle richtiger Anführungszeichen für die damals aufkommenden Schreibmaschinen eingeführt wurde und wie es im 20. Jahrhundert von dort in den ASCII-Code übernommen wurde.
Unter dem fortgeschritteneren X-Window-System von Linux besitzt die Tastatur jedoch auch die zehn korrekten typographischen Anführungszeichen („ “ ” , ‘ ’ » « › ‹) auf den Belegungsebenen mit AltGr und Shift-AltGr.
Auch unter macOS können immerhin acht typographisch korrekte Anführungszeichen auf den Belegungsebenen Option [⌥] und Option-Shift [⌥⇧] direkt eingegeben werden.
Die Wikipedia-Seite https://de.wikipedia.org/wiki/Anführungszeichen führt alle Tastenkombinationen zur Eingabe der Zeichen unter den drei verbreiteten PC-Betriebssystemen auf.

F4 =WENN(E4<=Grenzwert ; "Spannung zulässig" ; "Grenzwert überschritten!")

	A	B	C	D	E	F
1	Zulässige Spannung:	10	N/mm²			
2	**Zugspannungen in Rechteckquerschnitten**					
3	**Höhe [mm]**	**Breite [mm]**	**Fläche [mm²]**	**Zugkraft [N]**	**Spannung [N/mm²]**	**Bewertung**
4	7	22	154	1624	10,5	Grenzwert überschritten!
5	10	18	180	1686	9,4	Spannung zulässig
6	3	20	60	2276	37,9	Grenzwert überschritten!
7	30	11	330	2283	6,9	Spannung zulässig
8	18	17	306	1862	6,1	Spannung zulässig
9	30	18	540	2264	4,2	Spannung zulässig
10	23	28	644	983	1,5	Spannung zulässig
11	30	21	630	187	0,3	Spannung zulässig
12	11	8	88	1017	11,6	Grenzwert überschritten!
13	10	30	300	972	3,2	Spannung zulässig
14	11	6	66	2611	39,6	Grenzwert überschritten!
15	26	15	390	2503	6,4	Spannung zulässig
16	15	7	105	1660	15,8	Grenzwert überschritten!
17	19	25	475	1694	3,6	Spannung zulässig
18	15	28	420	2493	5,9	Spannung zulässig
19	25	24	600	477	0,8	Spannung zulässig
20	15	24	360	1361	3,8	Spannung zulässig

Abb. 25: Tabelle mit Fallunterscheidungen

Falls wir zwischen mehr als zwei Fällen unterscheiden möchten, können wir die Wenn-Funktion verschachteln. Anstelle eines direkten Wertes für „dann“ oder „sonst“ setzen wir einfach eine komplette weitere Wenn-Funktion ein.

Eine Stahlpreisliste enthält folgende Formulierung: *„Bei mehr als 16,1 Metern Länge ist (...) ein Aufpreis von 10 Euro, bei mehr als 18,1 Metern 20 Euro und bei mehr als 22,1 Metern 30 Euro pro Tonne zu berücksichtigen.“* – wir müssen also vier verschiedene Fälle unterscheiden. Dazu formulieren wir drei logische Aussagen: „L > 22,1“, „L > 18,1“ und „L > 16,1“. Der Übersicht halber stellen wir sie grafisch dar und schreiben unter jede Aussage, welche Antwort wir erwarten oder welche Aussage wir als nächstes prüfen müssen, wenn die jeweilige Aussage wahr oder falsch ist.

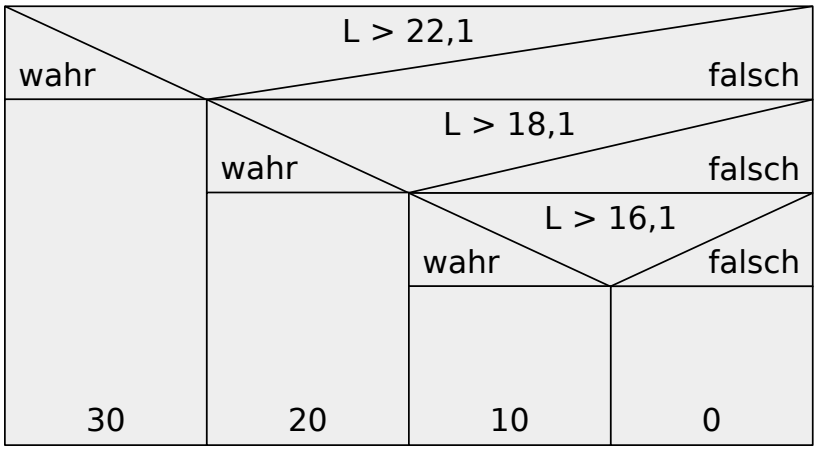


Abb. 26: Drei verschachtelte Fallunterscheidungen

In der Formelschreibweise einer Tabellenkalkulation stellen wir diesen Entscheidungsbaum so dar:

```
=WENN(L > 22,1; 30; WENN(L > 18,1; 20; WENN(L > 16,1; 10; 0)))
```

Dabei müssen wir darauf achten, dass die unvollständige sprachliche Konstruktion dieser Preisstaffelung so umgesetzt wird, dass nicht die Erfüllung der ersten Bedingung („bei mehr als 16,1 Metern Länge“) schon alle anderen Fälle einschließt.

2.9.7 VERWEIS, SVERWEIS und WVERWEIS

Weil Fallunterscheidungen mit mehreren Alternativen recht unübersichtlich werden, empfiehlt es sich, stattdessen eine kleine Tabelle anzulegen und zugehörige Werte daraus ablesen zu lassen. Deutschsprachige Tabellenkalkulationen kennen dazu die Funktion **`VERWEIS`** sowie ihre nahen Verwandten **`SVERWEIS`** und **`WVERWEIS`**.

Die Funktion **`VERWEIS(w; s; z)`** sucht einen Wert **`w`** in einem sortierten(!) Suchbereich **`s`** und gibt den korrespondierenden Wert aus dem gleich großen Zielbereich **`z`** zurück. Wenn **`w`** nicht in **`s`** enthalten ist, wird der größte Wert aus **`s`** verwendet, der kleiner als **`w`** ist.

In Abb. 27 sehen wir einen Ausschnitt aus einer Stahlbau-Profiltabelle. Der IPE-Bezeichnung in Spalte K ist dabei die Trägermasse in kg pro Meter zugeordnet. Um nun beispielsweise den Massenwert eines IPE-140-Trägers zu finden, verwenden wir die Formel **=VERWEIS(140; K4:K10; L4:L10)** und erhalten den Wert 13,2.

	K	L
3	IPE	kg/m
4	80	6,2
5	100	8,3
6	120	10,7
7	140	13,2
8	160	16,2
9	180	19,3
10	200	23,0

Abb. 27: VERWEIS

Als kleine Vereinfachung dürfen Such- und Zielbereich zusammengefasst werden, wenn sie unmittelbar benachbart sind. Die Formel **=VERWEIS(140; K4:L10)** gibt also ebenfalls den Wert 13,2 zurück.

Etwas umständlicher zu verwenden sind die Funktionen **SVERWEIS** und **WVERWEIS**, dafür kommen sie auch mit unsortierten Werten im Suchbereich zurecht.

Die Funktion **SVERWEIS(w; m; s; x)** sucht einen Wert **w** (oder den nächstkleineren) in der ersten Spalte der sortierten Matrix[1] **m** und gibt aus dieser Zeile den Wert in Spalte Nummer **s** der Matrix zurück. Die ganz linke Spalte hat dabei die Nummer 1. Hat **x** den Wert **0** (oder **FALSCH**), so zählen nur exakte Treffer. Die Matrix darf dann unsortiert sein.

Das Beispiel in Abb. 27 berechnet in Spalte **E** die Aufpreise für Spalte **D** gemäß der Untertabelle im Bereich **A2:B5**. Die Formel ist deutlich kürzer als die Fallunterscheidung aus Kapitel 2.9.6!

1 Als „Matrix“ bezeichnen wir hier einen beliebigen rechteckigen Zellbereich, der mindestens zwei mal zwei benachbarte Zellen umfasst.

E3 | fx Σ = | =SVERWEIS(D3; $A$2:$B$5; 2)

	A	B	C	D	E
1	Länge ≥ ...	Aufpreis		Länge	Aufpreis
2	0	0,00 €		15,8	0,00 €
3	16,1	10,00 €		17,5	10,00 €
4	18,1	20,00 €		19,5	20,00 €
5	22,1	30,00 €		15,7	0,00 €
6				18,6	20,00 €
7				23,0	30,00 €
8				17,4	10,00 €
9				14,8	0,00 €

Abb. 28: SVERWEIS

Das „S“ in **SVERWEIS** steht übrigens nicht für „Spalte“, sondern für „senkrecht“. Entsprechend gibt es daher eine Funktion **WVERWEIS(w; m; z; x)**, diese sucht „waagerecht“ einen Wert **w** (oder den nächstkleineren) in der ersten Zeile der Matrix **m** und gibt den zugehörigen Wert aus Zeile Nummer **z** der Matrix zurück.

2.9.8 Zielwertsuche und Solver

Tabellenkalkulationen bieten die Möglichkeit, mit der sogenannten Zielwertsuche einen Zellwert iterativ so lange zu verändern, bis das Ergebnis einer linearen Formel einem gewünschten Wert entspricht.

Beispiel (Abb. 29): **A1** enthält die Höhe eines Rechtecks, **A2** die Breite und **A3** die Formel für die Fläche, **=A1*A2**. Gesucht wird die Breite, die bei einer Höhe von **8** die Fläche **20** ergibt.

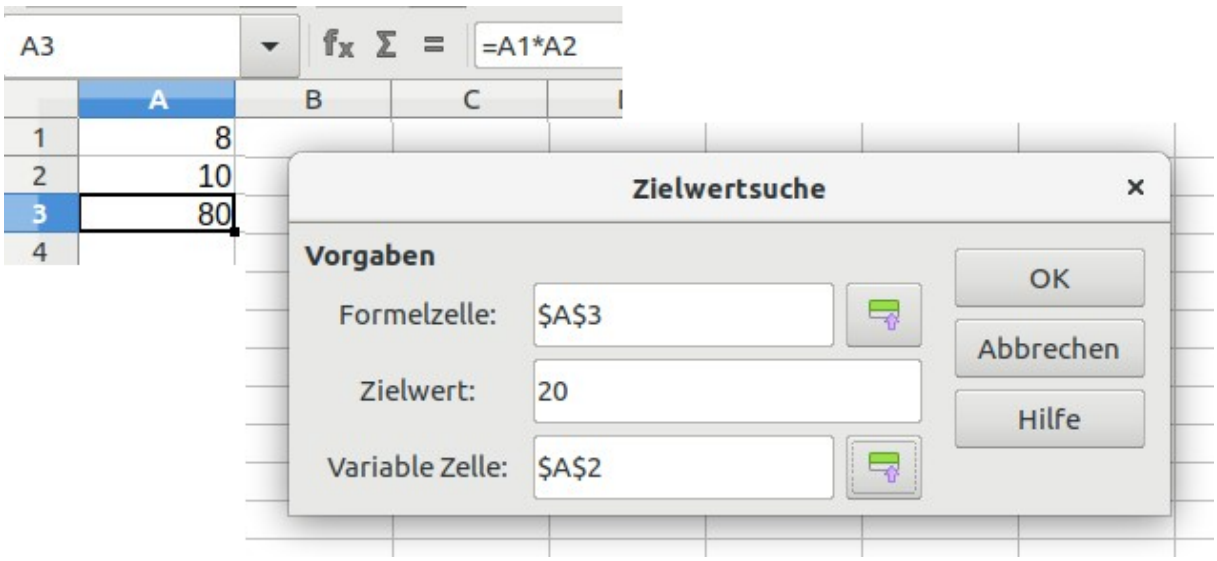


Abb. 29: Zielwertsuche

Die Zielwertsuche findet schnell das gesuchte Ergebnis, sie kann aber nur eine einzige Zelle verändern. Sind mehrere Eingangswerte einer Formel zu variieren, um ein bestimmtes Ergebnis zu erhalten, so greifen wir auf einen sogenannten „Solver“ zurück.

Angenommen, wir möchten herausfinden, welches die geringste Zahl von Cent-Münzen ist, mit der wir einen Betrag von 12,37 Euro zusammenstellen können. In Abb. 30 sind dazu in der mittleren Spalte „Münzwert“ (in den Feldern C2 bis C7) die Münzwerte von einem bis 50 Cent eingetragen. Links daneben in den Feldern B2 bis B7 ist die jeweilige Anzahl verzeichnet. In der Spalte „Produkt“ (D2 bis D7) wird das jeweilige Produkt aus Anzahl und Münzwert berechnet. In der mit „Summe“ gekennzeichneten Zeile 8 der Tabelle werden schließlich die Summen der Einträge der Spalten B und D gebildet.

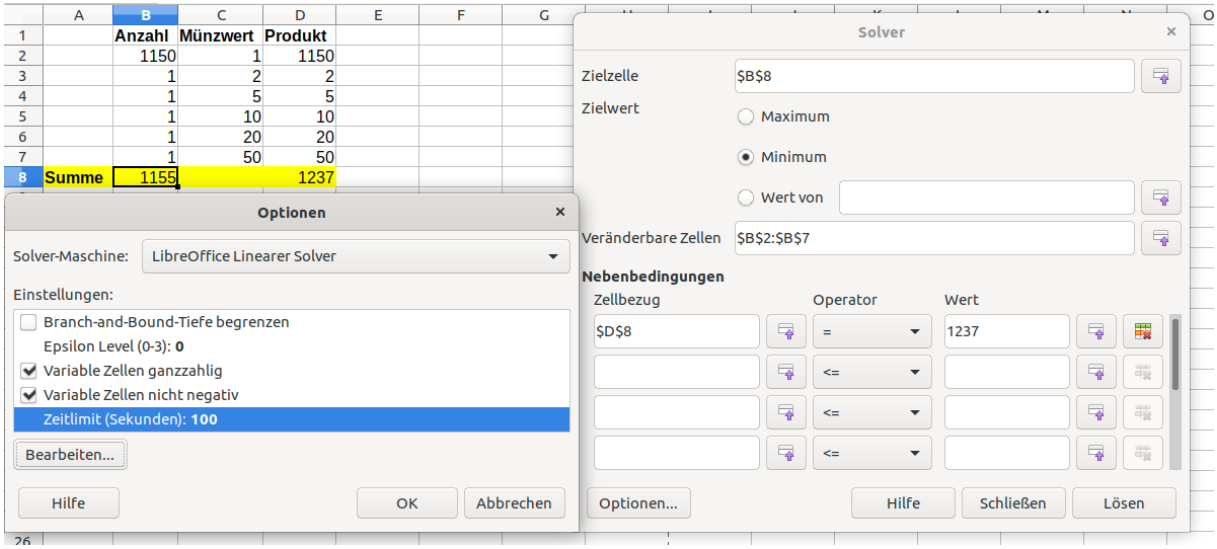


Abb. 30: Solver

Der Solver variiert nun alle Felder im Bereich B2:B7 so lange, bis die Anzahl der Münzen den kleinstmöglichen Wert erreicht, wobei stets die „Nebenbedingung“ einzuhalten ist, dass die Gesamtsumme der Münzwerte in Feld D8 genau 1237 Cent beträgt. Weil abzählbare Mengen, wie die gesuchte Anzahl von Münzen, immer ganzzahlig und positiv sind, vermerken wir dies zuvor in den Optionen des Solvers, da sonst keine Lösung gefunden werden kann.

Die beiden Funktionen „Zielwertsuche“ und „Solver“ sind in unterschiedlichen Tabellenkalkulation mehr oder weniger gut zu finden.

In LibreOffice sehen wir sie im Menü „Extras“.

Wer Microsoft Office Excel 2019 verwendet, schaltet zum Menübandregister „Daten“ um, klickt in der Icongruppe „Datentools“ in der dritten Spalte auf das Icon „Was-wäre-wenn-Analyse“ und wählt dort den Untermenüeintrag „Zielwertsuche“ aus. In der Icongruppe „Analyse“ (falls vorhanden) existiert mit etwas Glück auch der Eintrag „Solver“. Der Solver ist in Microsoft Office Excel allerdings standardmäßig nicht enthalten und muss dort üblicherweise als „Plugin“ nachinstalliert werden.

2.9.9 Matrixformeln

Tabellenkalkulationen können sehr komplexe Berechnungen über rechteckige Bereiche ausführen, die der Matrizenrechnung in der Mathematik entsprechen.

Angenommen, wir haben ein Gleichungssystem mit drei Unbekannten zu lösen.

$$\begin{array}{rrrcr} 4x & -2y & +2z & = & 184 \\ -3x & +5y & +4z & = & 29 \\ 4x & -7y & +12z & = & 749 \end{array}$$

Diese Gleichungen können wir in Matrixschreibweise so formulieren:

$$\begin{pmatrix} 4 & -2 & 2 \\ -3 & 5 & 4 \\ 4 & -7 & 12 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 184 \\ 29 \\ 749 \end{pmatrix}$$

Um die Unbekannten auf die rechte Seite zu bekommen, bilden wir den „Kehrwert“ (die Inverse) der Matrix auf der linken Seite.

$$\begin{pmatrix} 4 & -2 & 2 \\ -3 & 5 & 4 \\ 4 & -7 & 12 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 184 \\ 29 \\ 749 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Um die Matrixgleichung zu lösen, brauchen wir also eine Funktion, die eine Matrix invertiert und eine Funktion, die eine Matrix mit einem Vektor multipliziert. Wir finden diese Funktion in Calc und Excel als **`MINV`** und **`MMULT`**.

	A	B	C	D	E
1	4	-2	2	184	{=MMULT(MINV(A1:C3); D1:D3)}
2	-3	5	4	29	{=MMULT(MINV(A1:C3); D1:D3)}
3	4	-7	12	749	{=MMULT(MINV(A1:C3); D1:D3)}

Abb. 31: Matrixformeln

Das einzig wirklich komplizierte an Matrixformeln ist ihre Eingabe. Sie müssen dazu zuerst den Bereich markieren, in dem die Formel gelten soll (hier die Zellen von E1 bis E3), dann tippen Sie die Formel ohne die umgebenden geschweiften Klammern ein und drücken schließlich die Tastenkombination Steuerung–Umschalten–Eingabetaste (Strg ⇧ ↵).

Sie sollten nun die Lösung des Gleichungssystems vor sich sehen.

4	-2	2	184	12
-3	5	4	29	-23
4	-7	12	749	45

Abb. 32: Gelöstes Gleichungssystem

Die gesuchten drei Unbekannten lauten also x = 12, y = -23, z = 45.

2.9.10 Diagramme

Alle heutigen Tabellenkalkulationen bieten uns komfortable Möglichkeiten an, um einfache Diagramme aus eingetragenen oder berechneten Zellinhalten zu erzeugen. In der Regel müssen wir dazu nur den zu visualisierenden Datenbereich markieren (meistens genügt es schon, wenn sich die zuletzt angeklickte Zelle im Datenbereich befindet) und im Menü „Einfügen“ den Menüpunkt „Diagramm“ wählen.

Im Ingenieurwesen ist insbesondere die Diagrammform „x-y-Diagramm“ von Wert, da diese einen klaren Koordinatenbezug zwischen zwei Wertegruppen herstellen kann. In LibreOffice und Microsoft Office Excel wählen wir dazu den Diagrammtyp „XY“ aus. In Apples Numbers ist entsprechend in der Rubrik für 2D-Diagramme die Auswahlschaltfläche mit einzelnen Punkten anzuklicken.

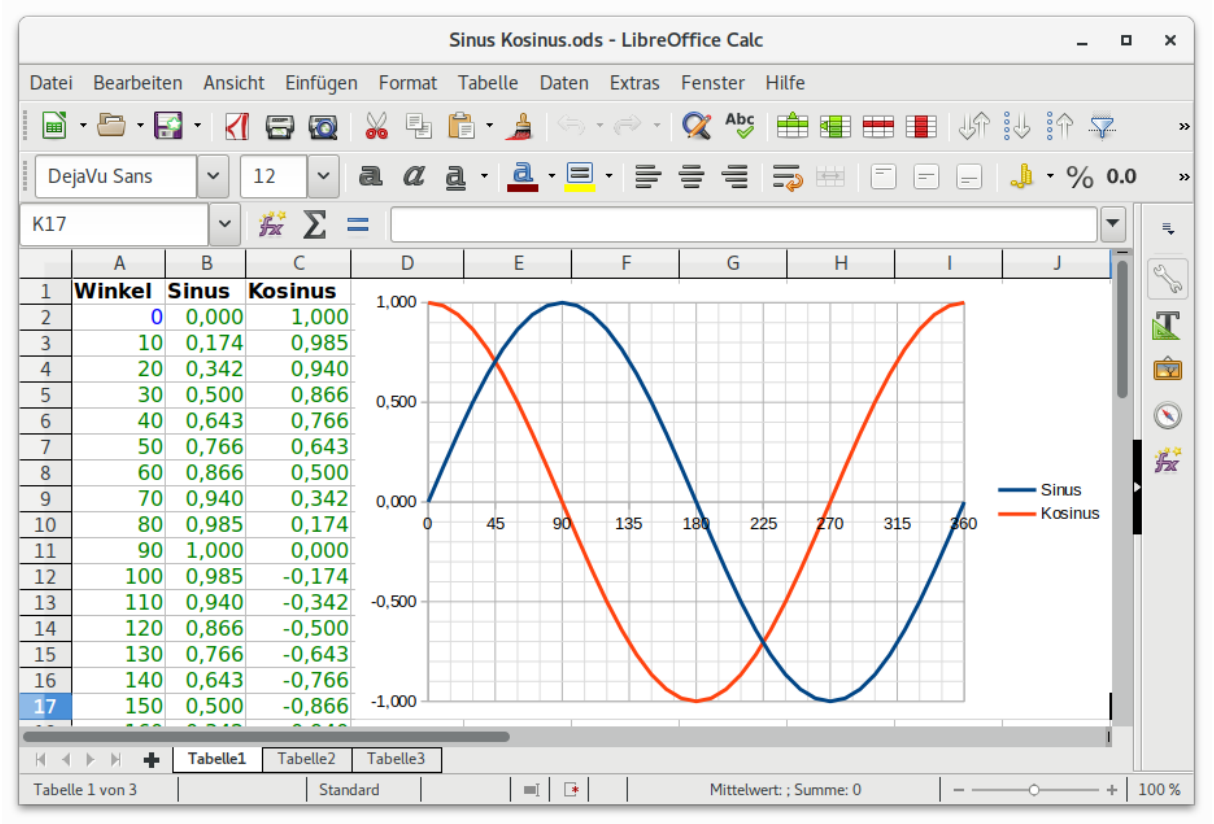


Abb. 33: x-y-Diagramm

2.9.11 CSV-Dateien und Tabellenkalkulationen

Tabellenkalkulationen unterstützen neben ihrem eigenen, vereinzelt extrem komplizierten[1], Dateiformat immer auch ein sehr simples und universelles Datenaustauschformat, in dem die einzelnen Daten (Zahlen oder Texte) zeilenweise und durch Kommas getrennt vorliegen. Nach diesen „kommagetrennten Werten“ oder „comma separated values“ heißen diese Dateien CSV-Dateien.

```
Mustermann,Max,21,Bochum,44801
Exempel,Elvira,20,Gelsenkirchen,45879
```

1 Die Beschreibung des unter abenteuerlichen Bedingungen (http://www.groklaw.net/article.php?story=2008032913190768) vorübergehend zur internationalen Norm gewordenen Microsoft-Office-Dateiformats OOXML ist über 6000 Seiten lang. Und sie ist unvollständig. https://de.wikipedia.org/wiki/Office_Open_XML

Texte werden in CSV-Dateien in der Regel in ASCII-Anführungszeichen gesetzt. Anstelle des Kommas ist als Trennzeichen oft auch ein Semikolon oder ein Tabulatorzeichen üblich.

```
"Mustermann";"Max";21;"Bochum";44801
"Exempel";"Elvira";20;"Gelsenkirchen";45879
```

Formeln, Formatierungen und Diagramme gehen beim Speichern einer Tabelle als CSV-Datei verloren.

Beim Import von Daten aus CSV-Dateien müssen wir zahlreiche Informationen ergänzen. Es muss klar sein, wie das Dezimaltrennzeichen aussieht, ob und wie Datumswerte interpretiert werden, in welcher Zeichenkodierung eventuell vorhandene Umlaute und Sonderzeichen vorliegen und noch einiges mehr.

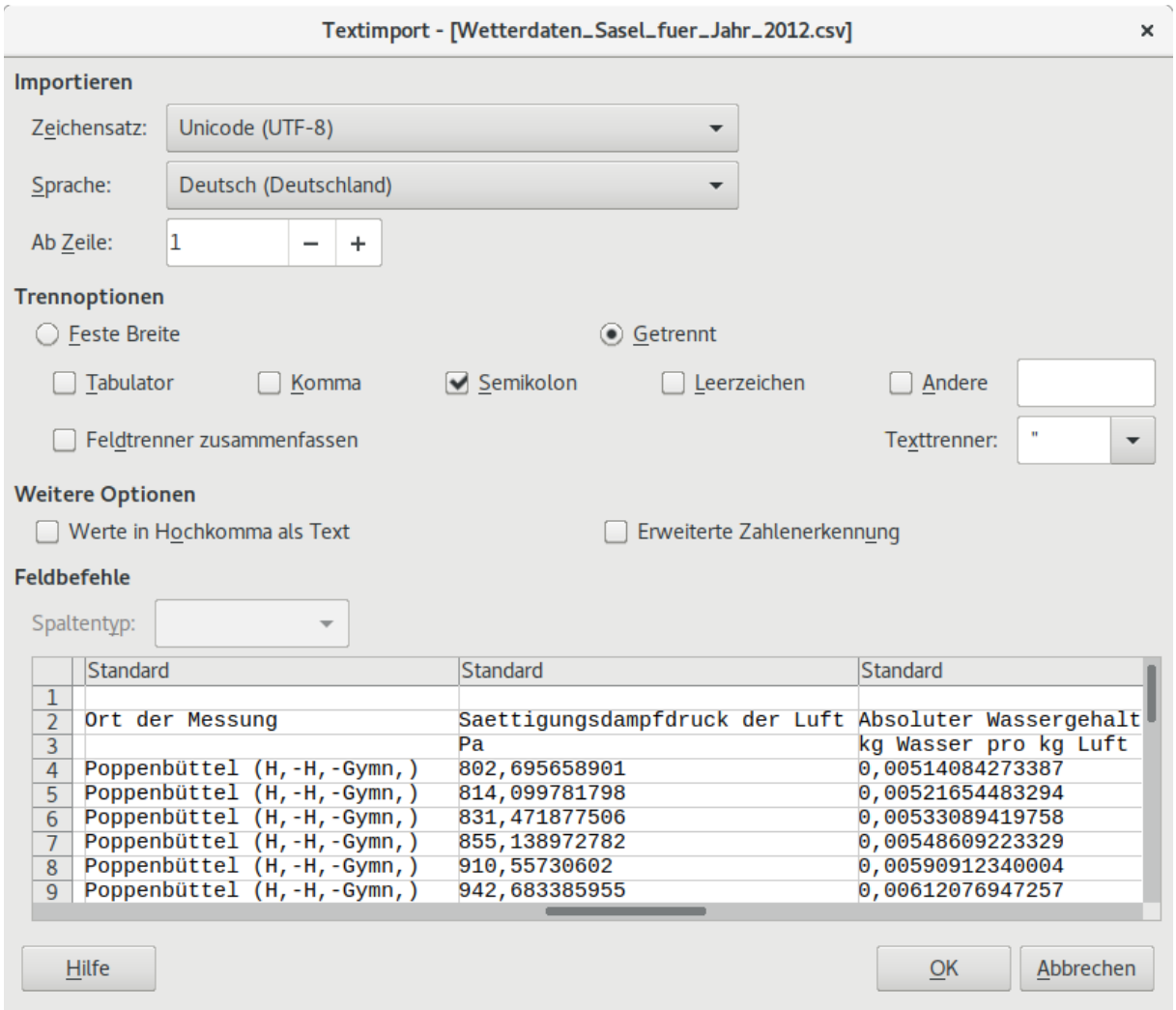


Abb. 34: CSV-Import-Dialog in LibreOffice Calc

Beim Öffnen einer CSV-Datei zeigt LibreOffice daher immer einen Textimport-Dialog an, in dem die aktuell benötigten Einstellungen überprüft werden können. Die Wahl des Dezimal- und Tausendertrennzeichens wird dabei indirekt über die Spracheinstellung vorgenommen. Ist das Dezimalzeichen ein Punkt anstelle eines Kommas, so sollte die Sprache auf „Englisch“ geändert werden.

Leider unterschlägt die verbreitete Tabellenkalkulation „Microsoft Office Excel“, diesen Dialog und verwendet nicht dokumentierte „Standardeinstellungen[1]“, was gelegentlich zu Schlagzeilen führt, wenn zum Beispiel entdeckt wird, dass 20 % aller Gentechnik-Studien wegen Excels unverlangter Datumsinterpretation importierter Daten fehlerhaft sind[2].

In Excel darf eine CSV-Datei daher niemals einfach geöffnet werden, sondern sie muss mithilfe des Textimportassistenten, der im Menübandregister „Daten“ in der Gruppe „Externe Daten abrufen“ hinter dem Eintrag „Aus Text“ versteckt ist, importiert werden.

Excels notorischer Drang, alle möglichen Kombinationen aus Zahlen und Trennzeichen in CSV-Dateien als Datumsangabe fehlzuinterpretieren, hat sogar schon zu diversen populären Memes geführt (Abb. 35).

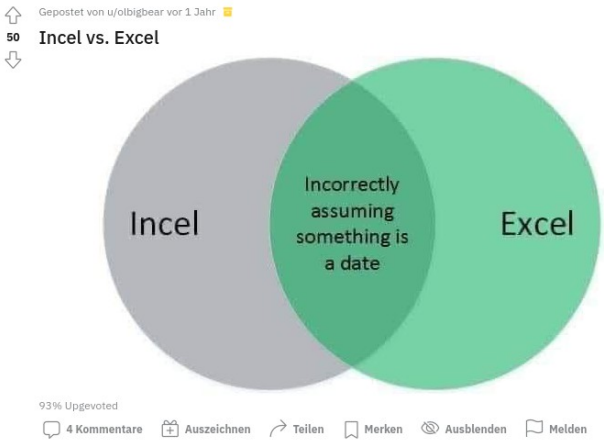


Abb. 35: Meme „Incel vs. Excel“

1 https://support.office.com/de-de/article/Importieren-oder-Exportieren-von-Textdateien-TXT-oder-CSV-5250ac4c-663c-47ce-937b-339e391393ba – ich möchte mich übrigens an dieser Stelle gerne dahingehend korrigieren lassen, dass die Standardeinstellungen doch irgendwo dokumentiert sind. Zuschriften mit Links auf die entsprechende Dokumentation bitte per Mail!

2 Gene name errors are widespread in the scientific literature
Mark Ziemann, Yotam Eren und Assam El-Osta
Genome Biology 2016 17:177
https://doi.org/10.1186/s13059-016-1044-7
Abgerufen: 23 August 2016

2.9.12 Anwendungsgrenzen

Obwohl Tabellenkalkulationen im Laufe der Jahrzehnte theoretisch immer größere Datenmengen verarbeiten können, liegt ihr sinnvolles Haupteinsatzgebiet weiterhin im Bereich überschaubarer Tabellengrößen. Für den Umgang mit wirklich großen Datenmengen ist die Verwendung von Tabellenkalkulationen in der Regel zu unhandlich und zu fehleranfällig.

US-Amerikanische IT-Forscher stellten in einer Metastudie fest, dass 88% aller untersuchten Excel-Tabellen Fehler enthalten[1]. Für die Verwaltung großer Datenbestände[2] sind Datenbankprogramme besser geeignet und für die ingenieurmäßige Verarbeitung und Analyse komplexer Daten und großer Datenmengen lernen wir die Sprache Python als wesentlich eleganteres und mächtigeres Werkzeug kennen.

Maximale Zeilenzahl	
Excel 95	16.384 (2^{14})
Excel 97	65.536 (2^{16})
Excel 2007, LibreOffice 3.3 (2011)	1.048.576 (2^{20})
Google Docs (Stand 2020)	1562 bis 400.000, je nach Spaltenzahl
Maximale Spaltenzahl	
Excel 5, Google	256: A…Z, AA, AB…IV
LibreOffice 3.3	1024: A…AMJ
Excel 2007	16.384: A…XFD

Maximale Zellenzahl	
Google Docs (Stand 2020)	400.000
LibreOffice	1.073.741.824 (2^{30})

1 What We Know About Spreadsheet Errors
Raymond R. Panko
Journal of End User Computing's, Band 10, Nr. 2, Seiten 15-21
http://panko.shidler.hawaii.edu/SSR/Mypapers/whatknow.htm
Abgerufen: 27. August 2019

2 Excel: Why using Microsoft's tool caused Covid-19 results to be lost
BBC / Leo Kelion
https://www.bbc.com/news/technology-54423988
Abgerufen: 13. Oktober 2020

3 Hypertext

Bevor wir in Kapitel 4 mit der Einführung in die Programmierung beginnen, wollen wir zunächst einmal einen Blick in die formell strukturierte Welt einer typischen „Computersprache“ werfen.

Eine der bekanntesten Sprachen, mit denen Texte so geschrieben werden können, dass sie sowohl von Menschen gelesen als auch von Computern richtig interpretiert werden, trägt den Namen „HTML“.

Die *„Hypertext Markup Language“* HTML stellt Möglichkeiten zur Verfügung, Textstellen so zu markieren, dass Verbindungen zwischen ihnen hergestellt werden können, dass ihre Struktur gut wiedergegeben werden kann und dass Informationselemente wie Listen, Bilder und Tabellen einfach und ohne besondere Werkzeuge eingefügt werden können.

Als Hypertext bezeichnen wir dabei einen strukturierten Text, der aktivierbare Komponenten enthält, mit denen eine Navigation von einer Textstelle zu einer bestimmten anderen Textstelle im selben oder einem anderen Hypertext möglich ist.

Die Start- und Zielpunkte dieser Verweise nennen wir „Anker“.

Die Verweise selbst werden als „Links“ oder auch „Hyperlinks“ bezeichnet. Da das englische Wort *„link“* auch „Kettenglied“ bedeutet, finden sich in Menüleisten oft Kettensymbole zum Einfügen eines Hyperlinks.

HTML ist zunächst einmal keine Programmiersprache: Sie ist nicht ohne weiteres dazu geeignet, wiederkehrende Prozesse zu automatisieren oder Berechnungen durchzuführen. Wir werden sie aber im Laufe dieses Semesters immer wieder dazu verwenden, Texte zu strukturieren und Inhalte für Webbrowser darstellbar zu machen.

Hinter nahezu jeder Webseite steckt ein HTML-Text. Um diesen sichtbar zu machen, genügt es in den meisten Webbrowsern unter Windows oder Linux, die Tastenkombination Strg U zu drücken. Was auf den ersten Blick wie ein chaotisches Gewimmel aus spitzen Klammern und kryptischen Abkürzungen anmutet, besitzt überraschend oft eine klare Struktur.

3.1 HTML-Tags

Die Markierungen, die in gewöhnliche Texte eingefügt werden müssen, damit aus diesen Hypertexte werden, heißen „*Tags*“. In HTML-Dateien sind sie auf den ersten Blick dadurch erkennbar, dass sie von spitzen Klammern umschlossen werden.

Üblicherweise treten HTML-Tags paarweise auf. Zu einem öffnenden Tag (Starttag) gehört ein schließendes Tag (Endtag), welches mit einem Schrägstrich eingeleitet wird.

Das englische Wort „*tag*“ (Aussprache: „tähg“) bedeutet nicht nur Markierung, sondern auch Etikett oder Auszeichnung. Als „RFID-Tags“ kennen Sie bestimmt die verbreiteten Funketiketten zur Diebstahlsicherung, in Musikdateien werden die Informationen zu Titel und Künstler im „ID3-Tag“ gespeichert und die Reviermarken von Sprühlackschmierern heißen ebenfalls „Tags“.

Abb. 36: Nerdwitz. Foto: Markus Tacker, Lizenz: CC BY-ND 2.0

3.2 Hierarchische Ordnung

Starttag, Inhalt und Endtag bilden gemeinsam ein HTML-Element. Diese Elemente können ineinander verschachtelt sein, dürfen sich aber nicht überschneiden. Bevor ein übergeordnetes Element durch ein Endtag geschlossen wird, müssen zuerst alle untergeordneten Elemente geschlossen werden.

Diese strenge Ordnung ist der Sprache XML (*Extensible Markup Language*) geschuldet, auf deren Struktur HTML und viele andere zur computerbasierten Verarbeitung entworfenen Sprachen aufbauen.

Ein einfaches Beispiel der hierarchischen Ordnung in einem HTML-Text zeigt das Diagramm in Abbildung 37. Das HTML-Element umfasst den gesamten Inhalt und gliedert sich in einen Kopfteil (*head*) und einen Inhaltsteil (*body*).

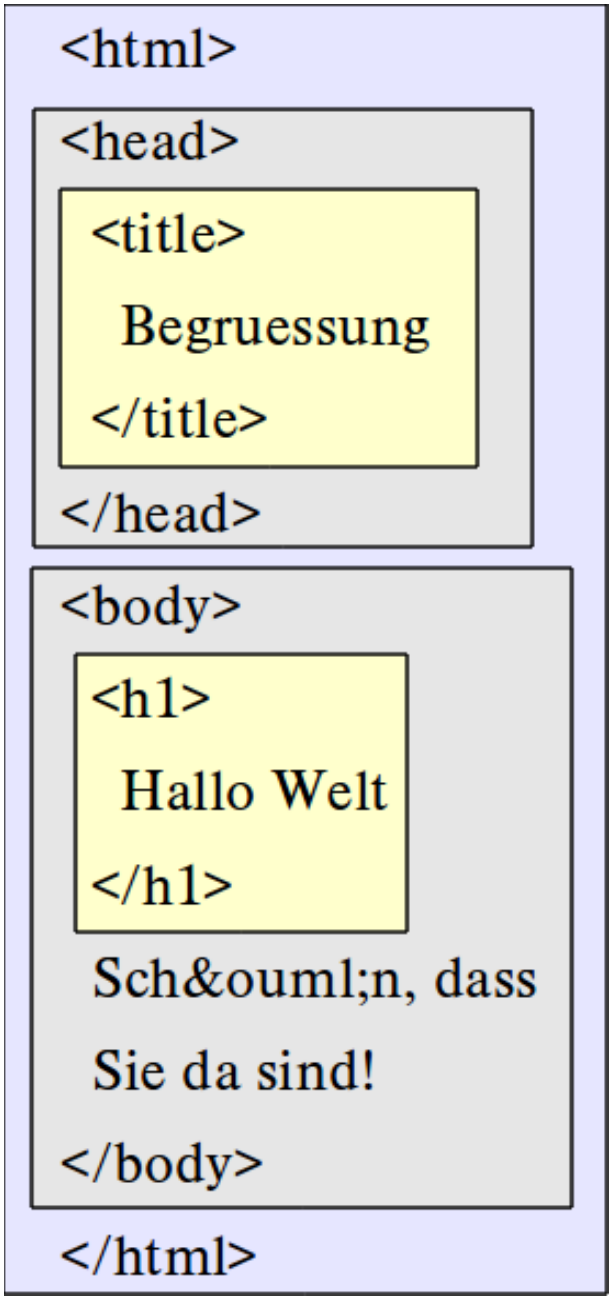


Abb. 37: HTML-Struktur

Der Kopfteil enthält hier nur das Element „*title*“. Es legt den vorderen Teil der Titelzeile des im nächsten Bild zu sehenden Webbrowsers fest. Üblicherweise finden wir im Kopfteil Angaben zu Autor und Zeichenkodierung sowie Hinweise für Suchmaschinen und diverse andere Informationen, die auf der Webseite hinterher nicht direkt zu sehen sind.

Der Inhaltsteil enthält den darzustellenden Text. In diesem Fall ist es der Ausruf „Schön, dass Sie da sind!“, welchem eine Überschrift ersten Grades mit dem Inhalt „Hallo Welt” vorangestellt ist.

Dass in dem Beispiel ausgerechnet das Wort „Schön“ so unschön als „Schön“ daherkommt, hat den Grund, dass diese Ersatzdarstellung auch unter ungünstigsten technischen Randbedingungen noch funktioniert. Mehr dazu in Kapitel 3.4.

Diese HTML-Datei wird in einem Webbrowser vermutlich so ähnlich dargestellt werden, wie es Abbildung 38 zeigt[1].

Abb. 38: Ein Browser stellt HTML-Seiten dar

Es ist gut möglich, dass das Erscheinungsbild in unterschiedlichen Browsern variiert, denn welche Schriftart verwendet wird, welche Abstände die Textbereiche untereinander haben und wie groß die verschiedenen Texte sind, steht nicht in der HTML-Datei. Es ist auch keine gute Idee, das durch „geschickte“ Verwendung von HTML-Strukturelementen festlegen zu wollen. Die Aufgabe der Sprache HTML liegt in der inhaltlichen Strukturierung eines Textes. Für das Layouten gibt es geeignetere Mittel (siehe Kapitel 3.5).

1 Solche Abbildungen eines Bildschirmfensters oder eines kompletten Bildschirms nennt man „Bildschirmfoto” oder auch „Screenshot“.

3.3 Attribute

Manche HTML-Tags sind mit Zusatzinformationen versehen, die wir „Attribute“ nennen. Attribute haben einen Namen und ihnen kann mithilfe eines Gleichheitszeichens ein Wert zugewiesen werden.

Eine Textstelle, die auf eine andere Webseite verweisen soll, benötigt beispielsweise die Information, wie die Adresse der Seite lautet, die beim Anklicken angesprungen werden soll. Wir verwenden dazu das Ankertag „a”, indem wir ein HTML-Element anlegen, das aus dem zu markierenden Text besteht, der mit einem Start- und Endtag umschlossen ist und weisen im Starttag dem Attribut mit dem Namen „href“ den Wert „http://www.hs-bochum.de/” zu.

In einem geeigneten Texteditor sieht das fertige HTML-Element dann so aus wie der Inhalt des grauen Feldes:

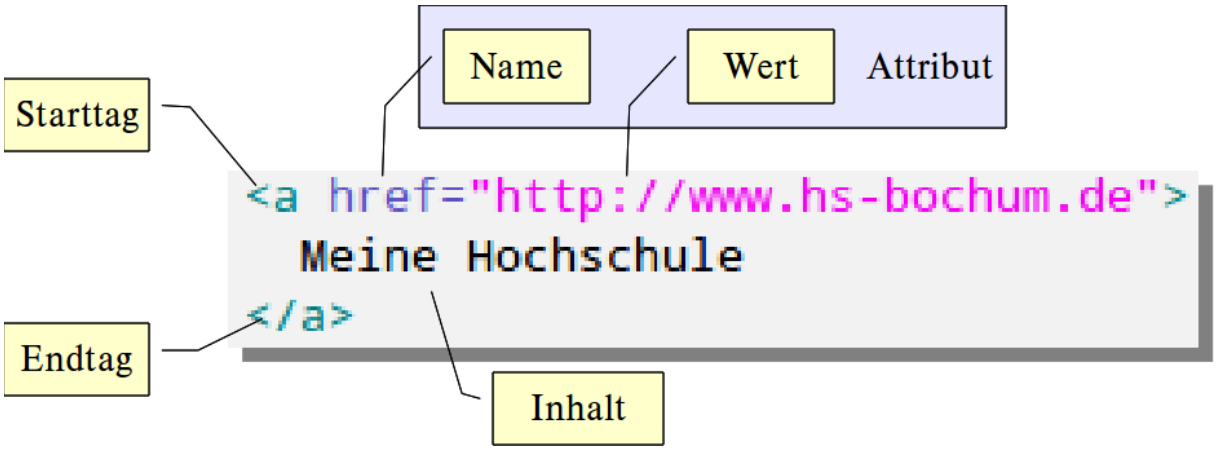


Abb. 39: Bestandteile eines HTML-Elements

3.4 HTML-Entitäten

HTML-Dateien müssen von unterschiedlichsten Rechnern verarbeitet werden. Manche dieser Geräte unterstützen jedoch nur eingeschränkte Einzelbyte-Zeichensätze mit wenig mehr als zweihundert unterschiedlichen Zeichen[1].

In HTML-Dateien können wir auch dann sämtliche Unicode-Zeichen verwenden, wenn die Datei selbst nur in einer Zeichenkodierung gespeichert wird, die nur den minimalen 7-Bit-ASCII-Zeichensatz mit weniger als hundert darstellbaren Zeichen kennt.

Den wichtigsten Sonderzeichen sind dabei kurze Namen zugeordnet. Die Kombination aus solch einem Namen, einem einleitenden Ampersand und einem abschließenden Semikolon nennen wir HTML-Entität.

Ein kleines **β** (Beta) lässt sich beispielsweise durch die HTML-Entität **β** umschreiben, das größer-gleich-Zeichen **≥** wird durch **≥** dargestellt und das Unendlich-Symbol **∞** wird mit **∞** umschrieben.

Auf http://unicode.e-workers.de/entities.php finden Sie eine Tabelle mit den wichtigsten HTML-Entitäten.

Eine vollständige sortierte Übersicht über sämtliche verfügbaren HTML-Entitäten erhalten Sie mit dem folgenden Python-Programm:

```
import html.entities
for i,j in sorted(html.entities.html5.items(),
                  key=lambda x:x[0].lower()):
    try:
        if ";" in i:
            print(j+"\t&"+i)
    except UnicodeEncodeError:
        pass
```

1 Siehe auch Kapitel 6.2, Zeichenkodierung – von ASCII bis Unicode

3.5 CSS

Ein sauberes HTML-Dokument enthält nur Informationen und Strukturelemente wie Überschriften, Textabsätze oder Tabellen. Das Aussehen auf dem Bildschirm oder Papier ist davon weitgehend unabhängig.

Um einem HTML-Dokument ein Layout mit bestimmten Schriftgrößen, Fonts, Farben und Abständen zuzuordnen, verwendet man die sogenannten „*Cascaded Style Sheets*“. Sie enthalten Anweisungen, wie bestimmte HTML-Elemente zu formatieren sind.

Auf diese Weise können Form und Inhalt sauber getrennt werden und es ist mit wenig Aufwand verbunden, denselben HTML-Text für ganz verschiedene Ausgabemedien, vom Mobiltelefon bis zum A3-Blatt, ansprechend zu gestalten.

In dieser Auflage des Skripts gehen wir nicht weiter auf das Layouten von HTML-Dokumenten durch CSS ein. Für das Selbststudium ist https://wiki.selfhtml.org/wiki/CSS ein ganz guter Einstieg.

4 Algorithmen und ihre Darstellung

Ein Algorithmus ist eine Handlungsvorschrift, mit der sich eine Aufgabe in einer endlichen Zahl von Lösungsschritten abarbeiten lässt.

Algorithmen haben zunächst einmal gar nichts mit Computern zu tun. Jede Verwaltungsvorschrift in einer Behörde kann ein Algorithmus sein, wenn sie die oben genannte Definition erfüllt. Bevor Sie damit beginnen, ein Computerprogramm zur Lösung eines Problems zu schreiben, ist es eine gute Idee, wenn Sie sich zuerst über den zugrunde liegenden Algorithmus Gedanken machen. Welche Schritte müssen in welcher Reihenfolge ausgeführt werden? Gibt es Schritte, die wiederholt werden müssen? Gibt es Schritte, die nur unter bestimmten Bedingungen ausgeführt werden?

Durch eine grafische Darstellung dieses Algorithmus lässt sich die Umsetzung in ein Programm oft erheblich vereinfachen. Für die Programmdokumentation ist ein Diagramm der Programmstruktur zudem ein unverzichtbarer Bestandteil, um eine gewählte Lösung nachvollziehbar festzuhalten.

4.1 Flussdiagramm

Bei sehr einfachen Algorithmen lässt sich die Reihenfolge der auszuführenden Schritte grafisch mit einem Flussdiagramm veranschaulichen.

Pfeile zeigen die Reihenfolge der Ausführung an. Berechnungen werden durch Rechtecke dargestellt, Ein- und Ausgabevorgänge durch Parallelogramme und Fallunterscheidungen mit Rauten.

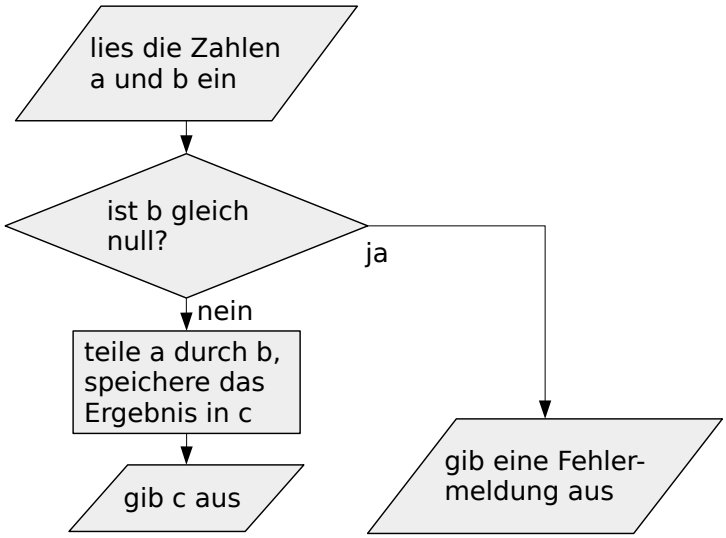


Abb. 40: Ein einfaches Flussdiagramm

Bei komplexeren Algorithmen mit vielen Fallunterscheidungen oder mit Schleifen, in denen Abschnitte wiederholt ausgeführt werden, verlieren Flussdiagramme schnell ihre Übersichtlichkeit. Zur Darstellung von nicht trivialen Algorithmen verwenden wir deshalb in der Regel die im folgenden Kapitel vorgestellten Struktogramme.

4.2 Struktogramm

Auch, wenn es auf den ersten Blick anders erscheinen mag: Struktogramme stellen eine besonders übersichtliche Form dar, um Algorithmen unabhängig von der Programmiersprache, in der sie später[1] umgesetzt werden, grafisch darzustellen.

Die Bearbeitungsreihenfolge der einzelnen Arbeitsschritte wird im Struktogramm streng von oben nach unten abgebildet, Schleifen werden eingerückt und bei Fallunterscheidungen fächert sich das Struktogramm in mehrere Spalten auf. Die im Flussdiagramm überall vorhandenen Pfeile gibt es in Struktogrammen nicht.

Struktogramme sind nach DIN 66261 genormt[2]. Nach ihren Erfindern Isaac Nassi und Ben Shneidermann werden sie auch als Nassi-Shneidermann-Diagramme bezeichnet.

Gelegentlich findet man im Netz Struktogramme, in denen anstelle einer sprachunabhängigen Darstellung eines Algorithmus Python- oder Java-Programmcode enthalten ist. Das widerspricht dem Sinn dieser Darstellungsart und sollte von uns nicht übernommen werden.

4.2.1 Reihenfolge der Arbeitsschritte

Arbeitsschritte, die nacheinander (sequenziell) ausgeführt werden sollen, werden im Struktogramm als untereinander liegende Rechtecke von gleicher Breite dargestellt. In den Rechtecken steht der jeweilige Arbeitsschritt in stichwortartiger Kurzform.

---

1 Lassen Sie mich hier bitte wenigstens so tun, als wüsste ich nicht, dass die meisten kleineren Programme nicht so herum entstehen, sondern bei Bedarf schnell in die Tastatur gehackt und, wenn überhaupt, erst danach dokumentiert werden.

2 Als Mitglied der Hochschule Bochum können Sie diese und viele weitere Normen für Sie kostenlos von https://nautos.de/U2P/login herunterladen, solange sie sich im Netzwerk der Hochschule befinden.

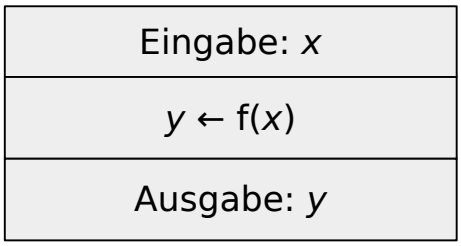


Abb. 41: Struktogramm: Sequenz von Arbeitsschritten

Um zu zeigen, dass in einem Arbeitsschritt einer Variable ein Wert zugewiesen wird, verwenden wir als Zuweisungszeichen „wird zu“ einen nach links gerichteten Pfeil „←“. Wir vermeiden dadurch jede Verwechslungsgefahr mit dem Vergleichsoperator „ist gleich“. Gelegentlich findet man in Struktogrammen auch die auf Schreibmaschinen und PCs ohne Compose-Taste[1] leichter zu tippende Ersatzdarstellung „:=“ anstelle eines Pfeils.

Anstelle von „Eingabe:“ und „Ausgabe:“ können wir auch kürzer „E:“ und „A:“ schreiben.

4.2.2 Fallunterscheidung

Gibt es in einem Algorithmus zwei Ausführungsalternativen, die von einer zu treffenden Entscheidung abhängen, so teilt sich das Struktogramm darunter in zwei Spalten auf.

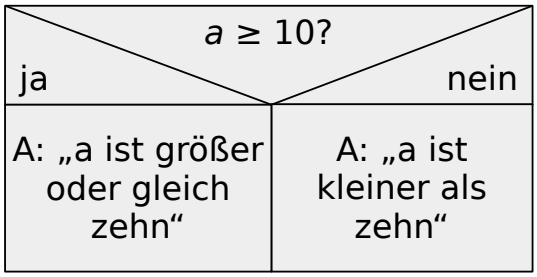


Abb. 42: Struktogramm: Fallunterscheidung

Die Kopfzeile der Fallunterscheidung ist dreigeteilt. Das obere Dreieck enthält die Frage und die beiden unteren Dreiecke die beiden möglichen Antworten.

1 Ich halte es für eine gute Idee, die für die meisten Menschen völlig überflüssige Feststelltaste (⇩, links neben der Taste „A“) zur Compose-Taste umzuwidmen. Den Pfeil nach links tippen Sie dann einfach mit der Tastenfolge „Compose < -“. Siehe auch https://de.wikipedia.org/wiki/Compose-Taste.

Je nachdem, welche Antwort die richtige ist, wird der Ausführungszweig in der linken oder in der rechten Spalte betreten.

Ausführungszweige dürfen auch leer sein.

Wird der Algorithmus im weiteren Verlauf wieder unabhängig vom gewählten Ausführungszweig abgearbeitet, so werden die gemeinsamen Ausführungsschritte wieder in voller Breite dargestellt.

4.2.3 Mehrfachauswahl

Gibt es mehr als zwei Wahlmöglichkeiten, von denen immer nur eine ausgeführt wird, so gliedert sich das Struktogramm in entsprechend viele Spalten auf.

x ist …		
< 10	> 10	sonst
A: „x ist kleiner als zehn“	A: „x ist größer als zehn“	A: „x ist gleich zehn“

Abb. 43: Struktogramm: Mehrfachauswahl

Die letzte Wahlmöglichkeit („sonst“) ergibt sich aus den anderen und muss nicht mehr explizit abgefragt werden. Diese Spalte der Fallunterscheidung wird im Struktogramm gegenüber den anderen abgesetzt.

4.2.4 Abweisende Schleife

Schleifen sind Abschnitte in einem Algorithmus, die mehrmals hintereinander wiederholt werden können.

Über die Anzahl der Wiederholungen entscheidet die im Schleifenkopf formulierte Schleifenbedingung. Vor jedem Schleifendurchlauf wird die Schleifenbedingung geprüft. Ist sie erfüllt, wird der Schleifendurchlauf ausgeführt.

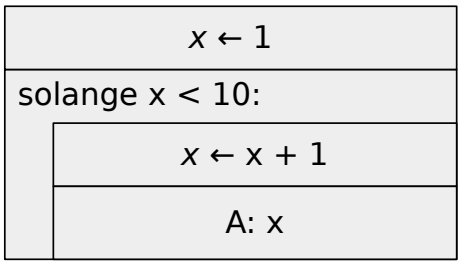


Abb. 44: Struktogramm: Schleife

Wenn die Schleifenbedingung bereits bei der ersten Prüfung nicht erfüllt ist, wird der Schleifenkörper niemals betreten. Man nennt diesen Schleifentyp daher auch abweisende Schleife. Weil die Abfrage im Schleifenkopf erfolgt, werden abweisende Schleifen auch „kopfgesteuerte Schleifen“ genannt.

Der zu wiederholende Schleifenkörper ist im Struktogramm stets eingerückt.

4.2.5 Nichtabweisende Schleife

Wenn die Schleifenbedingung erst zum Ende der Schleife geprüft wird und der Schleifenkörper daher immer mindestens einmal durchlaufen wird, spricht man von einer nichtabweisenden Schleife. Analog zur „kopfgesteuerten“ abweisenden Schleife werden nichtabweisende Schleifen auch „fußgesteuerte Schleifen“ genannt.

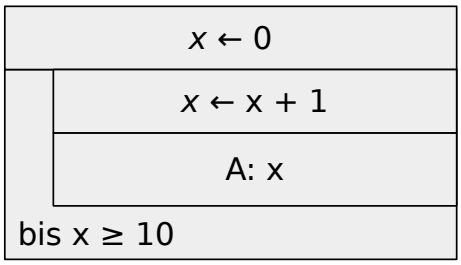


Abb. 45: Struktogramm: Nichtabweisende Schleife

4.2.6 Endlosschleife

Endlosschleifen sind Schleifen, deren Schleifenbedingung immer erfüllt ist. Sie werden in einem Struktogramm durch einen nach rechts eingerückten „schwebenden“ Block dargestellt.

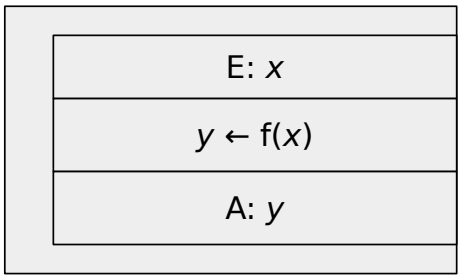


Abb. 46: Struktogramm: Endlosschleife

Eigentlich dürften Endlosschleifen niemals Teil eines Algorithmus' sein, denn dessen Definition lautet ja, dass eine Aufgabe in *endlich* vielen Schritten abzuarbeiten ist.

Ausbruch aus der Endlosschleife

Endlosschleifen in Struktogrammen besitzen daher eine Art Notausgang. Dazu wird eine Abbruchbedingung im Schleifenkörper abgefragt. Ist sie erfüllt, erfolgt ein gezielter Aussprung. Diesen Aussprung stellen wir im Struktogramm durch ein leeres[1] Feld dar, in dem durch zwei schräge Linien ein Pfeil nach außen angedeutet ist.

Der Algorithmus wird nach dem Aussprung unterhalb der Schleife fortgesetzt.

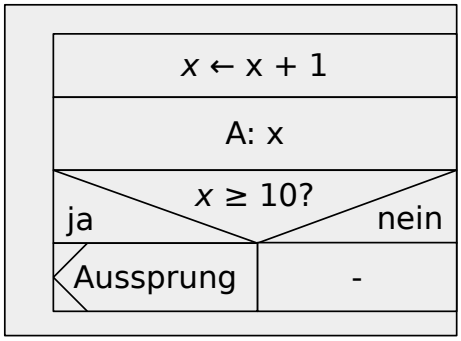


Abb. 47: Struktogramm: Endlosschleife mit Aussprung

Siehe dazu auch Kapitel „Aussprung mit break“ auf Seite 144.

Wenn der Aussprung den einzigen Zweck hat, eine nichtabweisende Schleife zu formen, sollten wir es unbedingt vorziehen, diese im Struktogramm wie in Kapitel 4.2.5 gezeigt darzustellen. Das ist übersichtlicher

1 In den Beispielen auf diesen Seiten ist es ausnahmsweise zusätzlich mit dem Wort „Aussprung“ versehen.

und sauberer. Denken Sie bitte immer daran, dass ein Struktogramm keine grafische Darstellung eines Computerprogramms ist, sondern die grafische Darstellung des darunter liegenden Algorithmus.

4.2.7 Beispiel für ein vollständiges Struktogramm

Das folgende Struktogramm stellt einen Algorithmus dar, mit dem sich in höchstens zehn Versuchen jede von der Anwenderin oder dem Anwender des Algorithmus ausgedachte ganze Zahl zwischen 1 und 1000 ermitteln lässt. Die Rückmeldung erfolgt über einen Tastendruck. Ist die ausgedachte Zahl größer als die vom Algorithmus „geratene“, so soll ein Minuszeichen eingegeben werden, ist sie zu groß, soll ein Pluszeichen eingegeben werden und jede andere Eingabe zeigt an, dass der Algorithmus die Ratezahl erfolgreich herausgefunden hat.

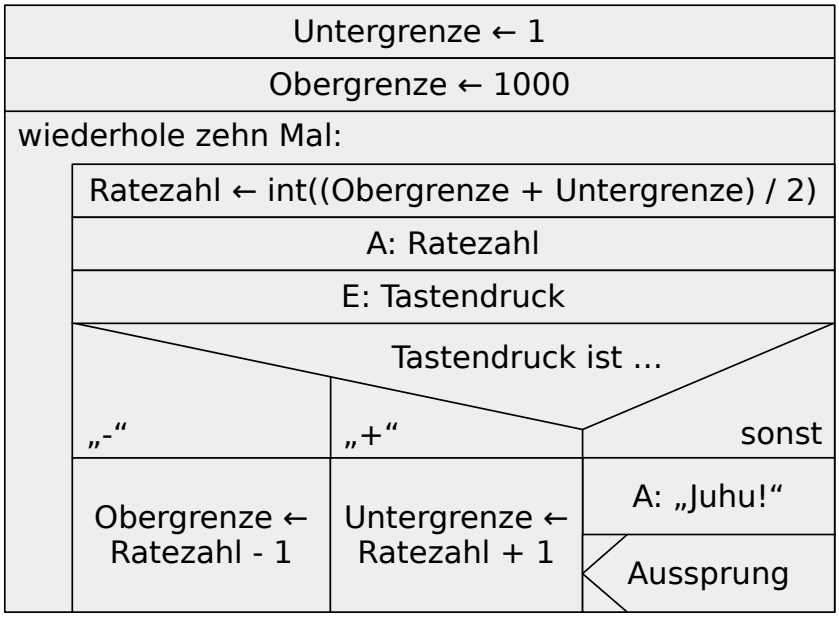


Abb. 48: Struktogrammbeispiel „Zahlenraten“

Der Algorithmus bestimmt dazu ein Suchintervall mit Ober- und Untergrenze. Anfangs liegen diese Werte bei 1 und 1000. Mit jedem Rateversuch wird das Intervall halbiert. Die Funktion **`int()`** sorgt durch Abschneiden der Nachkommastellen dafür, dass das Ergebnis der Halbierung wieder eine ganze Zahl wird. Nach spätestens 10 Halbierungen hat das Suchintervall die Größe 1 und die Zahl ist gefunden.

4.2.8 Struktogramm-Editor

Es gibt diverse Programme zum Zeichnen von Struktogrammen. Sie benötigen keines davon. Ein Stift, ein Lineal und ein Geodreieck oder ein einfaches Zeichenprogramm wie Dia[1] oder LibreOffice Draw reichen völlig aus.

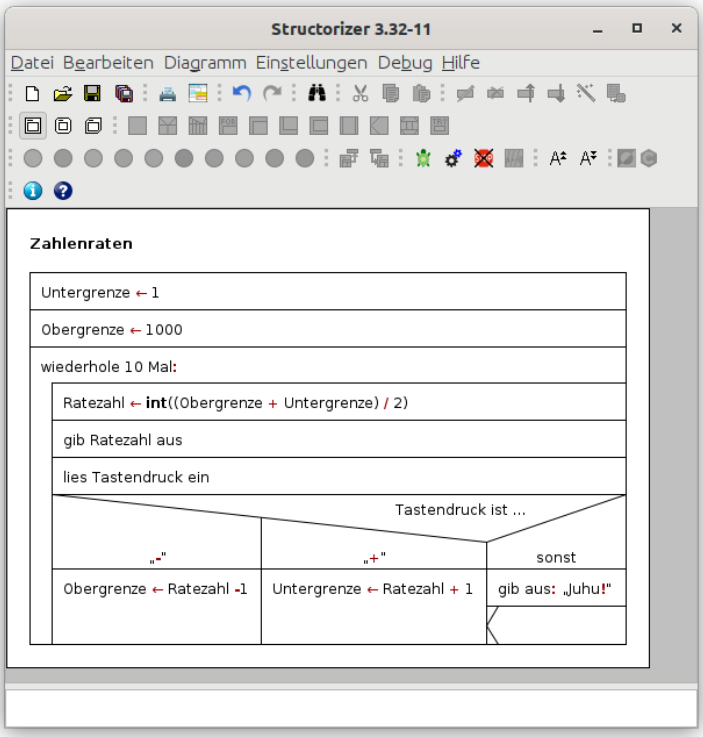


Abb. 49: Der Struktogramm-Editor „Structorizer"

Wenn Sie unbedingt ein spezielles Programm wie Structorizer[2] verwenden möchten, exportieren Sie die erzeugte Grafik bitte in einem Vektorformat wie SVG oder PDF, damit das Ergebnis einigermaßen lesbar wird. Die ganzen Überprüfungs-Einstellungen im Programm können Sie deaktivieren. Sie helfen Ihnen nur, wenn Sie versuchen, in dem Struktogramm-Editor Java-Programme zu schreiben.

1 http://dia-installer.de/
2 https://structorizer.fisch.lu/

5 Python

Zur Umsetzung unserer Algorithmen in von einem Computer ausführbare Programme benötigen wir eine Programmiersprache. Die Auswahl der zur Verfügung stehenden Sprachen ist riesig. Es gibt zahlreiche spezialisierte Programmiersprachen für die verschiedensten Anwendungsgebiete von der kaufmännischen Buchhaltung bis zur künstlichen Intelligenz. Es gibt aber auch Sprachen, die universell für eine Vielzahl von Aufgabenstellungen einsetzbar sind. Manche sind schwer zu erlernen, andere wiederum bieten so gut wie keine Einstiegshürden.

Gelegentlich werden Sprachen nur für ein bestimmtes Betriebssystem oder für eine bestimmte Hardware (man sagt auch: für eine bestimmte Plattform) angeboten, einige Sprachen funktionieren sogar nur innerhalb eines bestimmten Softwarepakets. Es gibt jedoch auch Sprachen, die für eine Vielzahl von Plattformen erhältlich sind. Nicht zuletzt kann man für Programmiersprachen richtig viel Geld bezahlen – oder sie kostenlos herunterladen.

Aus all diesen Überlegungen heraus haben wir uns für eine Sprache entschieden, die 1991 von Guido van Rossum veröffentlicht wurde: Python.

Python ist kostenlos, läuft auf so gut wie allen verbreiteten Betriebssystemen, ist sehr einfach zu erlernen, eignet sich jedoch als Vielzwecksprache für zahlreiche unterschiedliche Anwendungsgebiete. Sie ist objektorientiert und besitzt nur wenige Sprachelemente, kann aber durch Bibliotheken fast beliebig modular erweitert werden.

Abb. 50: Guido van Rossum 2006 (dsearls, CC-BY-SA 2.0)

Außerdem ist Python eine Skriptsprache, deren Programmtexte unmittelbar vom Python-Interpreter ausgeführt werden können. Sprachen wie Java, C oder Fortran benötigen dagegen einen Compiler, welcher den Quelltext erst mehr oder weniger aufwendig in ein ausführbares Programm übersetzt, das dann unter Umständen nur auf der Plattform läuft, auf der es kompiliert wurde.

5.1 Download und Installation

Python ist frei erhältlich. Die Downloadpakete für Windows und macOS finden Sie auf http://www.python.org/download/. Wenn Sie eine Paketverwaltung wie WinGet[1] unter Windows oder Synaptic unter Linux einsetzen, können Sie Python auch einfach damit installieren (Abb. 55).

Für Mobilgeräte gibt es angepasste Entwicklungsumgebungen, beispielsweise Pydroid 3 für das Linux-Betriebssystem Android[2] oder Pythonista 3 für iOS[3]. Leider sind diese üblicherweise nicht kostenlos.

Beachten Sie, dass es in der Vergangenheit zwei unterschiedliche Versionen von Python gab, deren Quelltexte nicht ohne weiteres untereinander austauschbar[4] sind! Die Sprache wurde mit der 2008 erschienenen Version 3 erheblich aufgeräumt. Prüfen Sie vor allem ergoogelte Antworten sorgfältig darauf, dass sich diese nicht auf die völlig veraltete Version 2.7 von Python beziehen! Einige Forenantworten im Internet halten sich verblüffend hartnäckig.

Wir befassen uns in diesem Kurs nur mit der verbesserten Version Python 3. Die Unterversion (3.10 oder 3.11) spielt für uns keine große Rolle, jedoch sollten Windows-Anwenderinnen und -anwender möglichst Versionen von 3.6 an aufwärts installieren, um volle Unicode-Unterstützung zu genießen. Auch die komfortablen F-Strings zur formatierten Zahlenausgabe gibt es erst seit Version 3.6.

Unter Microsoft Windows kann Python sowohl mit als auch ohne Administrationsrechte installiert werden. Aus Gründen der Datensicherheit ist es empfehlenswert, das Installationsprogramm immer mit Rechtsklick „als Administrator“ zu starten und für alle Benutzerinnen und Benutzer des Computers zu installieren. Im Installationsprogramm selber hat es sich bewährt, die Option „Customize Installation“ zu wählen und ein paar

---

1 Im Anhang-Kapitel 7.3 finden Sie eine kurze Anleitung zu WinGet.

2 Pydroid 3 - Educational IDE for Python 3
https://play.google.com/store/apps/details?id=ru.iiec.pydroid3&hl=de

3 Pythonista 3 – https://itunes.apple.com/de/app/pythonista-3/id1085978097?mt=8

4 In der EDV bezeichnet man solche Unverträglichkeit als „Inkompatibilität“.

Häkchen zu setzen, die das nachträgliche Installieren von Modulen erleichtern. Im Zweifelsfall setzen Sie lieber ein Häkchen mehr als eines zu wenig.

Abb. 51: Wählen Sie „Customize installation“

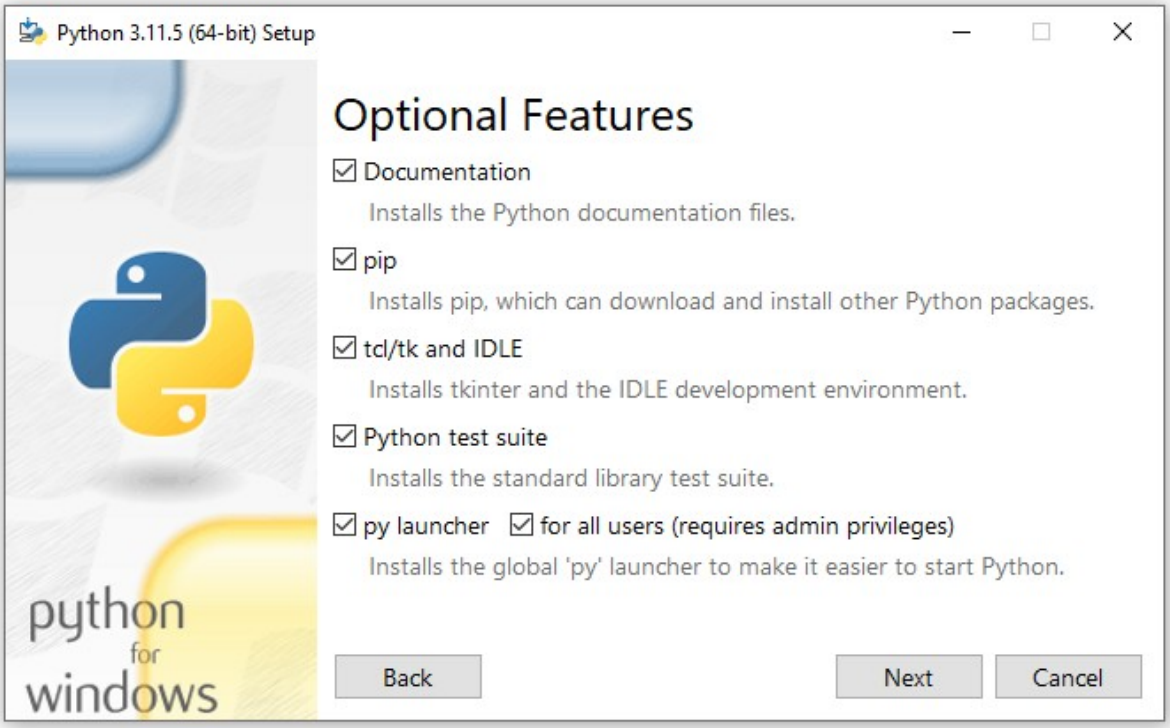


Abb. 52: Setzen Sie ruhig alle Häkchen

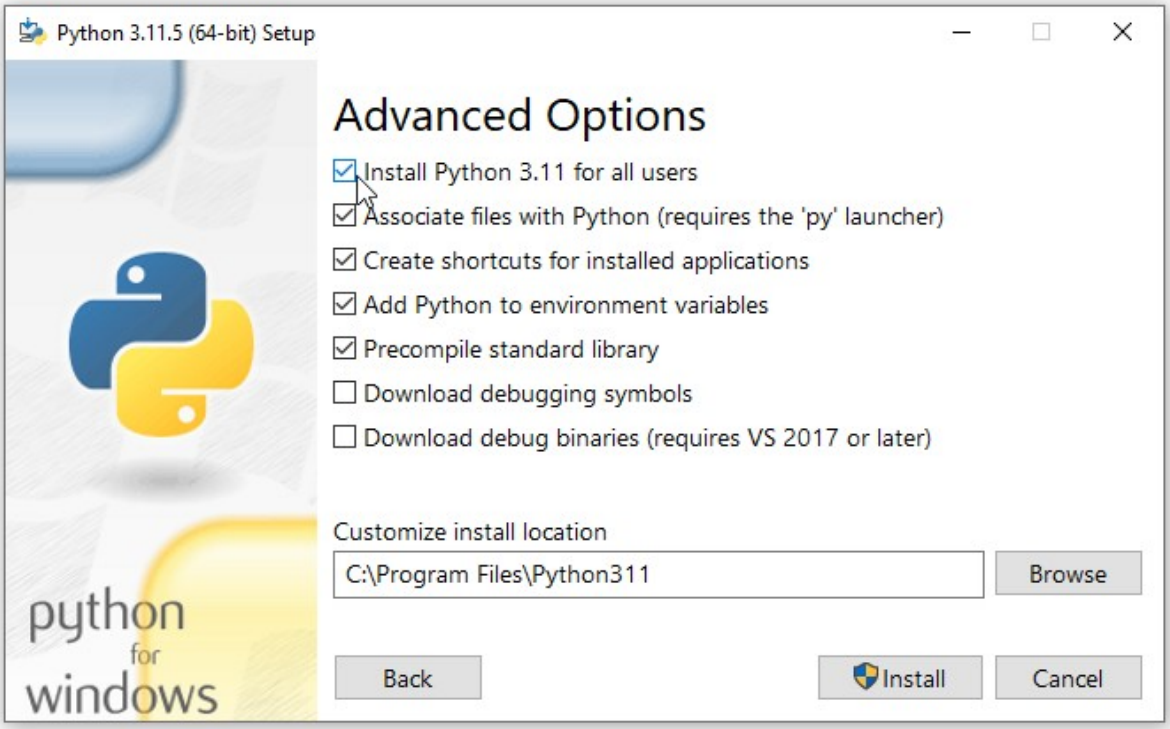


Abb. 53: Installation für alle Benutzerinnen und Benutzer

Nach der Installation können Sie die integrierte Entwicklungsumgebung IDLE unter Windows und Linux über das Startmenü aufrufen.

Unter Windows-Betriebssystemen finden Sie IDLE 3 unter „Start – (Alle) Programme – Python 3.x – IDLE (Python GUI)“ oder Sie drücken die Windowstaste ⊞ und tippen „id…“ (Abb. 54).

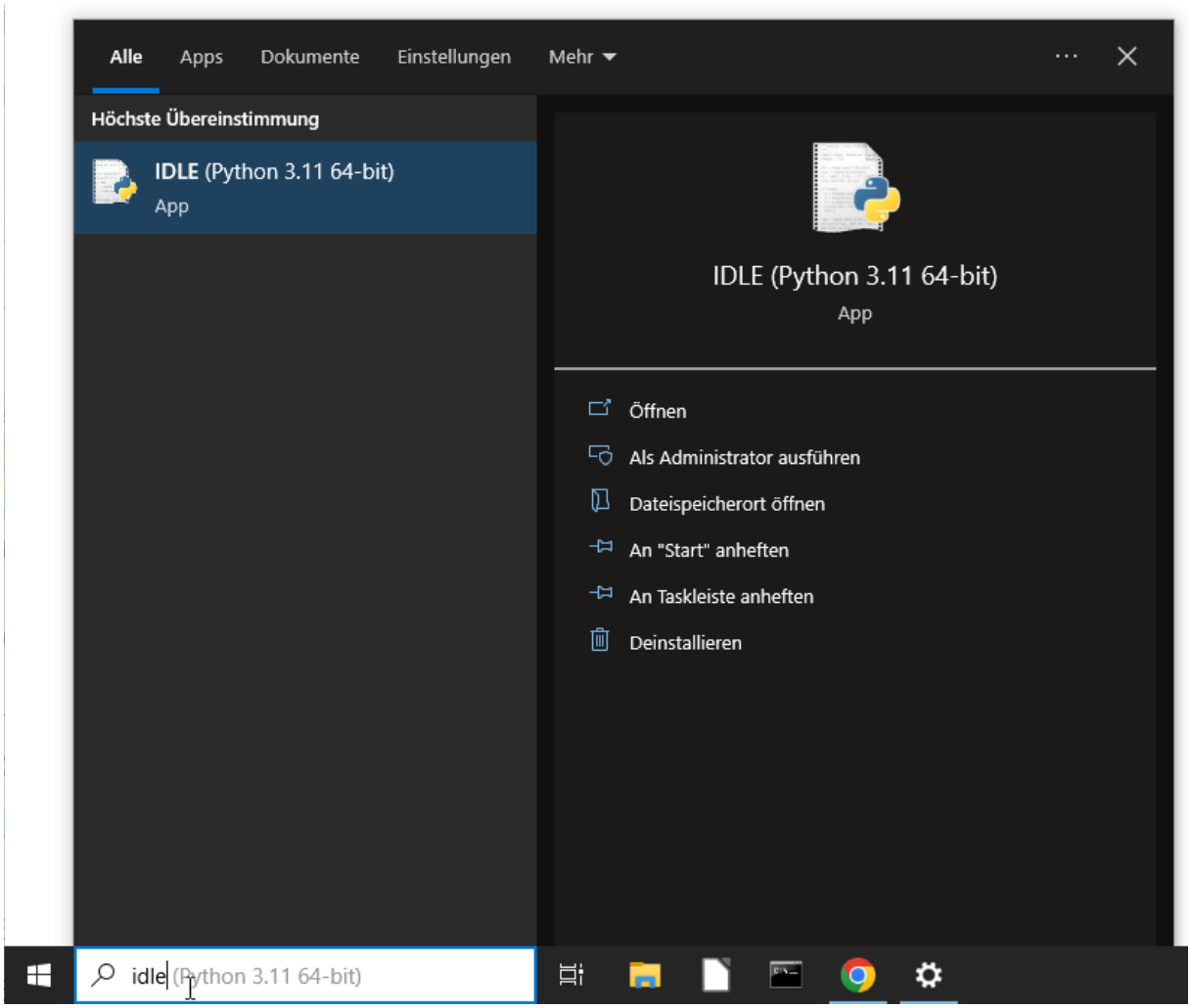


Abb. 54: Das Startmenü von Windows 11

Unter Linux-Desktopumgebungen wie beispielsweise Ubuntu Linux ist die Entwicklungsumgebung unter „Anwendungen – Softwareentwicklung – IDLE 3“ einsortiert … oder Sie drücken die Super-Taste (Windowstaste) und tippen „id…“.

Wenn Sie gerade nicht an Ihrem eigenen Rechner sitzen und daher keine Software installieren können oder dürfen, müssen Sie trotzdem nicht auf das Schreiben und Ausführen von Pythonprogrammen verzichten. Es gibt inzwischen zahlreiche Webseiten, über die Python 3 direkt in einem Webbrowser ausgeführt werden kann[1].

5.1.1 Module für wissenschaftliches Arbeiten

Für den Einsatz im wissenschaftlichen Bereich gibt es einige sehr interessante Zusatzpakete für Python, namentlich die Module für Numerik, symbolische Mathematik und wissenschaftlich anspruchsvolle Diagrammdarstellungen: NumPy, SymPy, SciPy und Matplotlib. Diese lassen sich mit der Paketverwaltung des Betriebssystems (Abb. 55) oder dem Python-Installationsprogramm PIP (Abb. 56) schnell installieren.

1 https://www.python.org/shell/
http://www.pythontutor.com/live.html
https://repl.it/languages/python3
https://www.jdoodle.com/python3-programming-online
https://www.tutorialspoint.com/execute_python3_online.php
https://www.onlinegdb.com/online_python_debugger

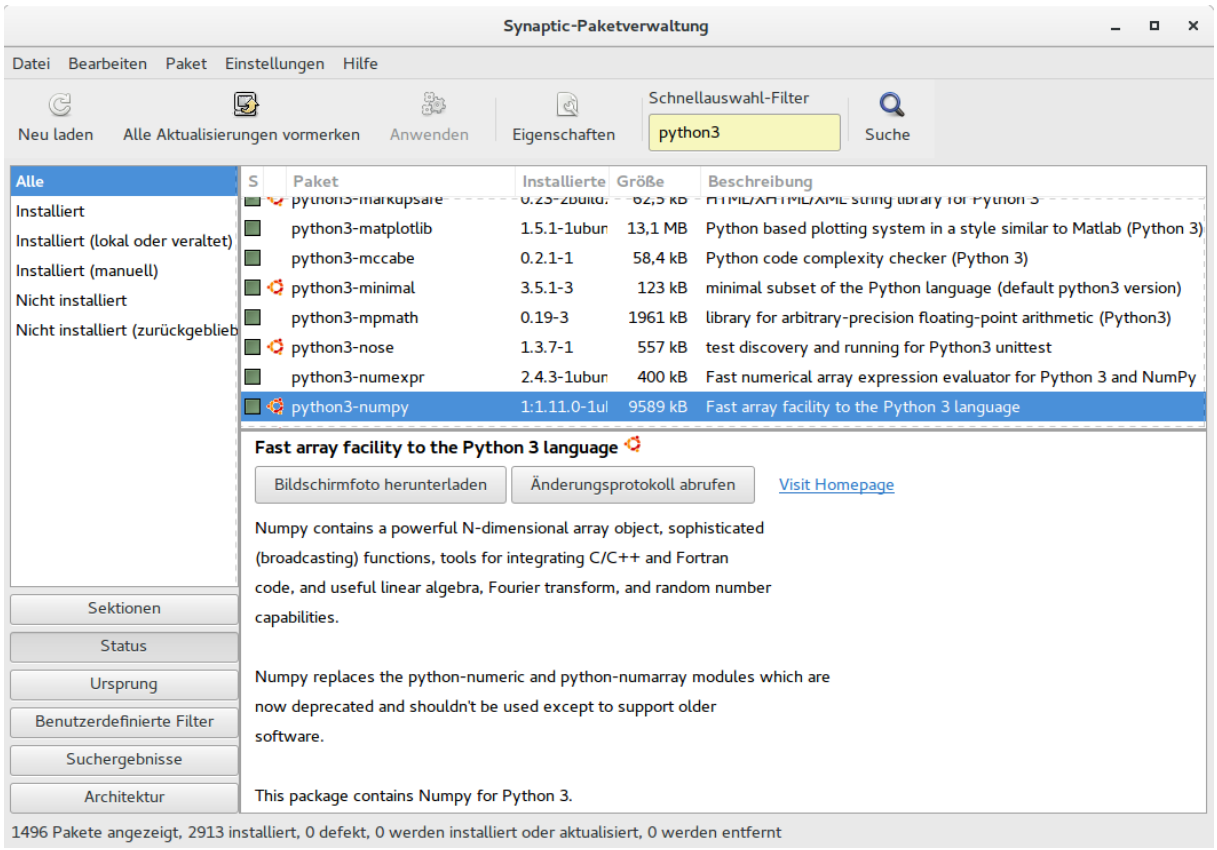


Abb. 55: Paketverwaltung Synaptic in Ubuntu Linux

Unter macOS öffnen Sie ein Terminalfenster und unter Windows die Administrator-Eingabeaufforderung[1] und geben dort „pip3 install *Modulname*“ ein. Wenn sich auf dem Rechner keine alte Python-2-Installation mehr befindet, genügt oft auch „pip install *Modulname*“.

1 Die Administrator-Eingabeaufforderung starten Sie, indem Sie die Windowstaste drücken und „CMD“ tippen. Danach können Sie entweder das Wort „Eingabeaufforderung“ rechtsklicken und „als Administrator ausführen“ wählen oder sie starten das Programm mit etwas Fingerakrobatik über Strg-Umschalten-Eingabetaste.

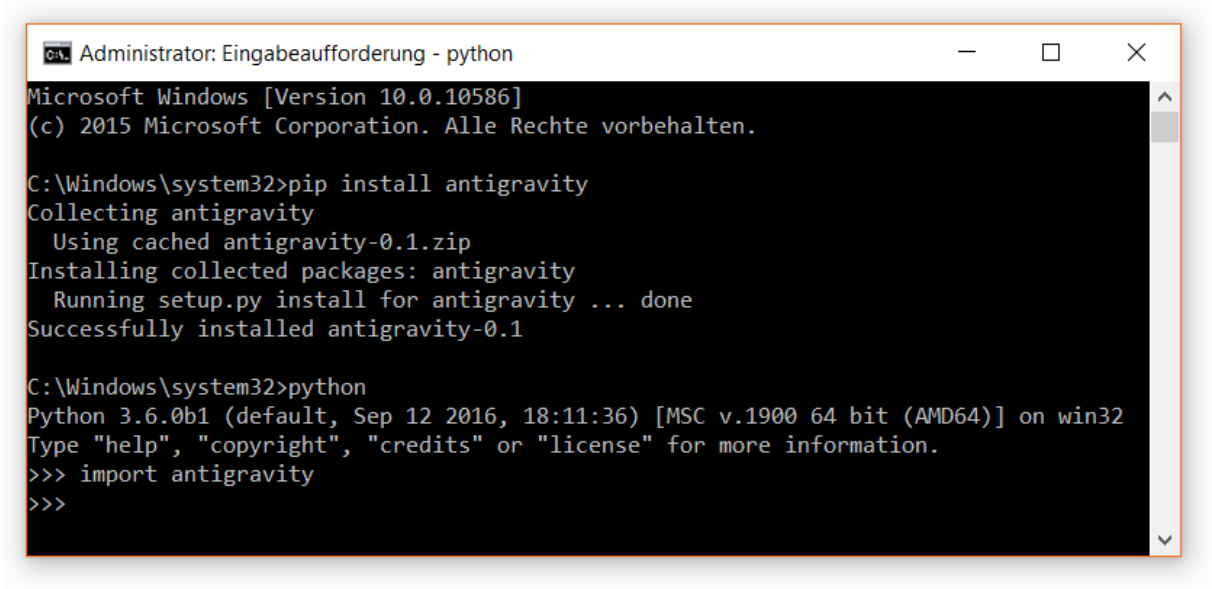


Abb. 56: Paketinstallation mit PIP unter Windows 10

Die importierbaren Module erweitern die Sprache Python um zusätzliche Vokabeln und sind gelegentlich durchaus in anderen Sprachen als Python geschrieben. Die Ausführungsgeschwindigkeit von C ist beispielsweise der des Python-Interpreters oftmals (nicht immer) dramatisch überlegen und für manche numerischen Aufgaben eignet sich das gute alte Fortran besser.

Das Paket Numpy bringt sogar ein Programm namens f2py mit, das Fortran- oder C-Quelltexte zu importierbaren Python-Modulen kompiliert.

Unter Windows kann es vorkommen, dass beim Versuch, PIP über die Eingabeaufforderung zu starten, die Meldung „Der Befehl "PIP" ist entweder falsch geschrieben oder konnte nicht gefunden werden“ erscheint. Dieses Problem entsteht, wenn bei der Python-Installation das Häkchen bei „Add Python to Path“ oder „Add Python to environment variables“ nicht gesetzt wurde, denn dann findet die „Eingabeaufforderung“ von Windows das Programm PIP nicht. Starten Sie in dem Fall die Python-Installation erneut, wählen Sie „Modify“ und setzen Sie das Häkchen.

Wenn Sie auf dem von Ihnen verwendeten Windows-Rechner nicht über Administratorrechte verfügen, können Sie PIP dennoch verwenden. Starten es dazu mit dem zusätzlichen Parameter „--user“ (siehe Abb. 57).

```
C:\Windows\System32\cmd.exe - python
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Windows\system32>pip3 install sympy --user
Collecting sympy
  Downloading https://files.pythonhosted.org/packages/dd/f6/ed485ff22efdd7b371d0dbbf6d77ad61c3b3b7e0815a83c89cbb38ce35de
/sympy-1.3.tar.gz (5.9MB)
    100% |                                | 5.9MB 3.3MB/s
Collecting mpmath>=0.19 (from sympy)
  Downloading https://files.pythonhosted.org/packages/ca/63/3384ebb3b51af9610086b23ea976e6d27d6d97bf140a76a365bd77a3eb32
/mpmath-1.1.0.tar.gz (512kB)
    100% |                                | 522kB 819kB/s
Installing collected packages: mpmath, sympy
  Running setup.py install for mpmath ... done
  Running setup.py install for sympy ... done
Successfully installed mpmath-1.1.0 sympy-1.3

C:\Windows\system32>python
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from sympy import *
>>> x = Symbol("x")
>>> y = x**2 - 5*x + 12
>>> y.diff()
2*x - 5
>>> y * (x**2 + 2*x +1)
(x**2 - 5*x + 12)*(x**2 + 2*x + 1)
>>> solve(x**2 + 3*x - 4)
[-4, 1]
>>>
```

Abb. 57: Paketinstallation ohne Administratorrechte

Falls Sie einmal versuchen sollten, Matplotlib mithilfe von PIP für eine gerade neu herausgekommene Pythonversion zu installieren, und dabei seitenweise rote Fehlermeldungen und gelbe Warnungen erhalten, könnte es daran liegen, dass noch nicht alle benötigten Bibliotheken von ihren Entwicklerinnen und Entwicklern auf die neue Version anpasst wurden. Bleiben Sie dann besser noch eine Weile bei der vorherigen Python-Version. Solche Probleme werden üblicherweise recht bald behoben.

5.1.2 Virtuelle Umgebungen

Da es beim Nachinstallieren mancher neuer oder experimenteller Module (derzeit vor allem im Bereich des maschinellen Lernens) zu Konflikten mit anderen Modulen oder mit Sicherheitsmechanismen des Betriebssystems kommen kann, ist es möglich, eine Pythoninstallation in einer sogenannten virtuellen Umgebung (virtual environment) abzuschotten. Module, die mittels PIP innerhalb einer virtuellen Umgebung installiert werden, sind für die „normale“ Python-Installation außerhalb der virtuellen Umgebung nicht sichtbar.

In diesem Kurs kommen wir noch gut ohne virtuelle Umgebungen aus und gehen daher nicht weiter auf diese Technik ein.

5.2 Erste Schritte in der IDLE-Shell

Die Python-Entwicklungsumgebung IDLE begrüßt uns mit großer Schlichtheit. Ein paar Zeilen Text mit Versionsnummern und drei dunkelrote Größer-als-Zeichen mit einem senkrechten Strich dahinter sind zunächst scheinbar alles, was uns angeboten wird:

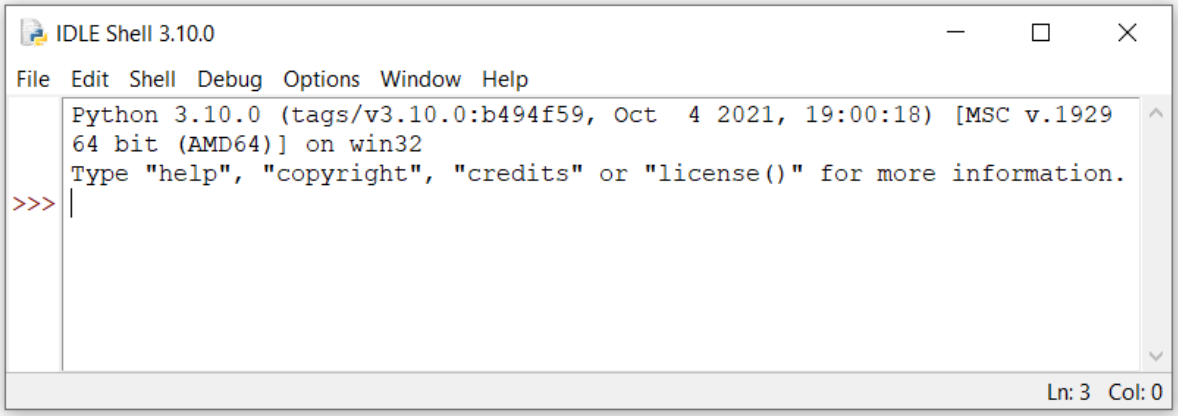


Abb. 58: Die Python-Shell der IDLE unter Windows

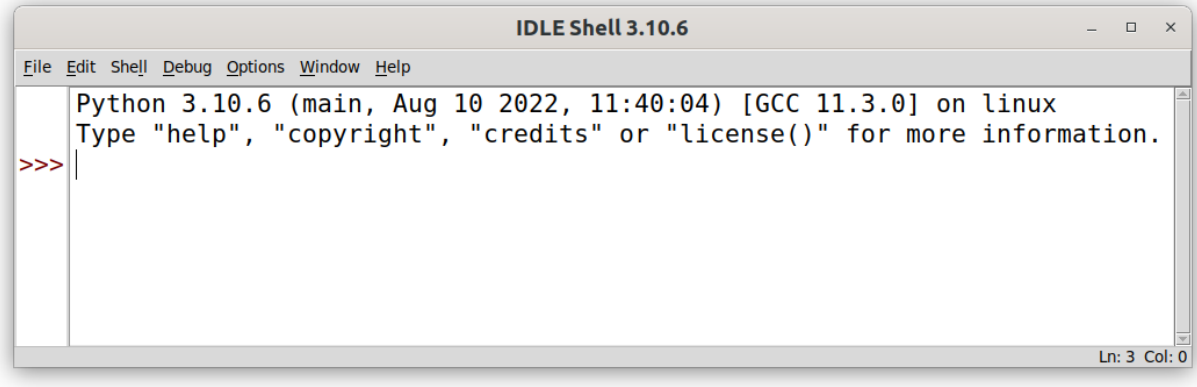


Abb. 59: Die IDLE-Shell unter Linux

Die drei größer-als-Zeichen **>>>** bilden den sogenannten Prompt. Mit ihm gibt der Python-Interpreter zu erkennen, dass er alle Aufgaben bearbeitet hat und nun auf neue menschliche Eingaben wartet. Der blinkende senkrechte Strich dahinter ist der Cursor, der die aktuelle Schreibposition anzeigt.

Ohne einen einzigen Python-Befehl zu kennen, können Sie die Python-Shell jetzt schon als eine Art einfachen Taschenrechner verwenden, der immerhin Klammern kennt und die Priorität von Punktrechnung vor

Strichrechnung beherrscht. Außerdem protokolliert die Shell alle Ein- und Ausgaben.

Tippen Sie einfach mal nacheinander folgende Zeilen ein und drücken Sie am Ende jeder Zeile die Eingabetaste:

```
1 + 1
2 + 3 * 4
(2 + 3) * 4
1 / 7
"warum ist da kein Komma?"
```

Das Resultat sollte ungefähr so aussehen:

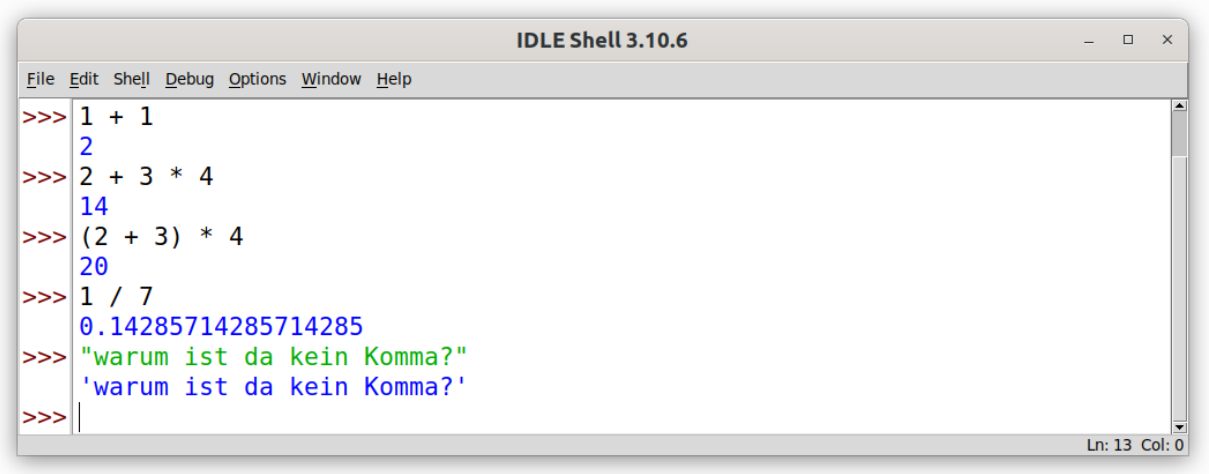


Abb. 60: Die IDLE-Shell als Taschenrechner

Bei der Division fallen zwei Dinge auf: die letzte Stelle des Ergebnisses ist nicht gerundet, sondern abgeschnitten[1] und als Dezimalzeichen wird der Punkt und nicht das Komma verwendet.

Die letzte Eingabezeile zeigt, dass Python auch mit Texten umgehen kann. Sofern Zeichenfolgen mit Anführungszeichen oder Hochkommas umschlossen werden, gibt Python sie unverändert wieder. Fehlen die Anführungszeichen, so versucht Python, die Eingabe als Befehl oder Variablenname zu verstehen.

1 Wer das bei **1/7** nicht erkennt, weil periodische Dezimalbrüche in der Schule nicht behandelt wurden, möge das Experiment mit **2/3** wiederholen.

Nur die unterste Zeile in IDLE nimmt Eingaben entgegen. Ältere, im Protokoll bereits nach oben gerutschte Eingaben können wir nicht nachträglich ändern. Wenn wir einmal auf eine frühere Eingabe zurückgreifen wollen, können wir die Tastenkombinationen Alt-P und Alt-N verwenden (Merkhilfe: P wie *previous* und N wie *next*). Damit lassen sich die bisher vorgenommenen Eingaben durchblättern.

5.3 Fehlermeldungen

Auch wenn Python-Fehlermeldungen auf den ersten Blick vielleicht ein bisschen alarmierend und unverständlich wirken mögen: sie sind eine große Hilfe bei der Fehlersuche in einem Programm. So wird uns immer die Zeile im Quelltext angegeben, an der der Fehler aufgetreten ist und eine (englischsprachige) Beschreibung der Fehlerumstände hilft uns, die Fehlerursache schnell zu erkennen.

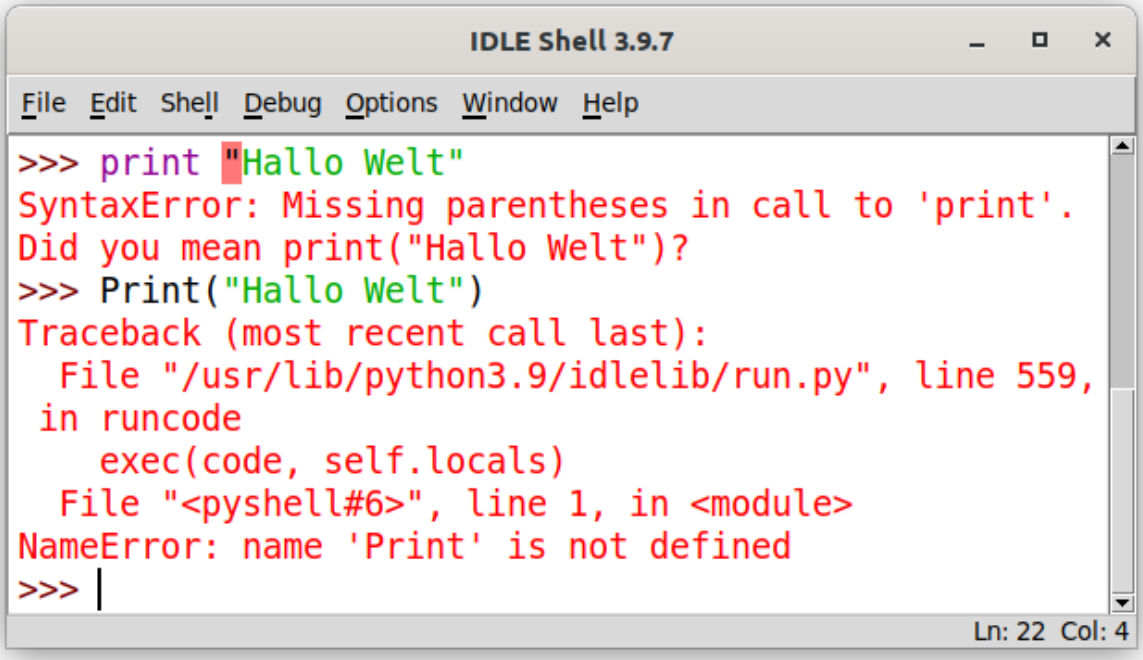


Abb. 61: Python-Fehlermeldungen

Der einfachste und auch häufigste Fehler ist der sogenannte Syntaxfehler; man kann ihn als Grammatikfehler verstehen. Er tritt beispielsweise auf, wenn Klammern fehlen oder ein Wort an einer vom Python-Interpreter nicht erwarteten Stelle steht. Bei komplexeren Fehlern (diese werden auch „Ausnahmen“ genannt) ist die Fehlermeldung länger. Die letzte Zeile der Fehlermeldung gibt dann den Grund für die Ausnahme an und in der zweiten Zeile der Fehlermeldung finden wir die Zeile im Programmtext, an der der Fehler aufgetreten ist. Hier oder in der Programmzeile darüber ist meistens irgendetwas zu reparieren.

Wenn in der Zeile, deren Nummer in der Fehlermeldung angezeigt wird, partout kein Fehler erkennbar ist, besteht die Ursache des Fehlers meistens darin, dass irgendwo weiter oben eine schließende Klammer fehlt.

Anfangs ist es völlig normal, dass ein Programm anstelle der gewünschten Ergebnisse zahlreiche Fehlermeldungen ausgibt. Lassen Sie sich dadurch nicht entmutigen, Fehler gehören zum Programmieren dazu. Nach dem Schreiben eines Programms heißt die nächste Arbeitsphase immer „Fehlersuche“ oder, weil Programmierfehler traditionell als Bugs bezeichnet werden, „Debugging“. In IDLE gibt es dazu sogar einen eigenen Menüpunkt in der Hauptmenüleiste.

Eine Tabelle mit den häufigsten Fehlermeldungen, ihrer Übersetzung und Ratschlägen zur Vermeidung und Behebung befindet sich im Anhang dieses Lehrbuchs auf Seite 324.

5.4 Konstanten

Konstanten sind feste Werte in einem Programmtext. Wir haben sie schon kennengelernt, als wir in der IDLE Zeilen wie **`2 + 3 * 4`** oder **`"warum ist da kein Komma?"`** eingegeben haben.

Der Wert einer Konstanten kann zum Beispiel eine ganze Zahl sein, eine Gleitkommazahl (auch Dezimalzahl oder Fließkommazahl genannt), eine komplexe Zahl, eine Zeichenkette oder ein Wahrheitswert. Wir nennen diese Datentypen auch „Klassen“.

In Anlehnung an ihre englischsprachigen Bezeichnungen werden diese fünf Klassen in Python kurz **`int`**, **`float`**, **`complex`**, **`str`** und **`bool`** genannt.

Deutsche Bezeichnung	Englische Bezeichnung	Python-Klasse	Beispiel für eine Konstante dieser Klasse
Ganzzahl	integer	`int`	`42`
Gleitkommazahl	floating-point number	`float`	`3.141592653589793`
Komplexe Zahl	complex number	`complex`	`(1+1.4142135623730951j)`
Zeichenkette	character string	`str`	`"Hallo Bochum!"`
Wahrheitswert	Boolean value	`bool`	`True`

Eine Zeichenkettenkonstante muss von Anführungszeichen umschlossen werden, um nicht mit dem Namen einer Variable, einer Funktion oder eines anderen Objektes verwechselt zu werden.

Bei der Ein- und Ausgabe von Gleitkommazahlen ist darauf zu achten, dass in Python als Dezimalzeichen ein Punkt verwendet wird. Dies ist in fast allen anderen Programmiersprachen so und auch in den im Fach CAD in nächsten Semester verwendeten Programmen AutoCAD und BricsCAD. Das Komma hat als Trennzeichen eine eigene Bedeutung.

5.5 Variablen

In Programmiersprachen lassen sich Variablen als eine Art benannte Behälter verwenden, in denen Werte, wie beispielsweise Zahlen oder Zeichenketten, zur späteren Verwendung aufbewahrt werden können.

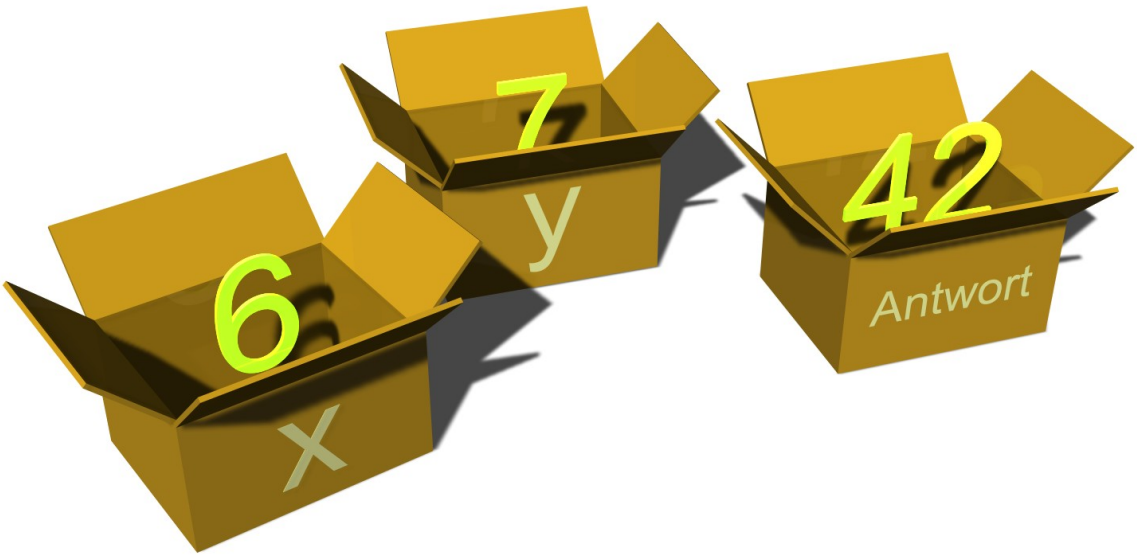


Abb. 62: Variablenmodell „beschriftete Kästchen“

Die Zuweisung eines Wertes zu einer Variable geschieht mit dem Zuweisungszeichen „**=**“ in der Form „**Variablenname = Wert**“. Da die Zuweisung ausschließlich von rechts nach links erfolgt, ist es eine gute Idee, dieses Zuweisungszeichen als „wird zu“ und nicht als „ist gleich“ zu lesen.

Ein anderer wichtiger Unterschied zur Mathematik: Jeder Variable kann beliebig oft ein neuer Wert zugewiesen werden, ohne dass dadurch andere Variablen beeinflusst werden.

Python, wie viele andere Programmiersprachen auch, merkt sich nicht, wie der Wert entstanden ist, der einer Variable zugewiesen wird. Angenommen, wir weisen den beiden Variablen **x** und **y** die Werte **6** und **7** zu. Eine dritte Variable heiße **Antwort** und erhalte das Ergebnis der Berechnung **x * y**. Der in der Variable **Antwort** gespeicherte Wert ist nun **42**. Ändern wir anschließend den Wert von **x** oder **y**, so hat das auf den Wert von **Antwort** keinerlei Auswirkungen mehr.

```
>>> x = 6
>>> y = 7
>>> Antwort = x * y

>>> x
6
>>> y
7
>>> Antwort
42

>>> x = 1
>>> y = 2

>>> x
1
>>> y
2
>>> Antwort
42
```

Eine Besonderheit von Python ist es, mehreren Variablen gleichzeitig denselben Wert zuweisen zu können.

```
>>> a = b = c = 22.5
>>> print(a, b, c)

22.5 22.5 22.5
```

Wir können auch mehreren Variablen gleichzeitig mehrere Werte zuweisen.

```
>>> a, b, c = 1, 2, "halb drei"
>>> a
1
>>> b
2
```

```
>>> c
'halb drei'
```

In Python können wir sogar die Werte mehrerer Variablen in einer einzigen Zeile gegeneinander austauschen.

```
>>> x = "X"
>>> u = "U"

>>> x, u = u, x

>>> x
'U'
>>> u
'X'
```

5.5.1 Variablennamen

Variablennamen dürfen aus einer ununterbrochenen Folge von Buchstaben, Ziffern und Unterstrichen bestehen.

Das erste Zeichen darf keine Ziffer sein.

Im Gegensatz zu vielen anderen Programmiersprachen erlaubt Python auch Buchstaben, die nicht im klassischen 26-Zeichen-Alphabet zu finden sind.

Gültige Variablennamen sind beispielsweise:

```
x
x_min
_
变数名
schalke05
öffnungsmaß
KamelSchreibweise
ich_habe_einen_langen_namen_und_ich_werde_ihn_benutzen
```

Der unscheinbarste Variablenname „_“ (Unterstrich) hat innerhalb der Entwicklungsumgebung IDLE eine besondere Stellung. Er enthält dort die jeweils letzte Ausgabe des Python-Interpreters.

```
>>> 3 * 4
12
>>> _ + 1
13
```

Der Unterstrich ist das einzige in einem Variablennamen erlaubte Zeichen, das kein Buchstabe und keine Ziffer ist. Operatoren wie „+“ oder „-“, Emojis sowie das Leerzeichen „ “ können nicht Bestandteil eines Variablennamens sein.

Manche Variablennamen sehen aus, als enthielten sie einen Punkt. Dieser Punkt hat jedoch eine ganz besondere Bedeutung. Verwenden Sie ihn nicht für Variablennamen!

```
>>> class x:
        min = 0
        max = 99

>>> x.min
0
```

```
>>> x.max
99
```

In Variablennamen sollten die Zeichen I (großes i) und l (kleines L) sowie 0 (Ziffer null) und O (großes o) vermieden werden, um Verwechslungen zu vermeiden.

Es gibt Schriftarten wie die in diesem Text für Quelltexte verwendete Dejavu Sans Mono[1], welche sich sehr um eine Unterscheidbarkeit dieser Zeichen bemühen.

```
>>> I = 0
>>> l = 1 + I
>>> lO = 10
>>> I0 = 1 + lO
```

Andere Schriftarten, wie Microsofts Arial, machen das Lesen von Quelltexten zum Ratespiel.

```
>>> I = 0
>>> l = 1 + I
>>> lO = 10
>>> I0 = 1 + lO
```

Den Inhalt (Wert) einer Variable finden wir heraus, indem wir den Variablennamen mit korrekter Groß- und Kleinschreibung am Prompt eingeben[2] oder indem wir die Funktion **`print()`** zur Ausgabe einer oder mehrerer Variablen verwenden.

```
>>> print(I, l, lO, I0)
0 1 10 11
```

1 Die Deja-Vu-Fontfamilie steht unter einer freien Lizenz und kann kostenlos von https://dejavu-fonts.github.io/ heruntergeladen werden.

2 „Eingeben“ heißt, etwas zu tippen und dann die Eingabetaste ↵ (Enter-Taste) zu drücken.

5.6 Rechenoperationen

Python unterstützt die Grundrechenarten und beachtet dabei die aus der Mathematik bekannte Ausführungsreihenfolge: Punktrechnung (**·** und **:**) geht vor Strichrechnung (**–** und **+**), eine noch höhere Priorität haben Potenzen, und Klammern stehen über allem.

Alle Operatorsymbole folgen der im PC-Bereich üblichen Schreibweise, die sich daraus ableitet, dass zur Eingabe in der Regel eine klassische Schreibmaschinentastatur verwendet wird. Daher werden anstelle des Multiplikationspunktes **·** der Stern ***** und anstelle des Divisionsdoppelpunktes **:** der Schrägstrich **/** verwendet. Trotz der neuen Operatoren gelten Multiplikation und Division weiterhin als Punktrechnung!

Der Multiplikationsoperator darf nicht entfallen, wenn Variablen mit Zahlenwerten multipliziert werden. Anstelle von **3x** muss es daher beispielsweise stets **3 * x** heißen.

Potenziert wird nicht durch Hochstellen, sondern mit einem Doppelsternchen **. Die Kubikzahl 2^3 wird also in Python als **2 ** 3** geschrieben[2].

2 Die Verwendung des Doppelsternchens als Potenzierungsoperator wurde schon 1954 von John Backus für die Sprache Fortran festgelegt. Obwohl dieser Operator auch in viele andere Sprachen, wie zum Beispiel COBOL, übernommen wurde, verwendeten John Kemeny und Thomas Kurtz stattdessen in ihrer 1964 vorgestellten Sprache BASIC als Potenzierungsoperator das Zirkumflex ^. Dessenungeachtet führte Dennis Ritchie ebendieses Zirkumflex in der 1972 von ihm verfassten Sprache C als Exklusiv-oder-Operator ein, was später von James Gosling für Java und Guido van Rossum für Python übernommen wurde (Quelle: https://softwareengineering.stackexchange.com/questions/331388/why-was-the-caret-used-for-xor-instead-of-exponentiation/331392#331392).

Für ganzzahlige Divisionen mit Rest (zum Beispiel „14 durch 4 ergibt 3, Rest 2“) verwendet Python die Operatoren `//` und `%`.

```
>>> 14 / 4
3.5
>>> 14 // 4
3
>>> 14 % 4
2
```

Dabei erhalten wir mit dem Operator `//` den ganzzahligen Quotienten zweier Zahlen und mit `%` den dazu gehörenden Divisionsrest. Weil der Divisionsrest in der Mathematik auch Modulo heißt, wird das Prozentzeichen `%` in diesem Zusammenhang Modulo-Operator genannt.

5.7 Funktionen und Module

Aus der Mathematik kennen wir bereits Funktionen wie sin(x) oder f(x,y), die zu einem oder mehreren Eingangswerten einen bestimmten Funktionswert zurückgeben.

Im einfachsten Fall nimmt eine Funktion eine beliebige Anzahl von Eingangswerten an und gibt ein davon abhängiges Ergebnis zurück.

Abb. 63: Funktion mit Eingangswerten und Rückgabewert

Außer diesen „reinen“ Funktionen gibt es auch Funktionen, die eine Wirkung haben. Sie steuern beispielsweise ein Gerät oder geben etwas auf dem Bildschirm aus.

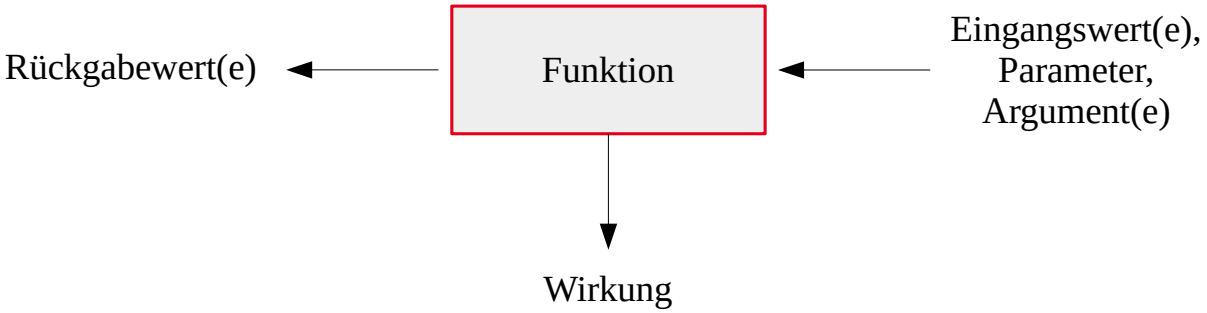


Abb. 64: Funktion mit Wirkung

Die Wirkung einer Funktion wird manchmal auch „Nebenwirkung“ oder, als Fehlübersetzung des englischen „side effect“, „Seiteneffekt“ oder sogar „Nebeneffekt“ genannt. Innerhalb eines Programms kann man die im Sinne sauberer Programmierung verpönte Manipulation von außerhalb der Funktion eingeführten Variablen mithilfe des Schlüsselworts **`global`** so bezeichnen.

Eine Funktion ohne definierten Rückgabewert gibt ein besonderes Objekt zurück: **`None`**. Dadurch ist sichergestellt, dass wir keine Fehlermeldung erhalten, wenn wir einer Variablen den Rückgabewert einer Funktion zuweisen.

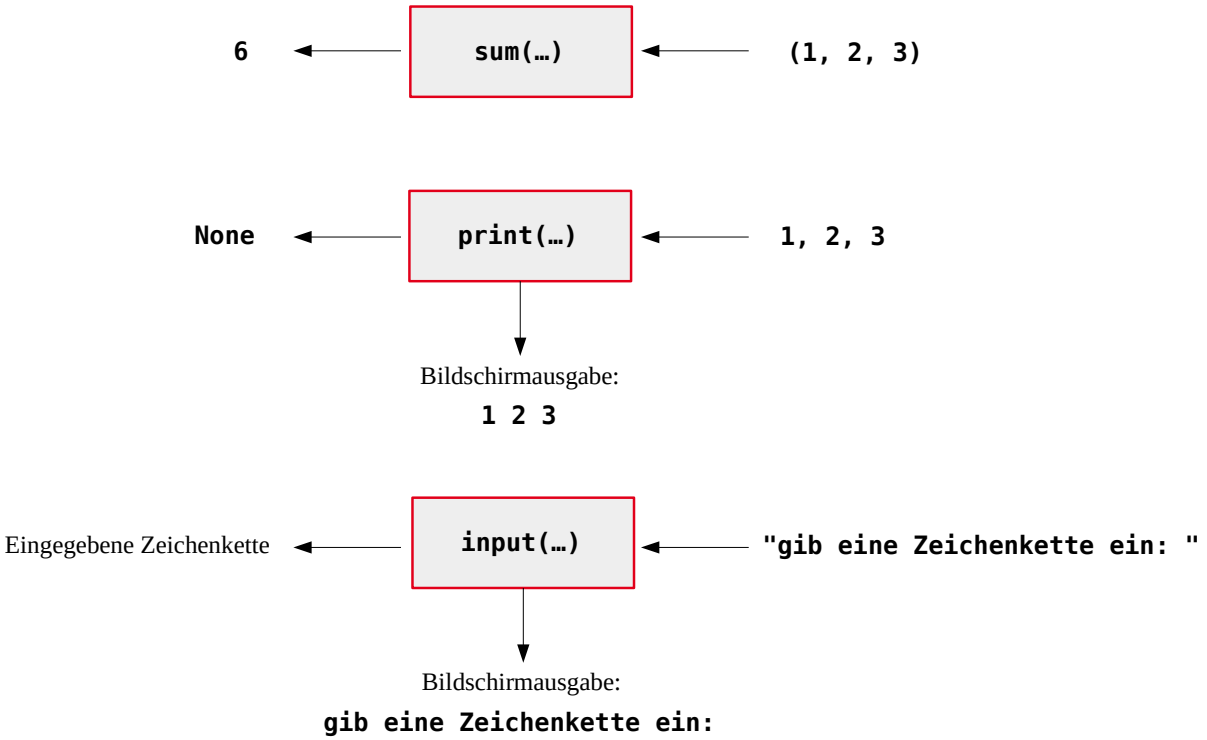


Abb. 65: Funktionen mit und ohne Wirkung oder Rückgabewert

Beispiele für eingebaute Funktionen	
Absolutwert einer Zahl	`abs(-4)`
kleinster bzw. größter Wert einer Anzahl von Elementen	`min(a, b, c, …)` `max(a, b, c, …)`
Länge eines iterierbaren Objekts, beispielsweise einer Zeichenkette	`len("Text")`
Summe der Zahlenwerte eines iterierbaren Objekts, beispielsweise einer Liste	`sum([1, 2, 3])`
Eine neue sortierte Liste aus einem iterierbaren Objekt erzeugen	`sorted([3, 1, 2])`
Zahl a auf n Nachkommastellen runden	`round(a, n)`
Ausgabe von Werten auf dem Bildschirm	`print("Text", 123)`
Eingabe einer Zeichenkette	`input("Text: ")`
Hilfe (englisch)	`help("sum")` `help("builtins")`

Python verfügt im Verhältnis zu anderen Programmiersprachen über relativ wenige eingebaute Funktionen. Bei der Installation werden jedoch zahlreiche Module mitgeliefert, die tausende von Funktionen für alle möglichen Einsatzgebiete zur Verfügung stellen. Um diese Funktionen zu nutzen, müssen wir sie lediglich importieren.

5.7.1 Funktionsweiser Import

Aus einem Modul können wir entweder alle darin vorhandenen Funktionen auf einen Schlag importieren ...

```
from Modulname import *
```

... oder wir beschränken uns beim Import auf einzelne ausgewählte Funktionen.

Das ist zwar etwas mehr Tipparbeit, aber sehr sinnvoll. Denn wenn mehrere Module die gleichen Funktionsnamen verwenden, kann es sonst passieren, dass die zuletzt importierten Module unkontrolliert bereits vorhandene Funktionen überschreiben.

```
from Modulname import Funktionsname
```

Die importierten Funktionen lassen sich direkt mit ihrem Namen ansprechen:

```
>>> from math import cos
>>> cos(0)
1.0
```

Falls uns der Name einer zu importierenden Funktion unpassend erscheint, weil er beispielsweise unbequem lang oder der deutsche Name angebrachter ist, können wir auch einen eigenen Namen für die Funktion vergeben.

Das Modul **math** besitzt beispielsweise eine Funktion **gcd(x, y)**, die den *greatest common denominator*, also den größten gemeinsamen Teiler zweier ganzer Zahlen x und y zurückgibt. In der Schule haben wir diese Funktion als „ggT“ kennengelernt. Wir können die Funktion nun so importieren, dass sie in unserem Programm nicht **gcd**, sondern **ggt** heißt:

```
from math import gcd as ggt
```

5.7.2 Modulweiser Import

Es ist auch möglich, alle Funktionen eines Moduls so zu importieren, dass es dabei nicht zu Namenskonflikten kommt. Dazu verwenden wir die folgende Schreibweise:

```
import Modulname
```

Nachdem ein Modul auf diese Art importiert wurde, können wir alle Funktionen des Moduls ansprechen, indem wir ihrem Funktionsnamen den jeweiligen Modulnamen voranstellen:

```
Modulname.Funktionsname()
```

Beispiel:

```
>>> import math
>>> math.cos(0)
1.0
```

Das sorgt in den meisten Fällen für einen sehr lesbaren Quelltext, weil immer klar ist, aus welchem Modul eine Funktion stammt.

Eine Liste aller importierbaren Module können wir uns durch den Aufruf **help("modules")** anzeigen lassen. Alternativ können wir auch in IDLE nach dem Tippen von **import** die Tabulatortaste ⇆ drücken.

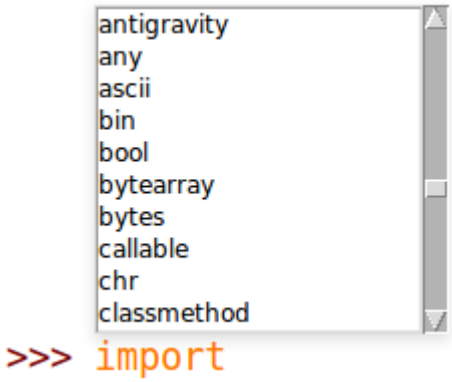


Abb. 66: Modulliste nach Drücken der Tabulatortaste

5.7.3 Das Mathematik-Modul: math

Weil im Bau- und Umweltingenieurwesen wohl kaum ein Berechnungsprogramm ohne Mathematikfunktionen auskommt, schauen wir uns dieses Modul noch etwas genauer an.

Eine vollständige Übersicht über den Umfang des Mathematikmoduls liefert uns die Funktion **`help()`**. Leider ist sie nur für diejenigen richtig hilfreich, die einigermaßen gut Englisch können.

```
import math
help(math)
```

Für alle anderen erklärt die folgende Tabelle deshalb einige ausgewählte Funktionen des Moduls „math“:

Funktion	Beschreibung
`floor(x)`	Rundet **`x`** ab.
`ceil(x)`	Rundet **`x`** auf.
`trunc(x)`	Schneidet die Nachkommastellen von **`x`** ab. Die Funktion verhält sich also bei positiven Zahlen wie **`floor(x)`** und bei negativen Zahlen wie **`ceil(x)`**.
`gcd(x, y)`	ggT der beiden ganzen Zahlen **`x`** und **`y`**.

Funktion	Beschreibung
sin(a) **cos(a)** **tan(a)**	Die Winkelfunktionen Sinus, Kosinus und Tangens. Der Winkel ist stets in Bogenmaß anzugeben!
asin(x) **acos(x)**	Die Umkehrfunktionen von Sinus und Kosinus (in der Mathematik oft als $\sin^{-1}()$ und $\cos^{-1}()$ geschrieben) ermitteln zu einer Zahl zwischen -1 und 1 den dazugehörigen Winkel in Bogenmaß.
atan(s) **atan2(x, y)**	Die Tangensfunktion hat gleich zwei Umkehrfunktionen: **atan()** nimmt die Steigung als einzelne Zahl entgegen und gibt einen Winkel zwischen -π/2 und π/2 zurück. **atan2()** benötigt zwei Zahlenwerte (z. B. Gegenkathete und Ankathete). Da diese Werte unterschiedliche Vorzeichen haben dürfen, lassen sich alle Winkel von -π bis π ermitteln.
radians(w)	Rechnet einen Winkel von Altgrad in Bogenmaß um.
degrees(a)	Rechnet einen Winkel von Bogenmaß in Altgrad um.
sqrt(x)	Quadratwurzel (square root). Kann auch ohne Verwendung des math-Moduls als **x**0.5** geschrieben werden. Bei negativen Werten von **x** wirft die Funktion **sqrt(x)** eine Fehlermeldung vom Typ „ValueError“ aus, wogegen **x**0.5** eine komplexe Zahl zurückgibt.
exp(x)	Exponentialfunktion e^x
pow(a, b)	Potenzfunktion a^b – identisch mit **a**b**
log(x)	Natürlicher Logarithmus (Basis e)
log(x, b)	Logarithmus zur Basis b
log10(x) **log(x, 10)**	Dekadischer Logarithmus (Basis 10)

5.7.4 Funktionszuweisungen

In Python ist es möglich, Funktionen genauso wie Zahlen oder Zeichenketten einer Variable zuzuweisen. Die Funktion kann dann unter dem Namen der Variable ausgeführt werden.

```
>>> a = 44801
>>> b = "Bochum"
>>> c = print

>>> c(a, b)
44801 Bochum
```

Umgekehrt ist es auch möglich (wenn auch fast immer unsinnig), die Namen vorhandener Funktionen wie **`print`** oder **`input`** als Variablennamen für Zahlenwerte oder Zeichenketten einzusetzen.

Bei der Zuweisung dürfen keine Klammern hinter dem Funktionsnamen stehen, sonst wird die Funktion sofort ausgeführt und anstelle der Funktion nur ihr Rückgabewert der Variable links vom Gleichheitszeichen zugewiesen.

5.8 Eingabe mit input()

Unsere Programme sollen nicht immer mit denselben Werten rechnen, sondern auch menschliche Eingaben entgegennehmen. Das können Mausaktionen sein, zum Beispiel das Anklicken oder Verschieben von Grafikelementen, aber auch im einfachsten Fall eine direkte Tastatureingabe.

Um in einem Programm eine Zeichenfolge von der Tastatur einzulesen, gibt es die Funktion **input()**. Ihren Rückgabewert weisen wir üblicherweise einer Zeichenkettenvariable zu.

```
>>> a = input()
Hallo!

>>> b = input("Wie heißt Du? ")
Wie heißt Du? Maggy Mustermann

>>> a
'Hallo!'

>>> b
'Maggy Mustermann'
```

Die Klammer nach einem Funktionsnamen ist zur Ausführung der Funktion unbedingt notwendig. Sie darf leer sein, kann aber im Falle von **input** auch einen in Anführungszeichen gesetzten Fragetext enthalten, um dem Bediener oder der Bedienerin mitzuteilen, welche Eingabe erwartet wird.

5.8.1 Lesen aus Textdateien

Viel häufiger als über die Tastatur erhalten Programme in der Ingenieurpraxis ihre Eingaben aus Dateien, die beispielsweise aus Messanlagen stammen oder von anderen Programmen erzeugt wurden.

Um mit Python den Inhalt einer Datei zu lesen, melden wir beim Betriebssystem an, dass wir die Datei zum Lesen öffnen wollen (Modus „**r**“ wie „*read*“). Wir erhalten dann ein Dateiobjekt, das sogenannte Dateihandle. Dieses Dateihandle, nennen wir es der Einfachheit halber „**meine_datei**“, besitzt die Methode **read()**, mit der wir den gesamten Inhalt der Datei als eine einzige lange Zeichenkette lesen.

Die folgenden Zeilen öffnen die Datei „**Liste.txt**“, lesen deren Inhalt in eine Variable „**Inhalt**“ und geben diesen Inhalt auf dem Bildschirm aus.

```
with open("Liste.txt", "r") as meine_datei:
    Inhalt = meine_datei.read()

print("In der Datei „Liste.txt“ steht folgendes:")
print(Inhalt)
```

Da das Dateihandle ein iterierbares Objekt darstellt, können wir auch mit Schleifen auf die Dateiinhalte zugreifen. Eine Vertiefung dieses Themas finden Sie in Kapitel 5.22.1.

5.9 Ausgabe mit print()

Die Funktion **print()** gibt die ihr übergebenen Inhalte auf dem Bildschirm aus und beginnt danach eine neue Zeile.

```
>>> a = 3
>>> b = 4

>>> print("Die Variable a hat den Wert", a)
Die Variable a hat den Wert 3

>>> print("Die Variable b hat den Wert", b)
Die Variable b hat den Wert 4

>>> print("Die Summe beider Zahlen ist", a+b)
Die Summe beider Zahlen ist 7
```

Zwischen den Klammern dürfen beliebig viele durch Kommas getrennte Funktionsparameter stehen. Wir bezeichnen diese Parameter auch als die Argumente der Print-Funktion. Sie werden von ihr, voneinander jeweils durch ein Leerzeichen getrennt, hintereinander ausgegeben.

Lassen wir die Klammer leer, so gibt **print()** nur eine Leerzeile aus.

5.9.1 Ausgabe in Textdateien

Mit der Print-Funktion können wir nicht nur Texte auf dem Bildschirm ausgeben; sie ist auch in der Lage, Texte in eine Datei zu schreiben. Wir denken uns dazu einen Namen für die neue Datei aus (zum Beispiel „**Liste.txt**“), öffnen die Datei zum Schreiben (Modus „**w**“ wie „write“) und erhalten dabei, genau wie vorhin beim Lesen, ein Dateihandle. Nennen wir es auch hier wieder „**meine_datei**“. Jeder Aufruf der Print-Funktion, bei dem wir dieses Dateihandle angeben, schreibt nicht auf den Bildschirm, sondern in die Datei.

```
with open("Liste.txt", "w") as meine_datei:
    print("Dies ist eine Textdatei.", file=meine_datei)
    print("Sie ist zum Schreiben geöffnet.",
          file=meine_datei)
    print("Innerhalb der with-Einrückung kann ich "
          "beliebig oft in die Datei schreiben.",
          file=meine_datei)

print("Beim Verlassen der Einrückung wird sie geschlossen.")
```

Warnung!

Das Schreiben in eine Datei ist eine der wenigen Gelegenheiten, Schaden auf einem Rechner anzurichten. Eine existierende Datei, die im Modus „**w**“ erneut zum Schreiben geöffnet wird, verliert augenblicklich und ohne jede Rückfrage ihren gesamten Inhalt. Seien Sie aufmerksam beim Überschreiben wichtiger Daten! Machen Sie außerdem regelmäßig Backups!

5.9.2 Alternatives Trennzeichen: sep

Die Print-Funktion trennt alle auszugebenden Elemente standardmäßig mit einem Leerzeichen.

```
>>> a = 3
>>> b = 4
>>> c = 5

>>> print(a, b, c)
3 4 5
```

Anstelle des Leerzeichens können wir auch beliebige andere Zeichen oder Zeichenfolgen als Trennzeichen zwischen den mit **print()** auszugebenden Parametern verwenden.

Diese Trennzeichen nennt man auch Separatoren. Sie werden über den zusätzlichen Parameter **sep** an die Funktion übergeben.

Um beispielsweise die Inhalte der drei Variablen **a**, **b** und **c** mit jeweils einem Semikolon getrennt hintereinander auszugeben, schreiben wir:

```
>>> print(a, b, c, sep = ";")
3;4;5
```

Wird als Separator die leere Zeichenkette **""** eingestellt, so gibt Python die Werte der Parameter direkt hintereinander aus.

```
>>> print(a, b, c, sep = "")
345
```

5.9.3 Alternatives Zeilenende: end

Wenn es uns stört, dass die Print-Funktion nach der Ausgabe eine neue Zeile beginnt, oder wenn wir am Ende einer Zeile ein besonderes Zeichen sehen wollen, so können wir dieses Verhalten mit dem Parameter **end** beeinflussen.

Der Standardinhalt von **end** ist der Zeilenwechsel **"\n"**, es ist aber auch jede andere Zeichenfolge möglich.

Soll Python beispielsweise nach einer Print-Ausgabe ohne Zeilenwechsel weiterschreiben, so veranlassen wir das, indem wir **end=""** schreiben.

```
a = 3
b = 4
c = 5
print("Drei Zahlen: ", end = "")
print(a, b, c)
```

Als Ergebnis erhalten wir:

```
Drei Zahlen: 3 4 5
```

5.10 Typumwandlung

In Python ist es möglich, den Typ einer Variable durch Zuweisung eines neuen Inhalts zu ändern. Einer Variable, die eine Zeichenkette enthält, können wir beispielsweise ohne weiteres eine Gleitkommazahl zuweisen. Der alte Inhalt wird überschrieben und der Variablentyp ändert sich automatisch. Viele andere Programmiersprachen erlauben eine solche dynamische Typisierung nicht. Dort muss gegebenenfalls schon vor der Verwendung einer Variable explizit deren Typ festgelegt werden.

Für den Fall, dass es in einem Python-Programm erforderlich ist, den Typ einer Variable gezielt zu verändern, stehen dazu für die bisher behandelten Variablentypen die Funktionen **`int()`**, **`float()`**, **`complex()`**, **`str()`** und **`bool()`** zur Verfügung.

Im folgenden Beispiel wird eine Zeichenkette, die das Zeichen „3“ enthält, in eine Ganzzahl mit dem Zahlenwert 3 umgewandelt, woraufhin sich ihr Verhalten grundlegend ändert.

```
>>> a = "3"

>>> a
'3'

>>> 4 * a
'3333'

>>> a = int(a)

>>> a
3

>>> 4 * a
12
```

Insbesondere Tastatureingaben durch die Funktion **input()** müssen erst in einen Zahlentyp umgewandelt werden, wenn sie als numerische Werte weiterverarbeitet werden sollen, denn die **input()**-Funktion gibt in Python 3 immer eine Zeichenkette zurück – selbst, wenn ausschließlich Ziffern eingetippt wurden.

Wir können diese Umwandlung entweder in zwei getrennten Schritten vornehmen …

```
z = input("Gib eine Zahl ein: ")
a = int(z)
```

… oder wir fassen die beiden Schritte elegant zusammen und sparen dadurch sowohl eine Programmzeile als auch eine Hilfsvariable ein:

```
a = int(input("Gib eine Zahl ein: "))
```

Auf diese Art verschachtelte Funktionen werden vom Python-Interpreter stets in der Reihenfolge „von innen nach außen“ ausgewertet. Hier wird also zuerst die Funktion **input** aufgerufen und deren Rückgabewert an die Funktion **int** weitergegeben. Der Rückgabewert von **int** wird schließlich der Variable **a** zugewiesen.

5.10.1 Evaluation von Ausdrücken

Python-Ausdrücke, die als Zeichenkette vorliegen, wie beispielsweise

"13 * 17"

"'Zeichen' + 'kette'" oder

"sin(1/x)"

können mit der Funktion **eval()** ausgewertet (evaluiert) werden.

```
>>> a = "13 * 17"
```

```
>>> eval(a)
221
```

Entdeckt Python in der Zeichenkette Variablennamen, werden die aktuellen Werte dieser Variable dort eingesetzt; findet es Funktionen, so werden diese ausgeführt.

```
>>> eval("a")
'13 * 17'

>>> eval(a + a)
378573
```

Die letzte Zahl, 378573, ist dabei das Ergebnis der Auswertung der Zeichenkette "13 * 1713 * 17", die das Ergebnis der Verknüpfung a + a darstellt.

Die Funktion **eval()** bietet sich als perfekte Ergänzung zur **input()**-Funktion an. Abhängig von der jeweiligen Eingabe hat der Rückgabewert von **eval(input())** immer den richtigen Typ[1].

Die Fähigkeit von **eval()**, ganz beliebige Funktionen auszuführen, kann zu einem Sicherheitsrisiko werden, falls Pythonprogramme von böswilligen Personen ausgeführt werden. Für Pythonprogramme auf öffentlich erreichbaren Webservern gilt daher ein striktes Verbot, **eval()** zu benutzen.

```
>>> a = eval(input("Gib Dein Alter ein: "))
Gib Dein Alter ein: 22

>>> a
22
```

1 In Python 2 war diese Umwandlung noch in die **input()**-Funktion eingebaut. Wer tatsächlich die buchstabengetreue Eingabe im Programm verwenden wollte, musste dort auf die Funktion **raw_input()** zurückgreifen.

```
>>> a = eval(input("Gib Dein Alter ein: "))
Gib Dein Alter ein: open("index.html","w").write("Bruahaha!")
```

Der Fachausdruck für das ungewollte Einschleusen von ausführbaren Befehlen an einer Stelle, die harmlose Daten erwartet, lautet „Code Injection“[1].

1 Jeder, der mit dem Thema zu tun hat, kennt Bobby Tables: https://xkcd.com/327/

5.11 Das erste richtige Programm

Bis jetzt haben wir jeden Python-Befehl einzeln in die Shell getippt, damit er ausgeführt wird. Um Befehle, die stets in einer bestimmten Reihenfolge ausgeführt werden sollen, nicht immer wieder neu schreiben zu müssen, können wir diese als Datei speichern. Die Folge von Befehlen nennen wir „Programm“ und die Textdatei mit den Programmbefehlen ist dementsprechend eine „Programmdatei“.

Um in IDLE eine neue Programmdatei anzulegen, drücken wir die Tastenkombination Strg N oder wählen die Menüfolge „File – New Window“.

Es erscheint ein leeres Fenster, in das wir nun unseren Programmtext eintippen können.

```
print("Dieses Programm addiert zwei Zahlen.")
a = float(input("Gib einen Zahlenwert für a ein: "))
b = float(input("Gib einen Zahlenwert für b ein: "))
c = a + b
print("Die Summe von",a,"und",b,"ist",c)
```

Mit der Taste F5 oder über die Menüfolge „Run – Run Module“ können wir das Programm starten. Vorher wird es automatisch gesichert.

```
Dieses Programm addiert zwei Zahlen.
Gib einen Zahlenwert für a ein: 23.45
Gib einen Zahlenwert für b ein: 56.78
Die Summe von 23.45 und 56.78 ist 80.23
```

Beim ersten Start werden wir gefragt, in welcher Datei das Programm gespeichert werden soll. Es ist eine gute Idee, den gewählten Dateinamen auf „.py“ enden zu lassen, damit sofort klar ist, dass es sich dabei um ein Python-Programm handelt. Die meisten Editoren stellen außerdem die farbliche Syntaxhervorhebung eines Python-Programms erst dann dar, wenn die Datei korrekt als .py-Datei gespeichert wurde.

Falls die Dateiendung „.py“ unter Microsoft Windows nicht angezeigt wird, empfiehlt sich in Blick in Kapitel 2.3.2.

5.12 Quelltextformatierung

5.12.1 Kommentarzeilen

Um ein Programm für Andere oder uns selbst nachvollziehbar zu halten, sollten wir zwischen die auszuführenden Zeilen hilfreiche Kommentare schreiben. Während des Schreibens eines Programms scheint das Kommentieren oft völlig überflüssig zu sein. Wir wissen ja schließlich genau, warum wir jede einzelne Zeile genau so formuliert haben! In ein paar Wochen wissen wir es aber vielleicht nicht mehr. Python gilt zwar als eine der lesbarsten Programmiersprachen überhaupt, doch alte oder fremde unkommentierte Quelltexte analysieren zu müssen, ist eine Strafarbeit.

Zur Kennzeichnung von Kommentaren wird das Doppelkreuz-Zeichen **#** verwendet, welches oft auch „Nummernzeichen“, „Hash“ oder „Raute“ genannt wird[1].

Sobald der Python-Interpreter dem Zeichen **#** außerhalb einer Zeichenkettenkonstante begegnet, wird der Rest der Zeile von ihm ignoriert.

```
#!/usr/bin/env python3
"""
Programm zur
Addition zweier
Zahlenwerte
"""
a = 4        # a festlegen
b = 7        # b festlegen

# Addition durchführen und
# Ergebnis in c speichern
c = a + b

# Ergebnis ausgeben
```

1 Die Bezeichnung „Hashtag“ für dieses Zeichen ist verbreitet, aber falsch. Ein Hashtag besteht immer aus der Kombination eines Doppelkreuzes mit einem Stichwort, zum Beispiel #bochum.

```
print(c)
```

Kommentare in derselben Zeile unterzubringen, in der sich bereits Code befindet, ist zwar erlaubt, jedoch nur bei sehr kurzen Zeilen sinnvoll.

Wir können sogar ganze Programmabschnitte „auskommentieren“, um sie vorübergehend oder dauerhaft von der Ausführung auszuschließen. Der IDLE-Editor hat dazu einen eigenen Menüpunkt „Comment Out Region“ im Menü „Format“.

Auch Zeichenkettenkonstanten, die ohne weiteren Bezug im Quelltext stehen, können die Funktion eines Kommentars übernehmen. Mehrzeilige Kommentare lassen sich dann mit drei Anführungszeichen einleiten und abschließen.

Eine besondere Form des Kommentars wird dazu verwendet, einen kurzen Hilfstext zu selbstgeschriebenen Funktionen auszugeben. Wenn zu Beginn einer Funktion eine Zeichenkettenkonstante ohne weitere Zuweisung formuliert wird, so wird ihr Inhalt ausgegeben, wenn die Funktion **`help()`** mit dem Namen der Funktion aufgerufen wird. Siehe auch Kapitel 5.17.

5.12.2 Zeilenlänge

Grundsätzlich sollten wir stets darauf achten, Quelltextzeilen nicht länger als 72, höchstens 79, Zeichen werden zu lassen, damit die Lesbarkeit des Programms nicht leidet.

Der Editor der Entwicklungsumgebung IDLE hat noch nicht einmal eine Möglichkeit des horizontalen Scrollens vorgesehen. Das ist vielleicht etwas eigenwillig, aber aus erzieherischen Gesichtspunkten sehr hilfreich.

Tatsächlich sind Sie durch diese Beschränkung kaum eingeschränkt, da sich zu lange Python-Ausdrücke in der Regel gut auf mehrere Zeilen verteilen lassen.

Bei der Ausgabe von Zeichenkettenkonstanten können Sie einfach ein Anführungszeichen setzen und in der nächsten Zeile weiterschreiben.

```
print("eine sehr lange Zeile")
```

ist dasselbe wie

```
print("eine sehr "
      "lange Zeile")
```

Ebenso können Sie innerhalb von Klammern fast beliebige Zeilenumbrüche vornehmen. Lediglich innerhalb eines Schlüsselworts oder eines Bezeichners sind keine Zeilenumbrüche zugelassen.

```
Variable = (Wert1 + Wert2 * Wert3)
```

kann auch als

```
Variable = (Wert1 +
            Wert2 *
            Wert3)
```

geschrieben werden.

Auch ohne umschließende Klammern sind Zeilenumbrüche möglich. Dazu trennen Sie die zu lange Zeile mit einem Rückwärtsschrägstrich.

```
Variable = Wert1 + \
           Wert2 * \
           Wert3
```

5.12.3 Groß- und Kleinschreibung

In diesem Skript wird die Groß- und Kleinschreibung von Variablennamen und anderen Bezeichnern durchgehend willkürlich verwendet. Es gibt jedoch auch Softwareentwickler, die hier nach einem strengen Schema vor-

gehen. Falls Sie einmal in einem Team arbeiten sollten und dort nicht nur programmieren, um sich selbst die Arbeit zu erleichtern, sollten Sie sich möglichst untereinander abstimmen, um Missverständnisse zu vermeiden.

Es ist vielleicht keine schlechte Idee, hier der Gestaltungsrichtlinie PEP 8 zu folgen. Diese sieht folgende Namenskonventionen vor[1]:

Namensbeispiel	Verwendung
`kleinschreibung` **`mit_unterstrich`**	Für alle Namen von Modulen, Variablen, Attributen, Funktionen und Methoden. Zur besseren Lesbarkeit zusammengesetzter Namen sind Unterstriche hilfreich.
`KamelSchreibweise`	Für die Namen von Klassen. Unterstriche werden nicht verwendet, stattdessen Großbuchstaben mitten im Wort.
`GROSSBUCHSTABEN` **`MIT_UNTERSTRICH`**	Für Konstanten bzw. Variablen deren Wert während des gesamten Programmlaufs unverändert bleibt.

Dies ist aber kein Zwang. Gerade, wenn Sie deutschsprachige Variablennamen verwenden, sieht eine konsequente Kleinschreibung eher befremdlich aus und daran, dass Konstanten immer großgeschrieben werden sollten, hält Python sich nicht einmal selbst mit seinen Bezeichnern **`None`**, **`True`** und **`False`** oder der Konstanten **`pi`** im Modul **`math`**.

5.12.4 Shebang und Zeichenkodierung

Unter Linux und macOS hat die erste Zeile eines Quelltextes eine besondere Funktion: Diese auch im Deutschen *Shebang*[2] genannte Zeile gibt dem Betriebssystem an, mit welchem Programm der Quelltext ausgeführt werden soll. Sollen zum Beispiel als „ausführbar“ markierte Python-Datei-

1 https://www.python.org/dev/peps/pep-0008/

2 Man könnte das englische Wort „*Shebang*“ mit „Gedöns“ übersetzen, aber das führt zu nichts.

en beim Doppelklicken nicht nur in den Editor geladen, sondern auch gestartet werden können, so kann man diese Programme einfach mit der Shebang-Zeile

```
#!/usr/bin/env python3
```

einleiten, damit das Betriebssystem immer den aktuellen Python-Interpreter für die Ausführung zur Verfügung stellt.

Unter Microsoft Windows gibt es diesen Schutz nicht. Python-Dateien sind dort, wie viele andere Dateitypen auch, unmittelbar ausführbar und werden in der Regel in einem Terminalfenster gestartet, das beim Programmende wieder geschlossen wird.

Leider ist unter Windows die Information, mit welchem Programm eine Datei geöffnet wird, fest an die Dateinamenerweiterung gebunden, deshalb gibt es dort in der Regel Probleme, wenn mehrere Versionen einer Software installiert sind und eine Datei beispielsweise durch Doppelklick im Windows-Explorer gestartet werden soll. Windows kennt zwar die Möglichkeit, eine Datei im Explorer rechtszuklicken und über den Menüpunkt „Öffnen mit …“ ein Programm auszuwählen, dieses Verfahren versagt jedoch, wenn die zu verwendenden Programme den gleichen Namen haben und lediglich in unterschiedlichen Verzeichnissen abgelegt sind. Unter Windows ist es daher üblich, nur eine einzige Version jeder installierbaren Software zu verwenden.

5.13 Verzweigungen

Nicht immer sind Programmabläufe so linear wie in den vorangegangenen Beispielen. Die interessanteren Programme führen je nach Ausgangssituation ganz unterschiedliche Programmteile aus.

So ist es zur Vermeidung von Fehlern während der Programmausführung (sogenannte Laufzeitfehler) sinnvoll, vor bestimmten Berechnungen die Eingangswerte zu überprüfen. Nur wenn diese Eingangswerte überhaupt plausibel sind, sollten wir unser Programm rechnen lassen. Wenn nicht, sollten wir stattdessen eine hilfreiche (und nicht allzu unfreundliche) Fehlermeldung ausgeben.

Beispiel: wenn wir versuchen, durch die Wurzelfunktion **`sqrt`** aus der Bibliothek **`math`** die Wurzel aus einer negativen Zahl zu ziehen,

```
>>> sqrt(-1)
```

so bricht Python die Programmausführung mit einer Fehlermeldung vom Typ *„math domain error“* ab. Dasselbe geschieht, wenn der Arkussinus einer Zahl ausgerechnet werden soll, die nicht im Intervall [-1, 1] liegt.

Die Entscheidung darüber, ob ein Programm bestimmte Befehle ausführt oder nicht, wird mithilfe der sogenannten logischen Ausdrücke getroffen. Das sind Ausdrücke, denen eindeutig einer beiden booleschen Wahrheitswerte **`True`** oder **`False`** entspricht.

Wenn Sie gerne etwas mehr über logische Ausdrücke erfahren möchten, als in den folgenden Beispielen behandelt wird, ist ein Ausflug in Kapitel 5.27 das Richtige für Sie.

5.13.1 Fallunterscheidungen: if … elif … else

Mit dem Schlüsselwort **`if`** (wenn) können wir einen logischen Ausdruck dazu nutzen, den Programmlauf zu beeinflussen. Nur wenn der Ausdruck hinter dem **`if`** eine wahre Aussage oder ein gleichwertiger Ausdruck ist, wird der darauf folgende Programmblock ausgeführt. Wir sprechen von einer „bedingten Ausführung“.

Ein „Programmblock“ ist dabei ein Abschnitt eines Programms, der aus einer oder mehreren zusammengehörenden Programmzeilen besteht.

In Python wird der auf die **if**-Bedingung folgende Programmblock dadurch festgelegt, dass die Zeilen dieses Blocks mit vier Leerzeichen eingerückt werden.[1]

```
z = float(input("Gib eine Zahl ein: "))
if z < 0:
    print("Die Zahl ist negativ.")
```

Die eingerückte Print-Funktion wird also nur dann ausgeführt, wenn die Bedingung „**z < 0**“ wahr ist.

Was aber können wir tun, um auch dann eine passende Meldung auszugeben, wenn die Bedingung „**z < 0**“ nicht wahr ist? Wir könnten zum Beispiel zwei Abfragen hintereinander durchführen.

```
if z < 0:
    print("Die Zahl ist negativ.")
if z >= 0:
    print("Die Zahl ist positiv oder null.")
```

1 In den bisher ersonnenen Programmiersprachen gibt es ganz unterschiedliche Schreibweisen, um die Zusammengehörigkeit eines Programmblocks zu kennzeichnen. Verbreitet ist die Verwendung von Begrenzern in Form von Schlüsselwörtern oder Klammern. In Pascal (Delphi) beginnt ein mehrzeiliger Programmblock immer mit dem Schlüsselwort **begin** und endet mit **end;**. In Java und C beginnt ein mehrzeiliger Programmblock mit **{** und endet mit **};**. Einzeilige Programmblöcke werden in allen drei genannten Sprachen nicht notwendigerweise mit Begrenzern versehen. Sie enden am obligatorischen Semikolon.
Der große Vorteil von Pythons Blockbildung durch Einrückung liegt in einer hervorragenden Lesbarkeit des Programmtextes. Python gilt nicht ohne Grund als eine der lesbarsten Programmiersprachen überhaupt.
Üblicherweise beträgt das Maß der Einrückung 4 Leerzeichen. Es sind auch andere Werte möglich, Sie dürften theoretisch sogar Tabulatorzeichen zum Einrücken verwenden; die Einrückung innerhalb eines Blocks muss aber stets einheitlich sein.

Das funktioniert zwar, ist aber viel zu kompliziert. Denn daraus, dass die Bedingung „**`z < 0`**“ nicht erfüllt ist, ergibt sich ja bereits, dass „**`z ≥ 0`**“ wahr sein muss. Wir erinnern uns an die Fallunterscheidungen im Struktogramm (Kapitel 4.2.2):

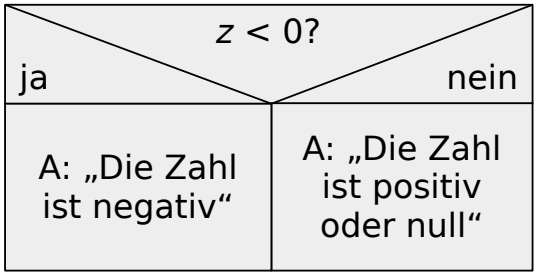


Abb. 67: Fallunterscheidung im Struktogramm

Hier ist nur eine einzige Abfrage notwendig. Würden wir den Algorithmus dazu verwenden, einem Menschen einen Auftrag zu geben, so würden wir ihm sagen: „Wenn z kleiner als null ist, dann schreibe »Die Zahl ist negativ«, sonst schreibe »Die Zahl ist positiv oder null«“.

Ein Schlüsselwort mit der Bedeutung „sonst“ gibt es in Python auch. Es heißt hier „**`else`**“. Wir können unser Programm also vereinfachen:

```
if z < 0:
    print("Die Zahl ist negativ.")
else:
    print("Die Zahl ist positiv oder null.")
```

Was aber, wenn wir drei oder mehr Fälle zu unterscheiden haben?

```
if z < 0:
    print("Die Zahl ist negativ.")
else:
    if z > 0:
        print("Die Zahl ist positiv.")
    else:
        print("Die Zahl ist null.")
```

Mit jeder zusätzlichen Abfrage müssen wir einen neuen Programmblock einrücken. Irgendwann würde der Platz knapp. Wir wollen ja keine Zeilen erhalten, die länger als 72, höchstens 79 Zeichen lang sind.

In Python dürfen wir daher auf die Einrückung verzichten, wenn direkt auf ein **else** wieder ein **if** folgt und wir beides mit dem Schlüsselwort **elif**[1] zusammenfassen.

Mit **elif** wird eine Abfrage in Python nur dann gestartet, wenn die vorangegangene Abfragebedingung nicht erfüllt wurde. Es können beliebig viele **elif**-Abfragen hintereinander ausgeführt werden. Ganz am Ende einer Kette von Abfragen können wir schließlich mit **else** alle übriggebliebenen Fälle einfangen, die bisher nicht berücksichtigt wurden.

Unser Programmbeispiel formulieren wir nun so:

```
z = float(input("Gib eine Zahl ein: "))
if z < 0:
    print("Die Zahl ist negativ.")
elif z > 0:
    print("Die Zahl ist positiv.")
else:
    print("Die Zahl ist null.")
```

1 Dass sowohl Else als auch Elif weibliche Vornamen sind, ist reiner Zufall.

Im Struktogramm sieht diese Fallunterscheidung so aus wie in Abb. 68.

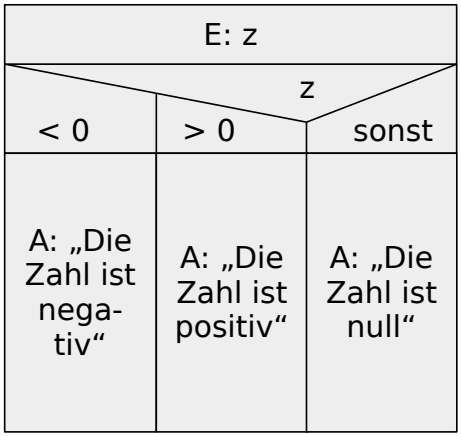


Abb. 68: if ... elif ... else im Struktogramm

Eine abschließende Bemerkung: Auch, wenn Fallunterscheidungen fast ein bisschen aussehen wie Schleifen: sie sind es nicht. Bitte, blamieren Sie sich nicht im Gespräch durch die Verwendung der Formulierung „If-Schleife“!

5.13.2 Fehlerbehandlung

Um unsere Programme nicht dadurch unnötig kompliziert zu machen, dass wir versuchen, alle möglichen Fehler vorherzusagen und durch kunstvoll geschachtelte **`if-elif-else`**-Konstruktionen abzufangen, können wir Python auch einfach anweisen, zu *versuchen*, einen Programmteil auszuführen und uns nur für den Fall, dass das schiefgeht, eine gute Reaktion überlegen.

Den Programmblock, dessen Ausführung versucht werden soll, leiten wir dabei mit **`try:`** ein (anschließend die Einrückung nicht vergessen!) und den Programmzeilen zur Fehlerbehandlung wird das Schlüsselwort **`except:`** vorangestellt.

Das könnte für den Versuch, die Wurzel aus dem Quotienten zweier Zahlen auszugeben (was bekanntlich immer dann schiefgeht, wenn der Nenner null oder der Bruch negativ ist), beispielsweise so aussehen:

```
try:
    print(sqrt(a/b))
```

```
except:
    print("Fehler!")
```

Wenn der **try**-Block ohne Fehler durchläuft, wird der **except**-Block nicht ausgeführt.

Die häufigste Fehlerquelle in einem laufenden Programm ist die Eingabe von Werten durch den Benutzer oder die Benutzerin. Hier kann von Leer–eingaben über Texteingaben und falsche Dezimalzeichen bis hin zu völlig unvorhergesehener Kreativität alles mögliche passieren. Das folgende Programmbeispiel fragt daher hartnäckig solange nach, bis es endlich einen gültigen Zahlenwert erhält. Die in der ersten Zeile angelegte boolesche Variable **ungültigeEingabe** bestimmt dabei, wie lange die Schleife[1] ausgeführt wird. Sie wird anfangs auf **True** gesetzt und verändert ihren Wert erst dann zu **False**, wenn die Umrechnung der Eingabe durch **float()** keinen Fehler hervorruft.

```
ungültigeEingabe = True

while ungültigeEingabe:
    try:
        a = float(input("Gib eine Zahl ein: "))
        ungültigeEingabe = False

    except:
        print("Das ist keine Zahl.")
```

Wer eine Neigung zu Tippfehlern hat, sollte bei der Verwendung der hier vorgestellten universellen Fehlerbehandlung allerdings vorsichtig sein, da auch Syntaxfehler den Sprung in den **except**-Block auslösen.

Um das zu vermeiden, gibt es in Python die Möglichkeit, für verschiedene Fehlerarten eigene Fehlerbehandlungsroutinen zu formulieren. Wer nur auf Zahlenumwandlungsfehler testen will, schreibt oben besser **except ValueError:** anstelle von **except:** hin.

1 Wie Sie die in Kapitel 4.2.4 vorgestellten Schleifen in Python realisieren können, erfahren Sie in Kapitel 5.14 auf Seite 142.

Damit unser Programm auf mehrere Fehlerbedingungen unterschiedlich reagieren kann, ordnen wir mehrere **except**-Blöcke untereinander an. Der Programmblock nach **except ValueError:** behandelt dann eine misslungene Zahlenumwandlung oder einen verfehlten Definitionsbereich, Nulldivisionen werden im Block nach **except ZeroDivisionError:** einer Fehlerbehandlung zugeführt und die eingerückten Programmzeilen nach **except KeyboardInterrupt:** kümmern sich um den Versuch der Anwenderin oder des Anwenders, das laufende Programm durch Drücken der Tastenkombination Strg C abzubrechen.

Wollen wir mit einem einzigen except-Block mehrere Fehlerbedingungen behandeln, so fassen wir diese zu einem Tupel zusammen. Um beispielsweise nach beiden oben genannten Rechenfehlern dieselbe Fehlerbehandlung auszuführen, schreiben wir

```
except (ValueError, ZeroDivisionError):
```

Eine Liste mit häufigen Fehlermeldungen finden Sie im Anhang auf Seite 324 dieses Textes.

5.14 Programmschleifen

Wenn unser Programm bestimmte Vorgänge wiederholen soll, dann ist es sinnvoll, die Programmbefehle dazu nicht mehrfach hintereinander zu schreiben, sondern nur einmal einen Programmblock zu formulieren, der dann mehrfach ausgeführt wird. In Python bilden wir einen Programmblock dadurch, dass wir ihn einrücken.

Diesem Programmblock stellen wir eine Zeile voran, die aussagt, wie oft oder unter welchen Bedingungen der Block wiederholt werden soll.

Die gesamte Konstruktion aus einleitender Bedingung und zu wiederholendem Programmblock nennen wir „Programmschleife“.

Python kennt zwei Schleifentypen: die bedingte Schleife, bei der die Wiederholung an den Wahrheitsgehalt einer logischen Aussage (Kapitel 5.27) gebunden ist, und die Zählschleife.

5.14.1 Bedingte Schleifen mit „while“

Eine bedingte Schleife wiederholt eine Anweisungsfolge solange, wie eine dazugehörige logische Aussage wahr ist. Sie wird mit dem Schlüsselwort **`while`** eingeleitet.

Als Beispiel diene ein einfacher Countdown-Zähler, der einen Zahlenwert **`i`** beginnend bei **`3`** so lange verkleinert, wie dieser größer oder gleich null ist.

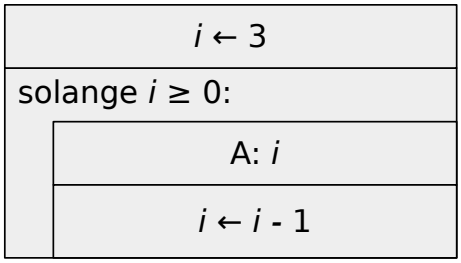


Abb. 69: Bedingte Schleife im Struktogramm

Die Umsetzung in einem Pythonprogramm sieht diesem Struktogramm bemerkenswert ähnlich:

```
i = 3
while i >= 0:
    print(i)
    i = i-1
```

Das Programm erzeugt folgende Ausgabe:

```
3
2
1
0
```

Vor der Schleife wird die Zählvariable **i** auf einen Startwert gesetzt. In der Schleife wird ihr Inhalt bei jedem Durchlauf erneut ausgegeben und anschließend um 1 verkleinert.

Schließlich springt die Programmausführung wieder zur Abfrage im Schleifenkopf oberhalb des eingerückten Schleifenkörpers, um erneut zu entscheiden, ob die Schleife wiederholt oder verlassen wird.

Damit die Schleife jemals wieder verlassen wird, muss innerhalb des Schleifenkörpers der Wahrheitswert der logischen Aussage in der einleitenden **if**-Abfrage geändert werden. Hängt diese Aussage allein von **i** ab, ist also irgendwo im Schleifenkörper der Wert von **i** zu ändern. Geschieht dies aufgrund eines Programmierfehlers nicht, läuft die Schleife endlos weiter. Eine recht häufige Fehlerursache ist das Testen zweier Gleitkommazahlen auf Gleichheit, da hier schon geringste Rundungsfehler dafür sorgen, dass die Zahlen als unterschiedlich angesehen werden. Siehe Kapitel 5.27.3.

Endlosschleifen in Computerprogrammen können schlimmstenfalls, je nach verwendeter Programmiersprache, den ganzen Rechner lahmlegen. Python ist da zum Glück recht umgänglich. Ein in eine Endlosschleife geratenes Python-Programm lässt sich mit der Tastenkombination Strg-C stoppen oder über den Fenster-Schließen-Button des Betriebssystems beenden.

Aussprung mit break

Im Gegensatz zu anderen Programmiersprachen, die zwischen abweisenden und nicht abweisenden Schleifen unterscheiden, gibt es in Python nur einen einzigen bedingten Schleifentyp.

Die Konsequenz daraus scheint zu sein, dass wir alle Algorithmen vom Typ „wiederhole etwas, solange eine Bedingung erfüllt ist“ so umformulieren müssen, dass sie als „solange eine Bedingung erfüllt ist, wiederhole etwas“ gelesen werden können.

Sie können ja mal versuchen, die nicht abweisende Schleife aus Abb. 70 in eine abweisende Schleife umzubauen, die dasselbe tut. Wie stellen Sie sicher, dass der Schleifenkörper mindestens ein Mal ausgeführt wird?

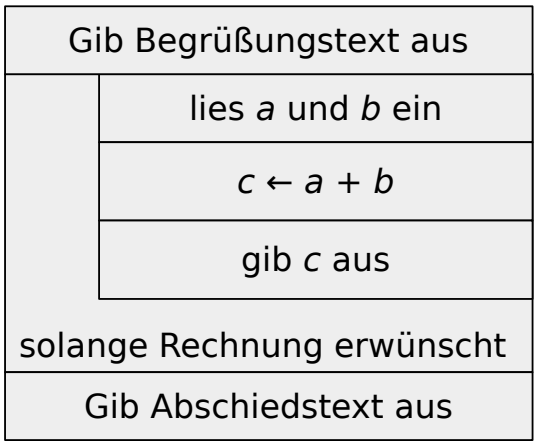


Abb. 70: Nicht abweisende Schleife im Struktogramm

Es wäre aber doch schade, wenn wir vorhandene Algorithmen extra umbauen müssten, um sie in Python umsetzen zu können.

Tatsächlich müssen wir das nicht und die Lösung ist auch gar nicht schwer. Damit die Schleife auf jeden Fall betreten wird, formulieren wir eine Schleifenbedingung, die immer erfüllt ist.

```
while True:
```

Nun schreiben wir den Programmblock des Schleifenkörpers und ganz am Ende formulieren wir eine **`if`**-Abfrage, die die Schleife beenden kann. Python erlaubt es, eine Schleife jederzeit zu verlassen, indem wir das Schlüsselwort **`break`** verwenden.

```
print("Der unermüdliche Addierer für a + b = c")

while True:
    a = float(input("Gib einen Zahlenwert für a ein: "))
    b = float(input("Gib einen Zahlenwert für b ein: "))
    c = a + b
    print("Die Summe von",a,"und",b,"ist",c)

    jn = input("Noch eine Berechnung (j/n)? ")
    if jn in ("n", "N"):
        break

print("Es war mir ein Vergnügen, für Dich zu addieren.")
```

Zum Vergleich hier ein ähnliches Programm, aber diesmal mit einer abweisenden Schleife ohne **break**. Nun ist zwar eine Hilfsvariable (**Berechnung_erwünscht**) notwendig, die zuerst einmal auf **True** gesetzt werden muss, damit die abweisende Schleife überhaupt betreten werden kann, das Programm wirkt jedoch aufgeräumter und lesbarer:

```
print("Der unermüdliche Addierer für a + b = c")

Berechnung_erwünscht = True

while Berechnung_erwünscht:
    a = float(input("Gib einen Zahlenwert für a ein: "))
    b = float(input("Gib einen Zahlenwert für b ein: "))
    c = a + b
    print("Die Summe von",a,"und",b,"ist",c)

    jn = input("Noch eine Berechnung (j/n)? ")
    Berechnung_erwünscht = jn not in ("n", "N")

print("Es war mir ebenfalls ein Vergnügen.")
```

Unstrukturierte Programmierung

Grundsätzlich sollten wir eine Schleife durch die Verwendung von **`break`** nur dann verlassen, wenn unser Programm dadurch wirklich lesbarer wird. Dasselbe gilt für das Zurückspringen zum Schleifenkopf mithilfe des ähnlichen Befehls **`continue`**. Ein Nachteil beider Sprungbefehle ist, dass sich die damit verbundenen Algorithmen nicht mehr strukturiert durch Nassi-Shneidermann-Diagramme darstellen lassen, was in der Informatikausbildung oft dazu führt, sie aus erzieherischen Gründen zu verbieten. Ebenso verhält es sich mit dem vorzeitigen Herausspringen aus einer Funktion mithilfe von **`return`**.

Mit Flussdiagrammen lassen sich die damit verbundenen „Abkürzungen“ im Programmlauf zwar noch darstellen, die damit verbundenen Überkreuzungen der Ablaufpfade führen aber bei intensiver Verwendung zu einem unleserlichen Programmierstil, der als „Spaghetticode“ verrufen ist.

5.14.2 Verkürzte Arithmetiknotation

Um ein wenig Tipparbeit zu sparen – und um unsere Programme lesbarer zu gestalten – können wir bei Zuweisungen, bei denen ein Variablenname beiderseits des Zuweisungszeichens **`=`** auftauchen würde, eine verkürzte Schreibweise anwenden. Dazu dürfen wir den Variablennamen auf der rechten Seite der Zuweisung fortlassen. Stattdessen stellen wir den verbleibenden Operator nun dem Zuweisungszeichen voran.

Folgende Schreibweisen sind jeweils gleichwertig:

ausführlich	verkürzt	Resultat
`x = x + 1`	**`x += 1`**	x wird um 1 erhöht (inkrementiert)
`x = x - 1`	**`x -= 1`**	x wird um 1 vermindert (dekrementiert)
`x = x * 2`	**`x *= 2`**	x wird verdoppelt
`x = x / 2`	**`x /= 2`**	x wird halbiert
`x = x ** 2`	**`x **= 2`**	x wird quadriert

5.14.3 Iterationsschleifen mit „for“

Eine Iterationsschleife wiederholt einen Programmblock für jedes einzelne Element einer aus mehreren Elementen bestehenden Sequenz, beispielsweise einer Liste.

```
for a in [1, 2, 3, "Hut", "Stock", "Regenschirm"]:
    print("•", a)
```

Ergebnis:

```
• 1
• 2
• 3
• Hut
• Stock
• Regenschirm
```

Iterierbare Objekte begegnen uns in der Datenverarbeitung überaus häufig. Auch die Buchstaben einer Zeichenkette, die Zeilen einer Textdatei oder die Dateien eines Verzeichnisses können sequenziell mit einer for-Schleife abgearbeitet werden.

5.14.4 Die Funktion range()

Die Funktion **`range()`** verhält sich wie ein iterierbares Objekt aus einer Folge ganzzahliger Werte, über die wir beispielsweise eine **`for`**-Schleife laufen lassen können.

Im einfachsten Fall wird sie mit der gewünschten Zahl der Folgenelemente als einzigem Parameter aufgerufen. Der Aufruf **`range(Anzahl)`** erzeugt ein **`range`**-Objekt, das uns in einer Schleife alle ganzen Zahlen von null bis **`Anzahl-1`** liefert.

```
for a in range(5):
    print("•", a)
```

Ergebnis:

```
• 0
• 1
• 2
• 3
• 4
```

Soll die Zahlenfolge nicht bei null beginnen, so geben wir einen Startwert vor: **`range(Startwert, Grenzwert)`**.

```
for a in range(3, 7):
    print("•", a)
```

Ergebnis:

```
• 3
• 4
• 5
• 6
```

Als dritten Parameter können wir schließlich noch eine ganzzahlige Schrittweite angeben, um den Startwert um einen anderen Wert als eins zu erhöhen: **`range(Startwert, Grenzwert, Schrittweite)`**. Die letzte erzeugte Zahl ist bei positiver Schrittweite immer kleiner als der Grenzwert.

```
for a in range(5, 11, 2):
    print("•", a)
```

Ergebnis:

```
• 5
• 7
```

```
• 9
```

5.14.5 Generatoren

Funktionen wie **range()** können wir uns leicht selbst schreiben. Anstelle des Schlüsselwortes **return**, welches üblicherweise[1] die Funktion beendet und einen Wert zurückgibt, müssen wir lediglich das Schlüsselwort **yield** verwenden. Solche Funktionen nennen wir Generatorfunktionen oder kurz Generatoren.

Der Aufruf einer Generatorfunktion erzeugt einen Iterator, über den wir eine for-Schleife laufen lassen können.

Als einfaches Beispiel diene die folgende Generatorfunktion, die ganz ähnlich wie **range()** funktioniert, jedoch auch mit Gleitkommazahlen umgehen kann.

```
def floatrange(von, bis, schrittweite):
    x = von
    while x < bis:
        yield x
        x += schrittweite
```

Rufen wir diese Generatorfunktion im Kopf einer for-Schleife auf …

```
for f in floatrange(1.5, 4.5, 0.5):
    print(f)
```

… so erhalten wir folgendes Ergebnis:

```
1.5
2.0
2.5
3.0
```

1 Siehe Kapitel 5.17

```
3.5
4.0
```

Generatorausdrücke

Wir können Generatoren auch ohne eine eigene Funktion dadurch erzeugen, dass wir einen Generatorausdruck formulieren. Das ist ein beliebiger Python-Ausdruck, dem ein Schleifenausdruck nachgestellt ist.

Der Generatorausdruck

```
(i**2 for i in range(1, 11))
```

liefert zum Beispiel einen Generator für alle Quadratzahlen von 1 bis 100.

Wir können Generatorausdrücke auch unmittelbar zur Listenerzeugung verwenden, indem wir eckige anstelle runder Klammern verwenden. Entsprechend erhalten wir durch Verwendung geschweifter Klammern Sets aus den Elementen des Generatorausdrucks.

```
>>> (i**2 for i in range(1, 11))
<generator object <genexpr> at 0x7f4468ce5a50>

>>> [i**2 for i in range(1, 11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

>>> {i**2 for i in range(1, 11)}
{64, 1, 4, 36, 100, 9, 16, 49, 81, 25}
```

Pythons Schreibweise bei Generatorausdrücken kommt der mathematischen Notation von Mengen sehr nahe. In der Mathematik definiert man die Quadratzahlen der natürlichen Zahlen ℕ beispielsweise als **{x² | x ∈ ℕ}** und in Python als **{x**2 for x in N}** – mit dem Unterschied, dass **N** hier nicht unendlich groß sein darf, sondern eine endliche Menge von Zahlen enthalten muss.

Zusätzlich zur obligatorischen Schleife kann bei Generatorausdrücken noch eine Auswahlbedingung formuliert werden. Das Ergebnis erhält dann nur diejenigen Werte, für die die angegebene Bedingung erfüllt ist.

Wollen wir beispielsweise eine Liste aller natürlichen Zahlen kleiner als 20 erzeugen, die weder durch 2 noch durch 3 teilbar[1] sind, so schreiben wir:

```
[i for i in range(20) if i%2 and i%3]
```

Heraus kommt diese Liste:

```
[1, 5, 7, 11, 13, 17, 19]
```

In der Literatur wird für die Listenerzeugung durch Generatorausdrücke gelegentlich auch in deutschsprachigen Werken die Bezeichnung „List Comprehension“ verwendet.

5.14.6 Else und die Schleifen

Schleifen können regulär oder irregulär beendet werden.

For-Schleifen werden regulär beendet, nachdem die Laufvariable alle Werte des zu durchlaufenden iterierbaren Objekts angenommen hat. While-Schleifen werden regulär beendet, wenn die Schleifenbedingung den Wert **`False`** annimmt.

Der Aussprung mit **`break`** beendet eine Schleife dagegen irregulär.

Um einen Programmblock nur dann auszuführen, wenn eine Schleife regulär beendet wurde, können wir einer Schleife einen **`else`**-Zweig anhängen.

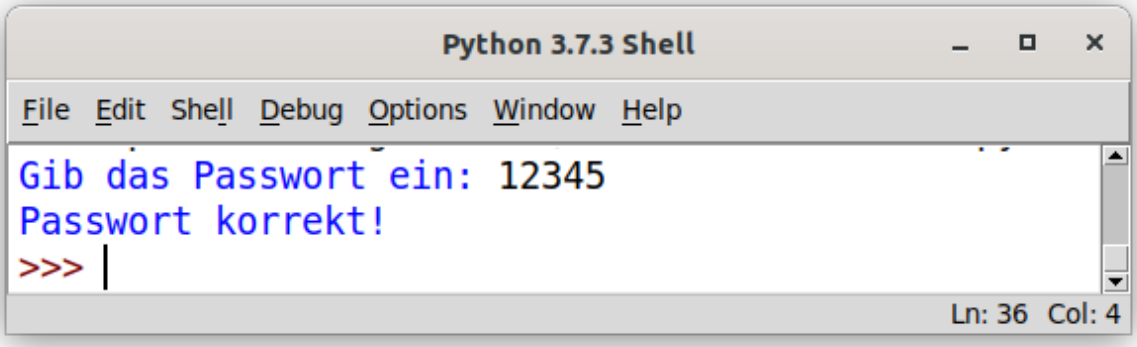
---

1 Den zur Feststellung der Teilbarkeit verwendeten Modulo-Operator % haben wir in Kapitel 5.6 auf Seite 109 kennengelernt – falls Sie sich nicht mehr an ihn erinnern, schauen Sie dort schnell nochmal nach.
Wenn Sie sich darüber wundern, dass in der Codezeile nur „i%2“ steht, obwohl Sie eigentlich „i%2 != 0“ erwartet haben, finden Sie die Auflösung in Kapitel 5.27.1.

Ein Beispiel: Ein Programm soll ein Passwort abfragen. Wenn das Passwort drei Mal falsch eingegeben wurde, soll eine Meldung ausgegeben werden.

```
for i in range(3):
    if input("Gib das Passwort ein: ") == "12345":
        print("Passwort korrekt!")
        break
else:
    print("Passwort wurde drei Mal falsch eingegeben.")
```

Je nachdem, ob die Schleife regulär oder irregulär beendet wird, kommt der Else-Zweig zur Ausführung oder nicht.



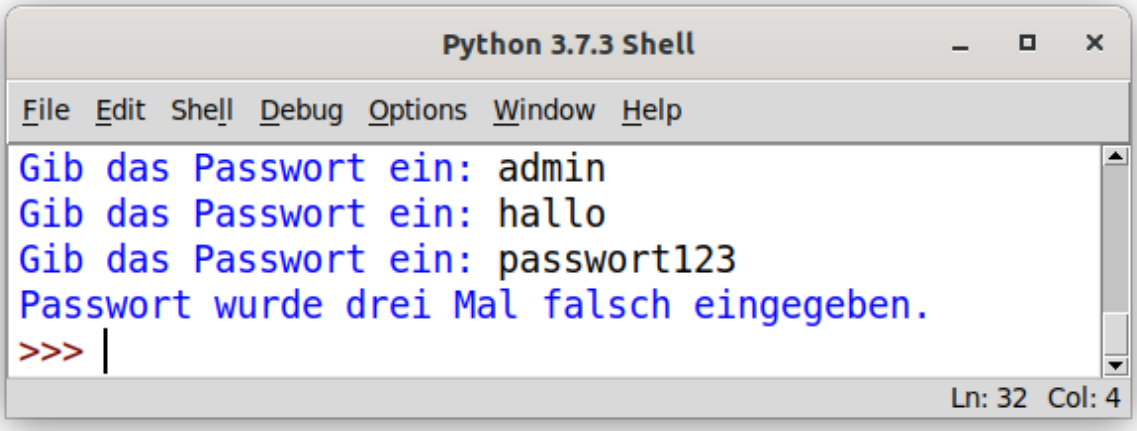


Abb. 71: Else-Zweig einer For-Schleife

5.14.7 Verschachtelte Schleifen

Schleifen dürfen andere Schleifen enthalten.

Der folgende Vierzeiler verwendet zwei ineinander verschachtelte Schleifen:

```
for a in range(1, 4):
    print(f"Vielfache von {a}:")
    for b in range(1, 4):
        print(f" {a} mal {b} ergibt {a*b}")
```

Bei jedem einzelnen Durchlauf der äußeren Schleife (a) werden sämtliche Durchläufe der inneren Schleife (b) erneut ausgeführt:

```
Vielfache von 1:
 1 mal 1 ergibt 1
 1 mal 2 ergibt 2
 1 mal 3 ergibt 3
Vielfache von 2:
 2 mal 1 ergibt 2
 2 mal 2 ergibt 4
 2 mal 3 ergibt 6
Vielfache von 3:
 3 mal 1 ergibt 3
 3 mal 2 ergibt 6
 3 mal 3 ergibt 9
```

Das Struktogramm in Abb. 72 zeigt dieselben beiden verschachtelten Schleifen wie das Programm oben.

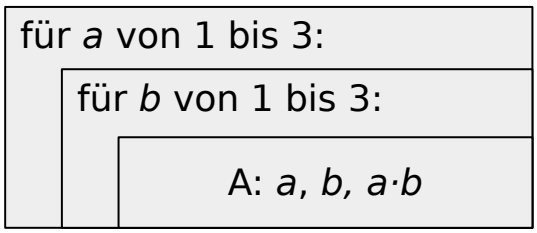


Abb. 72: Verschachtelte Schleifen im Struktogramm

Für den Algorithmus unwesentliche Details, wie die genaue Formulierung der Print-Ausgaben, sind in Struktogrammen fehl am Platz.

In Kapitel 4.2 finden Sie eine Übersicht der Möglichkeiten, mit denen Struktogramme Ihnen helfen können, Algorithmen programmiersprachen-unabhängig darzustellen.

5.15 Sequenzen

Sequenzen sind Objekte, die aus mehreren Elementen zusammengesetzt sind.

Einer Klasse von Sequenzen sind wir bereits kurz begegnet: der Zeichenkette. In den folgenden Kapiteln lernen wir zunächst weitere Sequenztypen wie Listen, Tupel, Mengen und Dictionarys kennen, bevor wir uns die Zeichenketten in Kapitel 5.21 noch einmal etwas näher ansehen.

5.15.1 Listen

Listen sind sehr praktische Datenstrukturen, mit denen wir eine große Anzahl von Werten in einer einzigen Variable speichern können. Die Elemente von Listen dürfen beliebige Objekte, wie zum Beispiel Zahlen, Zeichenketten, Funktionen oder sogar andere Listen sein.

Eine Liste wird in Python durch eckige Klammern dargestellt, zwischen denen sich die durch Kommas getrennten Listenelemente befinden.

```
Zahlen = [1, 5, 3, 2, 3]
Tiere = ["Hund", "Katze", "Maus"]
Mischmasch = [1, "Tisch", 2/3, False, Tiere, print]
```

Listen sind veränderlich. Es ist möglich, nachträglich Elemente anzuhängen, zu löschen oder auszutauschen.

```
>>> A = [2, 3, 5]
>>> A
[2, 3, 5]

>>> A.append(7)
>>> A
[2, 3, 5, 7]
```

```
>>> A.remove(3)
>>> A
[2, 5, 7]

>>> A[1] = 6
>>> A
[2, 6, 7]
```

Listen aus Listen

Wenn Sie eine Liste haben, die auch wieder aus Listen besteht, so können Sie auf jedes einzelne Element sowohl der äußeren Liste als auch der inneren Listen über Indizes zugreifen.

Angenommen, Sie arbeiten mit einer Liste, die aus Listen mit den Eigenschaften Name, Hülle und Länge einzelner Tiere zusammengesetzt ist,

```
L = [["Hund", "Fell", 75], ["Fisch", "Schuppen", 25],
     ["Wal", "Haut", 450]]
```

so finden Sie in dieser Tierliste demnach die drei Eigenschaftenlisten

`L[0]` mit dem Wert **`["Hund", "Fell", 75]`**
`L[1]` mit dem Wert **`["Fisch", "Schuppen", 25]`**

und

`L[2]` mit dem Wert **`["Wal", "Haut", 450]`**

und Sie können über Indizes auf alle Eigenschaften jedes einzelnen Tieres zugreifen.

`L[0][0]` hat den Wert **`"Hund"`**
`L[0][1]` hat den Wert **`"Fell"`**
`L[1][1]` hat den Wert **`"Schuppen"`**

und

`L[2][2]` hat den Wert **`450`**.

Sie können auch mit einer Schleife auf die Elemente zugreifen:

```
for Tier in L:
    print("Ein", Tier[0], "hat", Tier[1],
          "und ist", Tier[2], "cm lang.")
```

Dasselbe lässt sich noch kompakter so schreiben:

```
for Name, Hülle, Länge in L:
    print("Ein", Name, "hat", Hülle,
          "und ist", Länge, "cm lang.")
```

Gelegentlich kann es auch sinnvoll sein, durch Indize auf die Listenelemente zuzugreifen:

```
for i in range(len(L)):
    print("Ein", L[i][0], "hat", L[i][1],
          "und ist", L[i][2], "cm lang.")
```

5.15.2 Tupel

Ein Tupel kann grundsätzlich wie eine Liste verwendet werden, ist im Gegensatz zu dieser aber unveränderlich. Geschrieben werden Tupel als durch Kommas getrennte Werte in runden Klammern.

```
Liste = [1, 5, 3, 2, 3]
Tupel = (1, 5, 3, 2, 3)
```

5.15.3 Mengen (Sets)

Mengen sind nahe Verwandte von Listen, zeichnen sich aber dadurch aus, dass jedes Element nur genau einmal in ihnen vorkommen darf. Geschrieben werden Mengen als durch Kommas getrennte Werte in geschweiften Klammern.

```
>>> Menge = {1, 5, 3, 2, 3}
>>> Menge
{1, 2, 3, 5}
```

Die Reihenfolge der Elemente einer Menge ist nicht bestimmt. Mengen sind ungeordnete Sequenzen.

Mehrere Mengen können verknüpft werden, indem wir beispielsweise Vereinigungsmengen oder Schnittmengen bilden.

```
>>> A = {1, 3, 6, 5, 2}
>>> B = {4, 2, 6, 8, 3}

>>> A | B
{1, 2, 3, 4, 5, 6, 8}

>>> A & B
{2, 3, 6}
```

5.15.4 Dictionarys

Dictionarys sind, wie Mengen, ungeordnete Sequenzen, sie bestehen aber nicht aus einzelnen Elementen, sondern aus Schlüssel-Werte-Paaren. Jeder Schlüssel ist einmalig. Ihm können wir beliebige Werte zuordnen. Über den Schlüssel greifen wir auch wieder auf die einzelnen Werte des Dictionarys zu.

```
>>> Student = {"Name":"Kevin", "Alter":21}

>>> Student["Name"]
'Kevin'

>>> Student["Alter"]
21
```

Dictionarys können wir jederzeit erweitern, indem wir neuen Schlüsseln Werte zuweisen.

```
>>> Student["Hobby"] = "Tauchen"

>>> Student
{'Name': 'Kevin', 'Alter': 21, 'Hobby': 'Tauchen'}
```

Wenn wir versuchen, einen Wert abzufragen, zu dem kein Schlüssel existiert, wirft Python die Fehlermeldung **KeyError** aus. Wir können das vermeiden, indem wir den Wert nicht direkt, sondern über die Methode **get** des Dictionarys abfragen. Zu nicht vorhandenen Schlüsseln wird dann der Wert **None** zurückgegeben.

```
>>> print(Student.get("Geld"))
None
```

5.15.5 Indizes

Jedes Element einer geordneten Sequenz (Liste, Tupel, Zeichenkette) lässt sich über seine Position in der Sequenz ansprechen.

Die Indexierung der Elemente beginnt bei null. Besitzt eine Liste **L** zehn Elemente, so können diese in der Indexschreibweise mit **L[0]** bis **L[9]** adressiert werden.

```
Zahlen = [1, 2, 3]
Tiere = ["Hund", "Katze", "Maus"]
Mischmasch = [1, "Tisch", 2/3, False, Tiere, print]

print(Zahlen[0], Tiere[1], Mischmasch[2])
```

Ausgabe:

```
1 Katze 0.6666666666666666
```

Python erlaubt auch negative Indizes. Sie dienen dazu, auf die hinteren Elemente einer Sequenz zuzugreifen. **L[-1]** ist dabei das letzte Element der Liste **L**, **L[-2]** das vorletzte und so weiter.

5.15.6 Schleifen über Sequenzen

Sequenzen sind iterierbare Objekte. Eine besonders einfache Art, alle Elemente einer Sequenz zu verarbeiten, besteht darin, eine Schleife über die einzelnen Elemente laufen zu lassen. Die Schleifenvariable nimmt dabei nacheinander alle Werte der in der Sequenz enthaltenen Elemente an:

```
Tierliste = ["Hund", "Katze", "Maus"]

for Tier in Tierliste:
    print("•", Tier)
```

Ausgabe:

```
• Hund
• Katze
• Maus
```

Eine Besonderheit bilden Dictionarys. Hier läuft die Schleife nicht über die Werte, sondern die Schlüssel des Dictionarys.

```
Student = {'Name': 'Kevin', 'Alter': 21, 'Hobby': 'Tauchen'}
for Schlüssel in Student:
    print(Schlüssel, Student[Schlüssel])
```

Ausgabe:

```
Name Kevin
Alter 21
Hobby Tauchen
```

Da wir bei Dictionarys eigentlich immer zwei zusammengehörige Elemente haben, die uns interessieren, nämlich den Schlüssel und den dazugehörigen Wert, ist es nicht ganz ungeschickt, die Schleife gleich über beides laufen zu lassen.

```
Student = {'Name': 'Kevin', 'Alter': 21, 'Hobby': 'Tauchen'}
for Schlüssel, Wert in Student.items():
    print(Schlüssel, Wert)
```

Die Ausgabe ist dieselbe wie vorhin:

```
Name Kevin
Alter 21
Hobby Tauchen
```

5.15.7 Sequenzabschnitte (Slices)

Kopien einzelner Abschnitte einer geordneten Sequenz nennen wir „Slices“. Wir erzeugen sie, indem wir anstelle einer einzelnen Indexzahl einen Bereich **[von:bis]** angeben.

Die Angabe **bis** ist dabei der Indexwert des ersten Elementes, das nicht mehr zum Slice gehört. Wir kennen das ja schon von der Range-Funktion (Kapitel 5.14.4). Es ist daher mitunter intuitiver, sich die Indizes nicht so vorzustellen, dass sie auf die einzelnen Elemente zeigen, sondern auf die Trennlinie dazwischen.

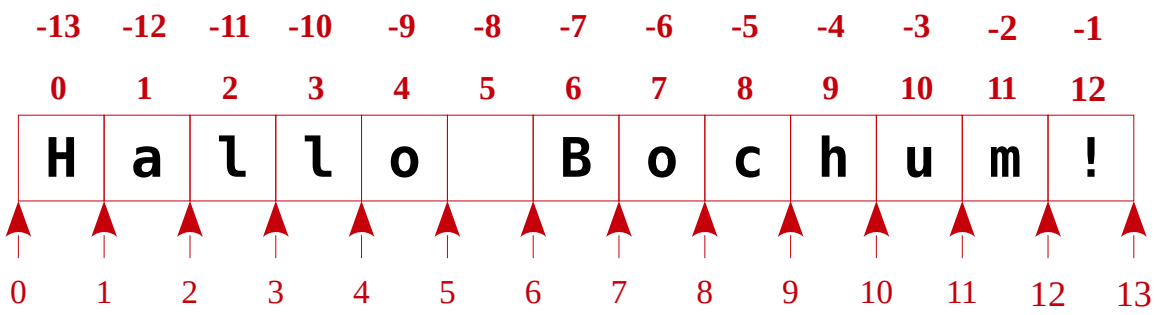


Abb. 73: Merkhilfe für Sequenzabschnitte

Slices enthalten standardmäßig eine Gruppe direkt aufeinander folgender Elemente. Es ist aber auch die Angabe einer Schrittweite erlaubt:

```
L [ von : bis : Schrittweite ]
```

Jede der drei Angaben dürfen wir weglassen. Anstelle von **von** wird dann **0** verwendet, anstelle von **bis** wird **len(L)** angenommen und die Schrittweite ist standardmäßig auf **1** gesetzt. Der Ausdruck **L[:]** erzeugt eine vollständige Kopie der Sequenz **L**.

Einige Beispiele:

```
Monat = ("Jan", "Feb", "Mär", "Apr", "Mai", "Jun",
         "Jul", "Aug", "Sep", "Okt", "Nov", "Dez")

print(Monat[3:6])
print(Monat[1:12:2])
print(Monat[::3])
```

Ergebnis:

```
('Apr', 'Mai', 'Jun')
('Feb', 'Apr', 'Jun', 'Aug', 'Okt', 'Dez')
('Jan', 'Apr', 'Jul', 'Okt')
```

5.15.8 Kopieren einer Sequenz

Im Gegensatz zu einfachen Datentypen wie Ganzzahlen oder Zeichenketten werden Sequenzen wie Listen, Mengen oder Dictionarys intern immer nur als Verweis auf einen Speicherbereich behandelt, an dem die eigentlichen Inhalte liegen.

Das erhöht zwar die Verarbeitungsgeschwindigkeit von Python sehr, führt aber zu dem unerwarteten Phänomen, dass die Zuweisung einer Sequenz zu einer Variable keine neue unabhängige Sequenz erzeugt, sondern nur einen Aliasnamen für die bereits vorhandenen Inhalte.

Um wirklich unabhängige Kopien unserer Daten zu erhalten, können wir die im vorigen Kapitel vorgestellten Slices verwenden.

Ein Beispiel: Angenommen, wir haben drei Listen **A**, **B** und **C**. Liste **A** enthält die drei Zahlen **1**, **2** und **3**. Liste **B** ist eine Kopie durch einfache Zuweisung und Liste **C** ist eine Kopie durch Slice-Bildung.

```
>>> A = [1, 2, 3]
>>> B = A
>>> C = A[:]
```

Nun weisen wir dem Element an der Indexposition 1 von Liste A einen neuen Wert zu.

```
>>> A [1] = "oh!"
```

Was passiert nun?

```
>>> A
[1, 'oh!', 3]
```

Das haben wir erwartet. An Indexposition 1 befindet sich das zweite Element von Liste A. Wir wissen ja, dass die Zählung bei null beginnt.

```
>>> B
[1, 'oh!', 3]
```

Das ist jetzt vielleicht überraschend. Liste B haben wir schließlich gar nichts explizit zugewiesen. Da sie aber ebenfalls nur auf die Inhalte der Liste A zeigt, und diese Inhalte von uns verändert wurden, betrifft die Änderung auch Liste B.

```
>>> C
[1, 2, 3]
```

Liste C zeigt sich davon völlig unbeeindruckt. Sie ist eine „echte Kopie“ der Inhalte von Liste A.

Auf der Website „Pythontutor“ wird das Phänomen grafisch recht anschaulich dargestellt:

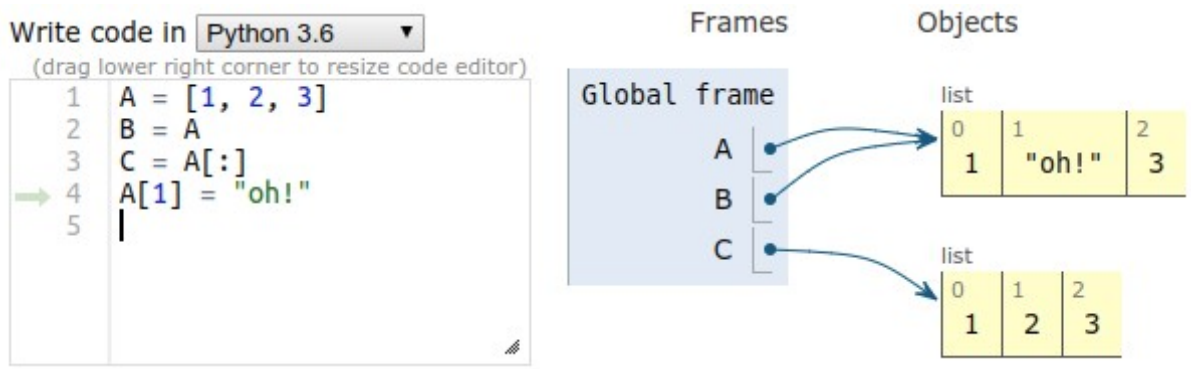


Abb. 74: www.pythontutor.com

Kopien verschachtelter Sequenzen

Wenn eine Sequenz wiederum Sequenzen enthält, genügt es nicht, eine Kopie der übergeordneten Sequenz mithilfe von Slices zu erzeugen, denn die untergeordneten Sequenzen verweisen ja immer noch auf dieselben Speicherbereiche wie zuvor. Um wirklich sicher zu sein, unabhängige Daten zu erhalten, müssen wir auch alle untergeordneten Sequenzen sorgfältig kopieren. Glücklicherweise bleibt es uns erspart, das selbst zu programmieren. Wir verwenden stattdessen die Funktion **`deepcopy`** aus dem Modul **`copy`**.

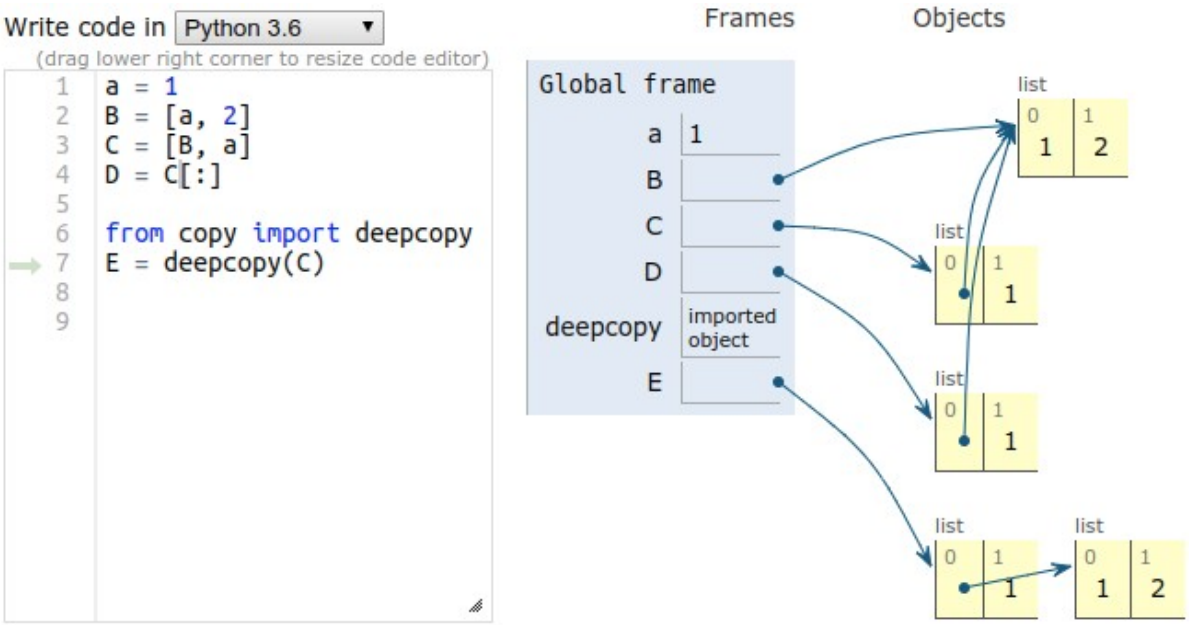


Abb. 75: deepcopy

5.15.9 Umwandlung eines Generator-Objektes in eine Liste

Mit der Funktion **`list()`** können wir nicht nur andere Sequenzen, sondern auch die von einem Generator (Kapitel 5.14.5) oder der Range-Funktion (Kapitel 5.14.4) erzeugten Objekte zu einer Liste zusammenfassen.

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

>>> list(floatrange(5.0, 7.1, 0.25))
[5.0, 5.25, 5.5, 5.75, 6.0, 6.25, 6.5, 6.75, 7.0]
```

5.15.10 Sequenzen sprengen

Wenn wir Sequenzen als Parameter an eine Funktion übergeben, so erhält die Funktion die Sequenz „am Stück“. Die Print-Funktion beispielsweise gibt eine Liste als Ganzes aus.

```
>>> Motto = ["per", "aspera", "ad", "astra"]

>>> print(Motto)
['per', 'aspera', 'ad', 'astra']
```

Wir können aber auch dem Sequenznamen ein Sternchen „*“ voranstellen, um die Sequenz zu sprengen. Der Funktion werden die einzelnen Sequenzelemente dann so übergeben, als seien sie beim Aufruf der Funktion einzeln, wie mit Kommas getrennte Parameter, geschrieben worden.

```
>>> print(*Motto)
per aspera ad astra
```

5.15.11 Das enumerate-Objekt

Wenn wir eine Schleife über alle Elemente einer Liste laufen lassen, haben wir oft den Wunsch, neben dem gerade betrachteten Listenelement auch dessen Indexwert zu kennen.

In vielen Programmiersprachen muss man sich damit behelfen, eine Schleife über die Indexwerte laufen zu lassen und mit der Laufvariable der Schleife wieder auf die Liste zugreifen:

```
A = ["alpha", "beta", "gamma"]
for i in range(len(A)):
    print(i, A[i])
```

Das ist in Python eher unüblich, denn hier gibt es die einzigartige Möglichkeit, gleichzeitig mehrere Variablen als Laufvariablen zu verwenden. Das **enumerate**-Objekt unterstützt uns zusätzlich, indem es die Elemente einer vorhandenen Liste gemeinsam mit deren Indexwerten zu Tupeln zusammenfasst.

Nun können wir zwei Variablen gleichzeitig über die von **enumerate** erzeugten Tupel laufen lassen und haben so in jedem Schleifendurchlauf beide Informationen zur Verfügung:

```
A = ["alpha", "beta", "gamma"]
for i, a in enumerate(A):
    print(i, a)
```

Das Resultat beider Programme ist dasselbe:

```
0 alpha
1 beta
2 gamma
```

Für den Fall, dass unser Zähler i nicht bei null beginnen soll, können wir seinen Startwert festlegen:

```
A = ["alpha", "beta", "gamma"]
for i, a in enumerate(A, start=1):
    print(i, a)
```

Das Resultat ist nun:

```
1 alpha
2 beta
3 gamma
```

5.15.12 Reißverschlussverfahren: das Zip-Objekt

Wir können eine beliebige Zahl von Sequenzen zu einer neuen Sequenz zusammenschnüren, in der jedes Element aus einem Tupel der korrespondierenden Elemente aller ursprünglichen Sequenzen besteht. Das erste Tupel besteht also aus den ersten Elementen aller Sequenzen, das zweite Tupel aus den zweiten Elementen und so weiter.

```
Obst = ["Kirsche", "Banane", "Apfel"]
Farbe = ["rot", "gelb", "grün"]
Obstfarben = list(zip(Obst, Farbe))
```

Die Liste „Obstfarben“ hat nun den Inhalt:

```
[('Kirsche', 'rot'), ('Banane', 'gelb'), ('Apfel', 'grün')]
```

Der Name der Funktion **zip** leitet sich aus dem englischen Wort für Reißverschluss (zipper) ab. Genau wie beim Reißverschluss wird abwechselnd jeweils ein Element aus der einen und aus der anderen Liste miteinander kombiniert.

Der Aufruf von **zip()** erzeugt noch keine neue Sequenz, sondern nur ein Zip-Objekt, das Daten aus den ursprünglichen Listen ausliest. Über dieses können wir in einer Schleife iterieren oder es durch **list()** in eine neue Liste umformen.

Vorsicht: dass der alleinige Aufruf von **zip()** noch keine neuen Listen erzeugt, hat zur Konsequenz, dass Änderungen in zuvor „gezippten“ Listen auch in den danach von **zip()** erzeugten Daten zu sehen sind.

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]

>>> c = zip(a,b)

>>> a[1] = 999

>>> list(c)
[(1, 4), (999, 5), (3, 6)]
```

Ein Gegenstück zu **zip** in Form eines eigenen Unzip-Objekts gibt es nicht[1]. Stattdessen verwenden wir **zip** auch zum Entpacken:

```
Obst, Farbe = zip(*Obstfarben)
```

Die Zip-Funktion ermöglicht es sogar, gleichzeitig zwei getrennte Sequenzen zu sortieren, beispielsweise, weil deren jeweilige Elemente einen Bezug zueinander haben. Die dritte Zeile im folgenden Beispiel fügt die beiden Listen **Obst** und **Farbe** zusammen, sortiert das Ergebnis und trennt die sortierte Liste wieder in zwei einzelne Listen:

```
Obst = ["Kirsche", "Banane", "Apfel"]
Farbe = ["rot", "gelb", "grün"]
Obst, Farbe = zip(*sorted(zip(Obst, Farbe)))
```

Das Ergebnis sieht nun so aus:

```
>>> Obst
('Apfel', 'Banane', 'Kirsche')
>>> Farbe
('grün', 'gelb', 'rot')
```

5.15.13 Funktionen für Sequenzen

Auf Sequenzen können wir eine Reihe von nützlichen Standardfunktionen anwenden. Die in der folgenden Tabelle aufgeführten Funktionen liefern die in der rechten Spalte beschriebenen Rückgabewerte, wenn ihnen als Argument eine Sequenz (S) übergeben wird:

Funktion	Rückgabewert
len(S)	Anzahl der Elemente in der Sequenz

1 Wer unbedingt eine Unzip-Funktion haben will, kann sie sich selbst schreiben:
```
def unzip(x):
    return zip(*x)
```

Funktion	Rückgabewert
sum(S)	Summe der Elemente, wenn alle Elemente aus Zahlenwerten bestehen
min(S)	Kleinstes Element
max(S)	Größtes Element
sorted(S)	Erzeugt aus den Elementen der Sequenz eine neue Liste in sortierter Form
all(S)	**True**, wenn alle Elemente **True** oder gleichwertig
any(S)	**True**, wenn irgendein Element **True** oder gleichwertig

5.15.14 Löschen von Sequenzen

Sequenzen können recht große Gebilde werden und eine Menge RAM belegen. Erzeugen wir zu viele große Objekte, beginnt das Betriebssystem, Inhalte aus dem RAM auf die Festplatte auszulagern. Dabei wird der Rechner unter Umständen bis zur Unbedienbarkeit verlangsamt.

Damit uns das nicht passiert, können wir mit dem Schlüsselwort **del** Sequenzen oder Teile davon löschen. Ein mit **del** gelöschtes Objekt verliert auch seinen Namen.

```
>>> Tiere = ["Hund", "Schnitzel", "Fußball", "Maus"]
>>> Student = {'Name': 'Kevin', 'Alter': 21, 'Hobby':
'Tauchen'}
>>> Gigabyte = list(range(111_111_100))

>>> del Tiere[1:3]
>>> Tiere
['Hund', 'Maus']

>>> del Student["Hobby"]
>>> Student
{'Name': 'Kevin', 'Alter': 21}
```

```
>>> del Gigabyte
>>> Gigabyte
Traceback (most recent call last):
  File "<pyshell#127>", line 1, in <module>
    Gigabyte
NameError: name 'Gigabyte' is not defined
```

In den meisten Fällen wird es niemals nötig sein, dass wir uns um das Ausgehen des freien Speichers Gedanken machen müssen. Objekte, die tatsächlich Speicher in der Größenordnung von mehreren Gigabyte belegen, sind in alltäglichen Projekten eher selten.

5.15.15 Methoden von Listen

Es gibt Funktionen in Python, die wir nur auf Listen anwenden können. Solche an bestimmte Objekte gebundene Funktionen nennen wir „Methoden“ dieser Objekte. Sie werden mit der Schreibweise **`Objektname.Methodenname(Argumente)`** aufgerufen.

In der folgenden Tabelle sind einige Methoden aufgeführt, die uns die Arbeit mit Listen erleichtern.

Methode und Beschreibung	Anwendungsbeispiel

`.append(Element)`
 	
Hängt ein neues Element an eine Liste an.
	`Baustoffe = ["Marmor", "Stein", "Eisen"]`
	`Baustoffe.append("Holz")`
	`print(Baustoffe)`
	
	↓
	
	`['Marmor', 'Stein', 'Eisen', 'Holz']`

Methode und Beschreibung	Anwendungsbeispiel

`.count(Element)`
 	
Gibt zurück, wie oft ein bestimmter Elementwert in einer Liste vorkommt.
	`Obst = ["Apfel", "Apfel", "Feige",`
	`  "Birne", "Apfel", "Banane"]`
	`print(Obst.count("Apfel"))`
	`print(Obst.count("Zucchini"))`
	
	↓
	
	`3`
	`0`

`.index(Element)`
 	
Gibt die Position der ersten Fundstelle eines Elementwerts zurück.
 	
Enthält die Liste kein passendes Element, tritt ein Fehler vom Typ **`ValueError`** auf.
	`Ziffern = ["null", "eins", "zwei",`
	`  "drei", "vier"]`
	`print(Ziffern.index("zwei"))`
	
	↓
	
	`2`

Methode und Beschreibung	Anwendungsbeispiel

`.insert(index, Element)`
 	
Fügt ein neues **`Element`** an der Stelle **`index`** in die Liste ein. Dahinter liegende Listenelemente werden um eine Position verschoben.
 	
Wird eine Indexposition ausserhalb der bestehenden Liste adressiert, wird das neue **`Element`** bei positiven **`index`**-Werten an die Liste angehängt und bei negativen **`index`**-Werten am Anfang der Liste eingefügt.
	`Fische = ["Hering", "Makrele", "Scholle"]`
	`Fische.insert(2, "Blauwal")`
	`print(Fische)`
	
	↓
	
	`['Hering', 'Makrele', 'Blauwal', 'Scholle']`

`.pop(index)`
 	
Gibt das Element an der Stelle **`index`** zurück und entfernt es aus der Liste. Dahinter liegende Elemente rücken um eine Position auf. Wird für **`index`** kein Wert übergeben, entnimmt **`.pop()`** das letzte Element der Liste.
 	
Ist die Liste leer oder wird eine nicht vorhandene Indexposition adressiert, tritt ein Fehler vom Typ **`IndexError`** auf.
	`Fische = ["Hering", "Makrele", "Blauwal", "Scholle"]`
	`Säugetier = Fische.pop(2)`
	`print(Fische)`
	`print(Säugetier)`
	
	↓
	
	`['Hering', 'Makrele', 'Scholle']`
	`Blauwal`

Methode und Beschreibung	Anwendungsbeispiel

`.remove(Element)`
 	
Löscht das erste Vorkommen eines Elementwertes aus einer Liste.
	`Fische = ["Hering", "Makrele",`
	`  "Blauwal", "Scholle"]`
	`Fische.remove("Blauwal")`
	`print(Fische)`
	
	↓
	
	`['Hering', 'Makrele', 'Scholle']`

`.reverse()`
 	
Kehrt die Reihenfolge einer Liste um.
	`Schlagzeile = ["Hund", "beißt",`
	`  "Mann"]`
	`Schlagzeile.reverse()`
	`print(*Schlagzeile)`
	
	↓
	
	`Mann beißt Hund`

`.sort()`
 	
Sortiert die Elemente einer Liste.
 	
Die Listenelemente müssen alle vom selben Typ sein.
	`Zahlen = [3, 1, 72, 21, 0, 20, 34,`
	`  2, 12, 96, 44, 61]`
	`Zahlen.sort()`
	`print(Zahlen)`
	
	↓
	
	`[0, 1, 2, 3, 12, 20, 21, 34, 44, 61]`

Die Methoden **`.index()`** und **`.count()`** sind hierbei die einzigen, die auch auf Tupel angewendet werden können.

5.16 Anwendung von Listen: Vektoren

Vektoren können in Python ganz einfach als Listen von Zahlenwerten dargestellt werden.

Anstelle der mathematischen Notation ...

$$\vec{a} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

... schreiben wir in Python:

```
a = [1, 2, 3, 4]
```

5.16.1 Vektoraddition

$$\vec{c} = \vec{a} + \vec{b}$$

Um Vektoren zu addieren, benötigen wir eine Schleife, die nacheinander alle Elemente zweier Listen (die Summandenvektoren) addiert und daraus eine dritte Liste (den Summenvektor) zusammensetzt.

Angenommen, wir haben die Listen A und B, die jeweils 5 Elemente aufweisen.

```
A = [1, 2, 3, 4, 5]
B = [20, 30, 40, 50, 60]
```

Um eine neue Liste C zu erhalten, könnten wir zunächst eine leere Liste anlegen ...

```
C = []
```

... und anschließend schrittweise die Summe der fünf Listenelemente **A[0] + B[0]** bis **A[4] + B[4]** als neues Listenelement (daher mit eckigen Klammern umschlossen) an die im Verlauf der Abarbeitung immer länger werdende Liste C anhängen.

```
for i in range(5):
    C.append(A[i]+B[i])
```

Dies entspricht der klassischen Vorgehensweise in den meisten Programmiersprachen. In Python geht es eleganter, kürzer und übersichtlicher, wenn wir uns die Möglichkeiten der funktionalen Listenerzeugung zunutze machen. Das Zip-Objekt, dem wir gerade in Kapitel 5.15.12 begegnet sind, hilft uns dabei, indem es Tupel aus den korrespondierenden Elementen beider Listen A und B erzeugt. Die Summen dieser Tupel bauen wir nun zu einer neuen Liste zusammen:

```
C = [a+b for a, b in zip(A, B)]
```

5.16.2 Skalarprodukt

Das Skalarprodukt zweier Vektoren ist die Summe der Produkte ihrer korrespondierenden Elemente.

Auch diese Berechnung können wir entweder als klassische Schleife formulieren, indem eine vor der Schleife auf null gesetzte Summe schrittweise um das Produkt der einzelnen Listenelemente erhöht bzw. verringert wird ...

```
A = [1, 2, 3, 4, 5]
B = [20, 30, 40, 50, 60]
SP = 0
for i in range(5):
    SP += A[i] * B[i]
```

... oder wir geben abermals der funktionalen Listenerzeugung den Vorzug, indem wir zunächst einen Generatorausdruck für alle Einzelprodukte formulieren und anschließend deren Summe über Pythons eingebaute Funktion **sum()** ermitteln:

```
A = [1, 2, 3, 4, 5]
B = [20, 30, 40, 50, 60]
SP = sum(a*b for a, b in zip(A, B))
```

5.16.3 Formatierte Ausgabe eines Vektors

Mit der in Kapitel 5.21.5 auf Seite 209 vorgestellten formatierten Zahlenausgabe mit Platzhaltern können wir die einzelnen Zahlenwerte eines Vektors gut lesbar geordnet untereinander ausgeben.

Die Funktion **druckeVektor** gibt die Elemente eines als Argument übergebenen Vektors mit jeweils 4 Nachkommastellen und 10 Stellen Gesamtbreite untereinander aus.

```
v = [2/7, 3.14, 0, 1000/13]

def druckeVektor(v):
    for x in v:
        print(f"{x:10.4f}")

druckeVektor(v)
```

Resultat:

```
    0.2857
    3.1400
    0.0000
   76.9231
```

Haben wir da gerade eine eigene Funktion definiert? Das sollten wir uns näher anschauen.

5.17 Eigene Funktionen definieren

In einem Programm mehrmals benutzte Formeln oder Programmteile können wir als Funktion definieren. Die Lesbarkeit unseres Programms erhöht sich dadurch erheblich, und wir vermeiden Redundanz durch sich mehrfach wiederholenden Programmcode.

Die Funktionsdefinition erfolgt durch das Schlüsselwort **def**, gefolgt vom Funktionsnamen, einem Klammerpaar und einem Doppelpunkt. Das Klammerpaar enthält eine Reihe von nur innerhalb der Funktion sichtbaren Variablen, in denen für die Dauer der Ausführung der Funktion die der Funktion übergebenen Eingangswerte gespeichert sind.

Auf die Funktionsdefinition folgt ein eingerückter Programmblock, der mit dem Schlüsselwort **return** und dem zurückzugebenden Wert oder den zurückzugebenden Werten abgeschlossen wird.

```
>>> def addiere(a, b):
        "Gibt das Ergebnis der Addition von a und b zurück."
        c = a + b
        return c

>>> addiere(17, 4)
21
```

Die Funktion **addiere(a, b)** nimmt genau zwei Eingangswerte entgegen, nennt diese **a** und **b** und gibt das Ergebnis der Addition dieser beiden Werte zurück.

In Python müssen wir uns keine besonderen Gedanken über die Typen der Eingangswerte machen. Solange die mit den übergebenen Werten durchgeführten Operationen gültig sind, können wir beliebige Datentypen an die Funktion übergeben.

```
>>> addiere("super", "klasse")
'superklasse'
```

Unsere Additionsfunktion funktioniert, wie man sieht, sowohl mit Zahlenwerten als auch mit Zeichenketten.

Eine einmal definierte Funktion kann immer wieder aufgerufen werden, indem im Programmtext ihr Name, gefolgt von den jeweiligen Eingangswerten in Klammern, verwendet wird.

An der Farbe des Funktionsnamens erkennen wir in IDLE, ob die Funktion zu den eingebauten Funktionen wie **print** oder **sorted** gehört, oder ob sie wie **addiere** neu definiert wurde. Die Namen eingebauter Funktionen werden purpurfarben dargestellt, die Namen neuer Funktionen in der Kopfzeile der Definition blau, ansonsten schwarz.

Die Textkonstante in der ersten Zeile unserer Funktion hat eine besondere Bedeutung. Sie ist deren offizieller Hilfetext. Mit der Funktion **help()** kann der Hilfetext zu einer Funktion abgerufen werden.

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

addiere(a, b)
    Gibt das Ergebnis der Addition von a und b zurück.
```

5.17.1 Eingangswerte (Argumente)

Die Eingangswerte einer Funktion heißen auch „Argumente“ oder „Parameter“ dieser Funktion. Alle Argumente müssen beim Aufruf einer Funktion in der richtigen Reihenfolge angegeben werden. Sind sie unvollständig, gibt Python eine Fehlermeldung aus.

```
>>> def addiere(a, b, c):
        return a + b + c

>>> addiere(1, 2, 3)
6
>>> addiere(17, 4)
```

```
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    addiere(17, 4)
TypeError: addiere() missing 1 required positional argument:
'c'
```

5.17.2 Vorbelegte Eingangswerte

Eingangswerte einer Funktion können mit einem Standardwert vorbelegt werden. Beim Aufruf der Funktion dürfen diese vorbelegten Werte, von der rechten Seite beginnend, weggelassen werden.

```
>>> def addiere(a=0, b=0, c=0):
        return a + b + c

>>> addiere(1, 2, 3)
6
>>> addiere(1, 2)
3
>>> addiere(1)
1
>>> addiere()
0
```

Im gezeigten Beispiel werden beim ersten Aufruf alle drei Funktionsvariablen verwendet, im zweiten Aufruf nur **a** und **b** (**c** bleibt auf null), im dritten Aufruf nur **a** und im vierten Aufruf wird überhaupt kein Parameter übergeben – alle drei Eingangswerte der Funktion bleiben auf ihrer Voreinstellung null.

5.17.3 Beliebig viele Argumente

Wenn wir dem letzten (oder einzigen) Argumentnamen in der Funktionsdefinition ein Sternchen * voranstellen, so wird der Funktion ein Tupel mit beliebig vielen Werten übergeben.

Innerhalb der Funktion kann dieses Tupel beispielsweise durch eine Schleife Element für Element abgearbeitet werden

```
>>> def addiere(*args):
        summe = 0
        for zahl in args:
            summe += zahl
        return summe

>>> addiere()
0

>>> addiere(1,2,3)
6

>>> addiere(100, 231, 675, 76234, 11384)
88624

>>> addiere(*range(101))
5050
```

Das der Funktion **range** vorangestellte Sternchen beim letzten Aufruf der Funktion ist das Gegenstück zum Sternchen innerhalb der Funktion. Beim Aufruf zerlegt es eine Sequenz in einzelne Werte, wogegen es innerhalb der Funktion Einzelparameter zu einem Tupel vereint.

5.17.4 Reihenfolge von Funktionsargumenten

Beim Aufruf einer Funktion müssen wir ihre Argumente üblicherweise genau in der Reihenfolge angeben, die bei der Definition der Funktion festgelegt wurde.

Ein Abweichen von dieser Regel ist möglich; dann müssen wir aber explizit die Namen der Funktionsargumente angeben.

```
>>> def dividiere(zähler, nenner):
        return zähler/nenner

>>> dividiere(3, 4)
0.75

>>> dividiere(4, 3)
1.3333333333333333

>>> dividiere(nenner=4, zähler=3)
0.75
```

Auch solche Schlüssel-Werte-Paare können wir gesammelt verarbeiten. Dazu stellen wir bei der Funktionsdefinition dem „Sammelargument“ ein doppeltes Sternchen voran. In der Funktion kommt dann ein Dictionary an:

```
>>> def test(**kwargs):
        print(kwargs)

>>> test(a=1, b=2, c=3)
{'a': 1, 'b': 2, 'c': 3}
```

Eine Funktion kann auch beide Arten von Argumenten verarbeiten:

```
>>> def test(*args, **kwargs):
        print(args)
        print(kwargs)

>>> test(18, 20, 2, 0, 4, a=1, b=2, c=3)
(18, 20, 2, 0, 4)
{'a': 1, 'b': 2, 'c': 3}
```

5.18 Sichtbarkeit von Variablen

Auf Variablen, die innerhalb einer Funktion eingeführt wurden, kann von außen nicht zugegriffen werden. Wir sprechen davon, dass der Namensraum dieser Variablen auf die Funktion begrenzt ist. Wenn solche lokalen Variablen dieselben Namen tragen wie Variablen, die vor Ausführung der Funktion bereits außerhalb dieser Funktion angelegt wurden, so bleiben die außerhalb der Funktion verwendeten Variablen unverändert.

```
>>> def Demo(A, B):
        C = A + B
        A = 22
        B = 33
        return A, B, C

>>> A = B = C = 123
>>> A, B, C
(123, 123, 123)

>>> Demo(10, 20)
(22, 33, 30)

>>> A, B, C
(123, 123, 123)
```

Das Programmbeispiel zeigt eine Funktion namens **Demo**, die drei lokale Variablen **A**, **B** und **C** verwendet. Diese Variablen sind völlig unabhängig von den namensgleichen Variablen **A**, **B** und **C** außerhalb der Funktion. Deren Wert (123) bleibt auch nach dem Aufruf der Funktion unverändert.

5.19 Klassen und Objekte

Wir sind bereits einer ganzen Menge unterschiedlicher Objekte begegnet und erinnern uns zum Beispiel an Listen, Tupel oder Zeichenketten. Jedes dieser Objekte gehört einer bestimmten Klasse an. Objekte einer Klasse verfügen über gemeinsame Attribute und Methoden.

Wie können in Python eigene Klassen definieren und von diesen anschließend einzelne Objekte ableiten.

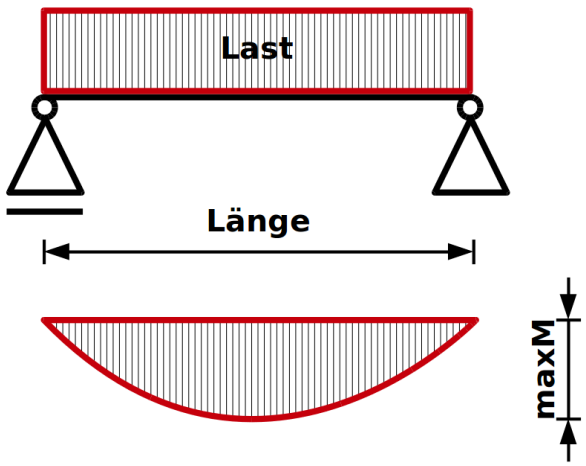


Abb. 76: Vorbild für ein Objekt: Ein Einfeldträger

Angenommen, wir wollen ein Programm schreiben, das mit verschiedenen Einfeldträgern umgehen soll. Jeder dieser Einfeldträger soll über zwei Attribute verfügen: **`Länge`** und **`Last`**. Weiterhin sollen unsere Einfeldträger-Objekte in der Lage sein, Auskunft über ihr Biegemoment **`maxM`** zu geben.

Mit dem Schlüsselwort **`class`** definieren wir die neue Klasse **`Einfeldträger`** und legen darin die beiden Attribute **`Länge`** und **`Last`** an.

```
class Einfeldträger:
    Länge = 0
    Last = 0
```

Schon sind wir in der Lage, durch Aufruf von **`Einfeldträger()`** neue Objekte der Klasse **`Einfeldträger`** zu erzeugen und diese einer Variable zuweisen. So eine Variable bezeichnen wir als „Instanz“ dieser Klasse.

Die öffentlich sichtbaren Attribute des neuen Objektes können wir mit der Konstruktion **`Objektname.Attributname`** wie gewöhnliche Variablen ansprechen.

```
T = Einfeldträger()
T.Länge = 5
T.Last = 2.5
```

Bevor wir unseren Einfeldträger dazu bringen, sein eigenes Biegemoment auszurechnen, schauen wir uns die Attribute eines Objektes noch etwas genauer an.

5.19.1 Attribute von Objekten

Als Attribute bezeichnen wir die innerhalb eines Objektes verwendeten Variablen. Bei unserem Einfeldträger würden die Attribute **`Last`** und **`Länge`** zweckmäßigerweise zwei Gleitkommavariablen sein.

Viele Informatiker vertreten die Ansicht, dass in der objektorientierten Programmierung niemals direkt auf die Attribute eines Objektes zugegriffen werden dürfe, sondern dies ausschließlich über die Methoden einer Klasse zu erfolgen habe. Es gibt viele gute Gründe[1], sich bei umfangreichen Projekten streng an dieses Geheimnisprinzip der Datenkapselung zu halten.

In Python sind Attribute zunächst einmal öffentlich. Wir können aber durch Voranstellen eines Unterstrichs vor den Attributnamen zum Ausdruck bringen, dass der direkte Zugriff zumindest unerwünscht ist. Lassen wir einem Attributnamen gar mit einem zweifachen Unterstrich beginnen, so ist das so benannte Attribut tatsächlich von außen unter diesem Namen unerreichbar.

1 Siehe http://de.wikipedia.org/wiki/Datenkapselung_(Programmierung)

```
>>> class Geheimnisträger:
        erstes_Attribut = "öffentlich"
        _zweites_Attribut = "mit Vorsicht zu behandeln"
        __drittes_Attribut = "streng geheim"


>>> Geheimnisträger.erstes_Attribut
'öffentlich'

>>> Geheimnisträger._zweites_Attribut
'mit Vorsicht zu behandeln'

>>> Geheimnisträger.__drittes_Attribut
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    Geheimnisträger.__drittes_Attribut
AttributeError: type object 'Geheimnisträger' has no
attribute '__drittes_Attribut'
```

Öffentliche Attribute können einem Objekt und sogar einer Klasse auch nachträglich noch hinzugefügt werden. Wir könnten ohne weiteres für unseren Einfeldträger noch ein Attribut **`Farbe`** einführen und dies entweder an ein bestehendes Objekt oder an die ganze Klasse binden. Mit **`T.Farbe = "grün"`** würden wir das konkrete Objekt **`T`** um das Attribut **`Farbe`** bereichern und mit **`Einfeldträger.Farbe = "rot"`** würden wir allen neuen und bereits existierenden Objekten der Klasse **`Einfeldträger`** das Attribut **`Farbe`** hinzufügen, falls sie es noch nicht besitzen.

```
>>> T = Einfeldträger()
>>> T.Farbe = "grün"

>>> T2 = Einfeldträger()

>>> Einfeldträger.Farbe = "rot"

>>> T3 = Einfeldträger()

>>> T.Farbe
'grün'
>>> T2.Farbe
'rot'
>>> T3.Farbe
'rot'
```

Wegen der Möglichkeit, damit auf einfache Weise ein schwer nachvollziehbares Chaos anzurichten, sollten Sie auf das nachträgliche Hinzufügen von Attributen zu Objektklassen weitestgehend verzichten.

5.19.2 Methoden von Objekten

Als „Methoden“ bezeichnen wir die Funktionen einer Objektklasse. Sie werden gemeinsam mit der Klasse definiert.

Damit eine Methode auf die Attribute eines Objektes zugreifen kann, muss sie wissen, um welches konkrete Objekt es sich handelt. In Python wird dazu traditionell der Bezeichner **`self`** gewählt, wenn es sich um die Attribute desselben Objektes handelt, zu dem auch die jeweilige Methode gehört.

Schauen wir uns dazu einmal noch einmal unseren Einfeldträger an:

```
>>> class Einfeldträger:
        Länge = 0
        Last = 0
        def maxM(self):
            return self.Last * self.Länge ** 2 / 8
```

Wenn wir einem aus dieser Klasse abgeleiteten Objekt die Attribute **Länge** und **Last** zuweisen, gibt die Methode **maxM** uns das maximale Biegemoment zurück:

```
>>> T = Einfeldträger()
>>> T.Länge = 5
>>> T.Last = 2.5
>>> T.maxM()
7.8125
```

Anfangs ist es bestimmt etwas gewöhnungsbedürftig, dass der Bezeichner **self** nur innerhalb einer Klassendefinition aufgeführt wird, nach außen aber nicht in Erscheinung tritt.

Der Sinn des „self-Bezuges“ wird uns vielleicht ein wenig klarer, wenn wir uns vorstellen, die Funktion **maxM()** sei keine Methode der Objektklasse **Einfeldträger**, sondern unabhängig von dieser definiert. Dann müssten wir bei jedem Aufruf explizit sagen, auf welches Objekt die Funktion anzuwenden sei:

```
>>> def maxM(Objekt):
        return Objekt.Last * Objekt.Länge ** 2 / 8

>>> T = Einfeldträger()
>>> T.Länge = 5
>>> T.Last = 2.5
>>> maxM(T)
```

```
7.8125
```

5.19.3 Die Methode __init__

Wenn in einer Klassendefinition eine Methode auftaucht, die den Namen **__init__** mit jeweils zwei Unterstrichen vorn und hinten trägt, so wird diese Initialisierungsfunktion direkt nach dem Erzeugen eines neuen Objekts dieser Klasse aufgerufen.

Diese spezielle Methode dient dazu, an das Objekt übergebene Argumente anzunehmen und das neue Objekt in einen definierten Zustand zu versetzen. Wie bei allen Methoden einer Objektklasse wird das Objekt selbst wieder mit **self** referenziert.

Da die Funktion **__init__** das neue Objekt gewissermaßen „konstruiert“, wird sie „Konstruktor“ der Klasse genannt. Die erzeugten Objekte nennen wir „Instanzen“ der Klasse.

```
>>> class Einfeldträger:
        def __init__(self, Länge, Last):
            self.Länge = Länge
            self.Last = Last
        def maxM(self):
            return self.Last*self.Länge**2/8

>>> T = Einfeldträger(5, 2.5)
>>> T.maxM()
7.8125
```

Methoden mit zwei Unterstrichen am Anfang und Ende des Namens haben in Python eine spezielle Bedeutung. Sie gelten als sogenannte „magische Methoden“, mit denen das Verhalten von Objekten bis tief in die Sprachsyntax hinein beeinflusst werden kann.

5.19.4 Vererbung

Wir können neue Klassen von bereits vorhandenen Klassen ableiten, um sie beispielsweise um neue Methoden zu erweitern, ohne dabei den ganzen schon vorhandenen Code noch einmal schreiben zu müssen. Bei der Definition unserer neuen Klasse geben wir dazu an, welche vorhandene Klasse wir „beerben“ wollen. Unsere neue Klasse „Deckenträger“ soll beispielsweise alles können, was auch die schon weiter oben definierte Klasse „Einfeldträger“ vermag und soll zusätzlich noch die Auflagerkräfte A_v und B_v zurückgeben.

Um eine Klasse von einer anderen Klasse abzuleiten, übergeben wir den Namen der übergeordneten Klasse als Argument:

```
class Deckenträger(Einfeldträger):
```

Unsere „magische“ Initialisierungsfunktion hat nun allerdings noch die Aufgabe, alle Argumente, die bei der Objekterzeugung mitgegeben werden (im Beispiel sind das die beiden Argumente **`Last`** und **`Länge`**) an die übergeordnete Funktion durchzureichen.

Wir bauen uns eine Durchreiche

Hierzu hat sich eine Schreibweise etabliert, die sicherstellt, dass ganz unabhängig von der Zahl der übergebenen Argumente und ganz unabhängig davon, ob diese mit oder ohne Schlüsselworte angegeben wurden, alle Argumente unverändert weitergereicht werden. Unsere Initialisierungsmethode leiten wir dazu so ein:

```
def __init__(self, *args, **kwargs):
```

Das Tupel **`args`** enthält dabei alle einzelnen Argumente, beispielsweise **`(5, 2.5)`**, und das Dictionary **`kwargs`** enthält alle übergebenen Schlüssel-Werte-Paare, beispielsweise **`{Länge: 5, Last: 2.5}`**. Welche Inhalte genau in den beiden Sequenzen enthalten sind, müssen wir zum Glück gar

nicht untersuchen; wir reichen beides einfach an die übergeordnete Klasse weiter. Deren Namen brauchen wir auch nicht explizit anzugeben, sondern wir verwenden dazu die Funktion **super()**[1].

Der folgende Aufruf sieht daher komplizierter aus, als er in Wirklichkeit ist. Tatsächlich bedeutet er nur, dass alle unserer Klasse übergebenen Argumente unverändert an die übergeordnete Klasse durchgereicht werden:

```
super().__init__(*args, **kwargs)
```

Nachdem diese Formalien erledigt sind, können wir uns um unsere eigene Klasse kümmern. Diese soll ja schließlich noch die Auflagerkräfte liefern. Das können wir nun mit einer einzigen Codezeile erreichen:

```
self.Av = self.Bv = self.Länge * self.Last / 2
```

Die gesamte Klassendefinition sieht nun also so aus:

```
class Deckenträger(Einfeldträger):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.Av = self.Bv = self.Länge*self.Last/2
```

In der IDLE-Shell können wir die neue Klasse gleich ausprobieren:

```
>>> D = Deckenträger(10, 5)
>>> D.Av
25.0
>>> D.Bv
25.0
>>> D.maxM()
62.5
```

1 Das lateinische Wort „super“ heißt soviel wie „über“ oder „oberhalb“. Die Super-Funktion ist also lediglich die übergeordnete Funktion und besitzt nicht notwendigerweise besonders atemberaubende Fähigkeiten.

Wie sollten nun noch darüber nachdenken, welche Konsequenzen es hat, dass **`maxM`** als Methode formuliert wurde und **`Av`** sowie **`Bv`** als Attribute. Was passiert bei einer Änderung der Attribute **`Länge`** und **`Last`**? Bauen Sie die Klassendefinitionen doch mal so um, dass die Auflagerkräfte mit zwei Methoden **`Av()`** und **`Bv()`** abgerufen werden können ...

5.20 Eigene Module

Wollen wir unsere selbstgeschriebenen Funktionen und Objektklassen in verschiedenen Programmen einsetzen, so ist es nicht nötig, deren Definitionen immer wieder in den Quelltext neuer Programme hineinzukopieren. Stattdessen binden wir die Python-Datei, in der sich die gewünschte Funktionsdefinition befindet, als Modul in unser neues Programm ein.

Angenommen, wir haben soviel Gefallen an unserer Funktion **`addiere()`** aus Kapitel 5.17 gefunden, dass wir sie zukünftig immer wieder verwenden wollen. Zusätzlich sollen außerdem noch die Funktionen **`subtrahiere()`**, **`multipliziere()`** und **`dividiere()`** zur ständigen Verfügung stehen.

Wir schreiben dann einfach alle Funktionsdefinitionen hintereinander in eine Datei, die wir zum Beispiel **`labermath.py`** nennen können.

Fortan stehen uns in allen neuen Programmen die Funktionen unseres ersten selbstgeschriebenen Python-Moduls zur Verfügung, sobald wir die Zeile **`from labermath import *`** in unser Programm aufgenommen haben.

Wenn wir die Funktionen immer sorgfältig mit Doc-Strings versehen, das sind Zeichenketten unmittelbar in der ersten Zeile einer Funktionsdefinition, dann stellt die Funktion **`help()`** sogar einen eigenen Hilfstext für unser Modul zusammen.

```
labermath.py - /home/martin/2019 WS 20/Informatik/Python-S...
File Edit Format Run Options Window Help
# Modul labermath
"Arithmetik für Geschwätzige"

def addiere(a, b):
    "Gibt die Summe von a und b zurück."
    return a + b

def subtrahiere(a, b):
    "Gibt die Differenz von a und b zurück."
    return a - b

def multipliziere(a, b):
    "Gibt das Produkt von a und b zurück."
    return a * b

def dividiere(a, b):
    """Gibt den Quotienten von a und b zurück.
    Rückgabewert für b == 0 ist nan (not a number
    """
    if b == 0:
        return float("nan")
    else:
        return a / b
Ln: 18  Col: 2
```

```
labermath-test.py - /home/martin/2019 WS 20/Inform...
File Edit Format Run Options Window Help
from labermath import *

x = 3
y = 4

print(f"{x} + {y} = {addiere(x,y)}")
print(f"{x} - {y} = {subtrahiere(x,y)}")
print(f"{x} * {y} = {multipliziere(x,y)}")
print(f"{x} / {y} = {dividiere(x,y)}")
```

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
3 + 4 = 7
3 - 4 = -1
3 * 4 = 12
3 / 4 = 0.75
>>> help("labermath")
Help on module labermath:

NAME
    labermath - Arithmetik für Geschwätzige

FUNCTIONS
    addiere(a, b)
        Gibt die Summe von a und b zurück.

    dividiere(a, b)
        Gibt den Quotienten von a und b zurück.
        Rückgabewert für b == 0 ist nan (not a number).

    multipliziere(a, b)
        Gibt das Produkt von a und b zurück.

    subtrahiere(a, b)
        Gibt die Differenz von a und b zurück.
Ln: 64  Col: 4
```

Abb. 77: Das Arithmetikmodul „labermath“

Damit das Betriebssystem unser Modul findet, ist es am einfachsten, es im selben Verzeichnis wie unsere anderen Pythonprogramme abzulegen.

Wenn Module in anderen Verzeichnissen liegen, müssen diese Verzeichnisse der Liste der von Python zu durchsuchenden Modulpfade hinzugefügt werden; siehe Kapitel 5.20.1.

Obwohl Importanweisungen an nahezu jeder beliebigen Stelle eines Programms stehen dürfen, ist es im Interesse der Lesbarkeit und Übersichtlichkeit des Quelltextes eine gute Idee, sie gesammelt an den Anfang zu setzen.

Übrigens können wir jedes beliebige Pythonprogramm einfach dadurch zu einem Modul machen, indem wir es in einem anderen Programm importieren. Allerdings würde das Programm dabei auch gleich gestartet[1].

1 Berühmtes Beispiel: **`import antigravity`**

Wenn wir jedoch nicht wollen, dass beim Import mehr passiert als dass Python die Funktionsdefinitionen „lernt“, können wir dem Ausführungsteil eine Abfrage voranstellen, die sicherstellt, dass die dort aufgeführten Programmbefehle beim Importieren des Moduls nicht ausgeführt werden.

Um Programmcode in einer Pythondatei nur dann auszuführen, wenn diese als Hauptprogramm direkt gestartet wird, nicht aber, wenn diese Datei als Modul eingebunden wird, können wir den betroffenen Programmbefehlen die Abfrage

```
if __name__ == "__main__":
```

voranstellen. Innerhalb eines Moduls enthält die Variable **__name__** stets den Namen des jeweiligen Moduls. Nur im Hauptprogramm lautet dieser Wert **"__main__"**.

5.20.1 Modulpfade

Ein Python-Interpreter, der einer **import**-Anweisung begegnet, durchsucht der Reihe nach bestimmte Verzeichnisse:

1. das Verzeichnis des aktuell laufenden Python-Programms
2. alle Modulverzeichnisse der Python-Installation
3. weitere an die Liste sys.path angehängte Verzeichnisse

Punkt 1 ist der Grund, warum es eine schlechte Idee ist, ein eigenes Python-Programm etwa **math.py** zu nennen. Alle anderen Pythonprogramme im selben Verzeichnis werden dann vermutlich ein Problem mit Sinus und Kosinus bekommen.

Die Liste der Modulverzeichnisse einer Python-Installation erhalten wir, indem wir uns die Variable **path** im Modul **sys** anschauen:

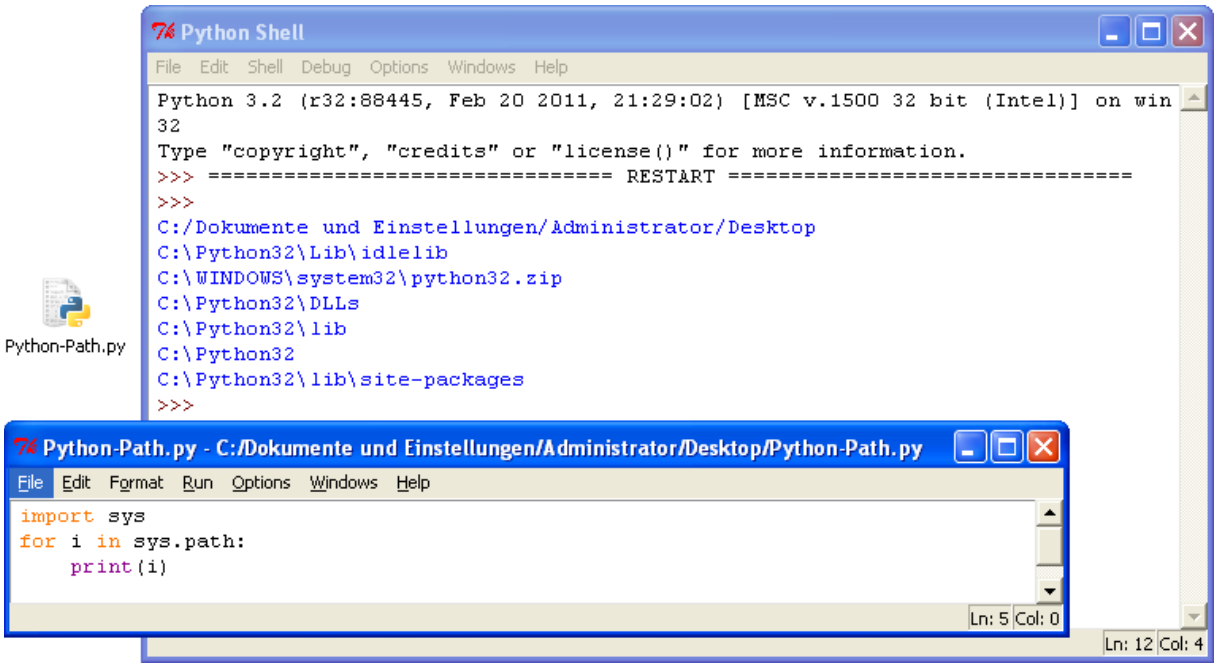


Abb. 78: Anzeige der Modulverzeichnisse unter Windows XP

5.20.2 Funktionsüberschreibungen

Die in Kapitel 5.7.1 angesprochenen Funktionsüberschreibungen können wir auch gezielt einsetzen. Angenommen, wir schreiben gerade ein Berechnungsprogramm, in dem sehr viele trigonometrische Berechnungen in Altgrad ausgeführt werden sollen.

Anstatt immer wieder in die für die Funktionen des Moduls **`math`** erforderliche Einheit Bogenmaß umzurechnen, definieren wir einfach einen Satz trigonometrischer Funktionen, die diese Umrechnung bereits eingebaut haben und verwenden diese stattdessen. Dazu schreiben wir alle Definitionen in ein neues Modul **`altgrad.py`**.

```
import math

def sin(x):
    return math.sin(math.radians(x))

def cos(x):
    return math.cos(math.radians(x))
```

```
def tan(x):
    return math.tan(math.radians(x))

def asin(x):
    return math.degrees(math.asin(x))

def acos(x):
    return math.degrees(math.acos(x))

def atan(x):
    return math.degrees(math.atan(x))

def atan2(dy,dx):
    return math.degrees(math.atan2(dy,dx))
```

Bei der Verwendung unserer neuen Altgradfunktionen müssen wir lediglich darauf achten, dass wir unser neues Modul **altgrad** erst *nach* dem Modul **math** importieren, damit die Überschreibungen wirksam werden.

```
from math import *
from altgrad import *

print("Trigonometrie")
print("~~~~~~~~~~~~~~")
print("Der Sinus von 30° ist",sin(30))
print("Der Kosinus von 0° ist",cos(0))
print("Der Tangens von 45° ist",tan(45))
print("Ein Sinuswert von 1 tritt bei",asin(1),"Grad auf.")
print("Bei 10% Steigung ist der Winkel",atan(0.1),"Grad.")
print("Eine Steigung von 1:2 entspricht",atan2(1,2),"Grad.")
```

Der Programmlauf zeigt, dass nun tatsächlich alle Winkelfunktionen mit Altgrad anstelle von Bogenmaß berechnet werden:

```
Trigonometrie
~~~~~~~~~~~~~
Der Sinus von 30° ist 0.49999999999999994
Der Kosinus von 0° ist 1.0
Der Tangens von 45° ist 0.9999999999999999
Ein Sinuswert von 1 tritt bei 90.0 Grad auf.
Bei 10% Steigung ist der Winkel 5.710593137499643 Grad.
Eine Steigung von 1:2 entspricht 26.56505117707799 Grad.
```

Wie man die vielen Nachkommastellen am einfachsten los wird, steht übrigens in Kapitel 5.21.5.

5.21 Zeichenketten

Zeichenketten sind Sequenzen von Buchstaben, Ziffern, Satz- und Sonderzeichen sowie Steuerzeichen von nahezu beliebiger Länge. Seit Python 3 können Zeichenketten alle weltweit verwendeten Unicode-Schriftzeichen enthalten.

5.21.1 Anführungszeichen in Zeichenketten

Zeichenkettenkonstanten dürfen wir wahlweise mit 'einzelnen' oder "doppelten" Anführungszeichen umgeben.

Den jeweils anderen Anführungszeichentyp können wir als gewöhnliches Zeichen im auszugebenden Text verwenden:

```
print("Conny's Frittenranch")
print('Sprich "Freund" und tritt ein!')
```

Wenn derselbe Typ von Anführungszeichen, der auch zur Umgrenzung verwendet wird, in der Zeichenkette enthalten sein soll, dann ist diesen Anführungszeichen jeweils ein Rückwärtsschrägstrich **** voranzustellen:

```
print("Man kann auch \"Backslash\" dazu sagen.")
```

5.21.2 Der Rückwärtsschrägstrich

Das Zeichen „\“ hat in Zeichenketten eine besondere Bedeutung.

Steht es vor den Kleinbuchstaben „t“, „n“ oder „r“, so werden „\t“, „\n“ und „\r“ durch Tabulatorzeichen, Zeilenumbruch und Wagenrücklaufzeichen ersetzt.

Die Kombinationen von „\x“ gefolgt von einer zweistelligen Hexadezimalzahl, „\u“ gefolgt von einer vierstelligen Hexadezimalzahl und „\U“ gefolgt von einer achtstelligen Hexadezimalzahl stehen für das Unicodezeichen mit der entsprechenden Codeposition.

Der Großbuchstabe „N“ hinter einem Rückwärtsschrägstrich erlaubt es, jedes Unicodezeichen über seinen genormten Namen[1] azusprechen. Das Eurozeichen € wird beispielsweise als „\N{EURO SIGN}“ umschrieben.

Um einen auszugebenden Rückwärtsschrägstrich oder ein Anführungszeichen innerhalb einer Zeichenkettenkonstante zu verwenden, stellen wir diesen Zeichen einen weiteren Rückwärtsschrägstrich voran.

```
>>> print("Die Zeichen \xe4, \xf6 und \xfc kann man "
          "auch\nals \"\\xe4\", \"\\xf6\" und \"\\xfc\" "
          "schreiben.")
Die Zeichen ä, ö und ü kann man auch
als "\xe4", "\xf6" und "\xfc" schreiben.
```

Der Rückwärtsschrägstrich wird gelegentlich auch im Deutschen mit seinem englischsprachigen Namen „Backslash“ bezeichnet.

5.21.3 Mehrzeilige Ausgabe

Umschließen wir eine Textkonstante mit jeweils drei Anführungszeichen, so darf der Text sich über mehrere Zeilen erstrecken und sowohl 'einzelne' als auch "doppelte" Anführungszeichen enthalten.

```
print("""
           SI-Maß: 1,12 m
           US-Maß: 3'8"
""")
```

Innerhalb der Anführungszeichen muss (darf) keine Rücksicht auf Pythons Einrückungen genommen werden. Dies macht Programmabschnitte, die mehrzeilige Zeichenkettenkonstanten innerhalb von Schleifen, Funktionsdefinitionen oder Fallunterscheidungen enthalten, mitunter schwer lesbar, sodass wir abwägen sollten, ob wir hier lieber der Einfachheit oder besser doch der Übersichtlichkeit den Vorzug geben.

1 Eine umfassende Liste von Unicode-Zeichen, ihrer Codepositionen und Namen findet sich unter anderem auf https://www.quackit.com/character_sets/unicode/.

```
>>> print("""
             __________
            |          |
            |  Python  |
            |    ist   |
            |  klasse! |
            |__________|
            (\__/)||
            (•ㅅ•)||
            /    づ
""")


             __________
            |          |
            |  Python  |
            |    ist   |
            |  klasse! |
            |__________|
            (\__/)||
            (•ㅅ•)||
            /    づ

>>> http://knowyourmeme.com/memes/sign-bunny
```

5.21.4 Zeichenketten-Methoden

Es gibt Funktionen in Python, die wir nur auf Zeichenketten anwenden können. Solche an bestimmte Objekte gebundene Funktionen nennen wir „Methoden“ dieser Objekte. Sie werden mit der Schreibweise **`Objektname.Methodenname(Argumente)`** aufgerufen.

Einige häufig verwendete Zeichenkettenmethoden sind im Folgenden aufgeführt.

.count(Suchtext)

gibt an, wie oft ein Suchtext in der zu durchsuchenden Zeichenkette vorkommt. Die einzelnen Fundstellen dürfen sich nicht überlappen.

```
A = "Elefantentanten"

A.count("nt")
3
```

.encode(Kodierung, Fehlerbehandlung)

wandelt eine Zeichenkette in eine Bytefolge um. Wenn Sie keine Kodierung angeben, verwendet Python die Standardkodierung UTF-8.

```
A = "Der Winkel α beträgt 45°."

A.encode()
b'Der Winkel \xce\xb1 betr\xc3\xa4gt 45\xc2\xb0.'
```

Wenn die Zeichenkette in der gewählten Kodierung nicht darstellbar ist (die unter Windows übliche Kodierung „Windows-1252“ kennt beispielsweise kein „α“), muss festgelegt werden, wie auf den Fehler reagiert werden soll. Das Standardverfahren ist **strict**, dabei bricht das Programm mit einer Fehlermeldung ab. Andere Verfahren sind **ignore**, **replace** und **xmlcharrefreplace**.

```
A.encode("windows-1252", errors="ignore")
b'Der Winkel  betr\xe4gt 45\xb0.'

A.encode("windows-1252", errors="replace")
b'Der Winkel ? betr\xe4gt 45\xb0.'

A.encode("windows-1252", errors="xmlcharrefreplace")
b'Der Winkel &#945; betr\xe4gt 45\xb0.'
```

.endswith(Suchtext)

gibt an, ob die Zeichenkette mit dem Suchtext endet.

```
A = "Dateiname.csv"

A.endswith(".csv")
True
```

.find(Suchtext)

gibt an, an welcher Stelle ein Suchtext in der zu durchsuchenden Zeichenkette erstmalig vorkommt. Wird der Suchtext nicht gefunden, lautet der Rückgabewert **-1**.

Soll nicht die erste, sondern die letzte Fundstelle zurückgegeben werden, verwendet man die Methode **.rfind()**.

```
A = "Elefantentanten"

A.find("nt")
5

A.find("xy")
-1

A.rfind("nt")
11
```

.isalnum()

gibt an, ob die Zeichenkette alphanumerisch ist, also nur aus Ziffern und Buchstaben besteht.

```
A = "123Test"
```

```
A.isalnum()
True

A = "123 Test"

A.isalnum()
False
```

.isalpha()

gibt an, ob die Zeichenkette nur aus Buchstaben besteht.

```
A = "Dortmund"

A.isalpha()
True

A = "Schalke04"

A.isalpha()
False
```

.isascii()

gibt an, ob die Zeichenkette nur aus ASCII-Zeichen besteht.

```
A = "Krefeld, Bochum, Alabama"

A.isascii()
True

A = "Münster, Αθήνα, Москва"

A.isascii()
False
```

.isdecimal()

gibt an, ob die Zeichenkette nur aus den Ziffern von 0–9 besteht.

```
A = "12345"

A.isdecimal()
True

A = "12.345"

A.isdecimal()
False
```

.join(iterierbares Objekt)

verwendet eine Zeichenkette, um damit die Elemente eines iterierbaren Objekts, beispielsweise einer Liste, zu verbinden. Die Elemente des iterierbaren Objektes müssen ebenfalls Zeichenketten sein.

```
A = ["Hund", "Katze", "Maus"]

", ".join(A)
'Hund, Katze, Maus'
```

.lower()

wandelt Buchstaben einer Zeichenkette in Kleinbuchstaben um.

```
A = "99 GRÜNE LuftBallons"

A.lower()
'99 grüne luftballons'
```

.replace(alt, neu)

ersetzt in einer Zeichenkette die Zeichenfolge **alt** gegen die Zeichenfolge **neu**:

```
A = "12,345 $"

A.replace("$", "€")
'12,345 €'

A.replace(",", ".")
'12.345 $'

A.replace(",", ".").replace("$", "€")
'12.345 €'
```

.split(Trennzeichen)

wandelt eine lange Zeichenkette in eine Liste aus kürzeren Zeichenketten um, wobei die erzeugten Listenelemente in der ursprünglichen Zeichenkette durch ein Trennzeichen oder eine bestimmte Zeichenfolge getrennt sein müssen.

```
A = "Hund;Katze;Maus"

A.split(";")
['Hund', 'Katze', 'Maus']
```

Wenn wir kein Trennzeichen angeben, trennt **split()** die zugehörige Zeichenkette an allen Leerzeichen, Tabulatorzeichen **\t** und Zeilenwechseln **\n**, dem sogenannten „Leerraum“ oder „Whitespace“[1].

1 Streng genommen gehören auch die historischen Steuerzeichen „Wagenrücklauf“ **\r** und „Seitenvorschub“ **\f** zu den Whitespace-Zeichen. Seitdem Drucker nicht mehr wie Schreibmaschinen funktionieren, sind diese Zeichen praktisch bedeutungslos. Das hält Microsoft Windows allerdings auch heute noch nicht davon ab, Textzeilen nicht nur wie andere Betriebssysteme mit einem Zeilenwechsel abzuschließen, sondern es stellt diesem immer auch noch einen Wagenrücklauf voran.

.startswith(Suchtext)

gibt an, ob die Zeichenkette mit dem Suchtext anfängt.

```
A = "IPE-400-Stahlträger"

A.startswith("IPE-")
True
```

.strip(abzustreifende Zeichen)

entfernt Zeichen am Anfang und Ende einer Zeichenkette. Wenn keine Zeichenkette aus abzustreifenden Zeichen übergeben wird, werden standardmäßig Leerzeichen, Tabulatorzeichen und Zeilenumbrüche[1] entfernt.

```
A = "/t/t# Inhalt/n"

A.strip()
'# Inhalt'

A.strip("/t/n #")
'Inhalt'
```

.upper()

wandelt Buchstaben einer Zeichenkette im Großbuchstaben um.

```
A = "99 grüne Luftballons"

A.upper()
'99 GRÜNE LUFTBALLONS'
```

1 … und alle anderen Whitespace-Zeichen …

5.21.5 Formatierung mit Platzhaltern

Bei der Ausgabe von Fließkommazahlen stellen wir gelegentlich fest, dass 15 Nachkommastellen zwar ganz schön für die Rechengenauigkeit sind, aber nicht unbedingt zur Lesbarkeit von numerischen Ergebnissen beitragen. Zwei oder drei Nachkommastellen reichen in der Praxis meistens aus.

Wenn wir mehrere Werte mit einem Aufruf der print-Funktion ausgeben wollen, geben wir sie üblicherweise als kommagetrennte Parameterliste an. Dabei können wir verschiedene Typen von Variablen, Konstanten oder ganzen Python-Ausdrücken beliebig mischen. Jedes Komma der Parameterliste wird standardmäßig durch ein Leerzeichen ersetzt.

```
a = "Länge"
b = 11/7
print("Die", a, "beträgt", b, "Meter.")

Die Länge beträgt 1.5714285714285714 Meter.
```

Die oben zu sehende Zahl hat 16 Nachkommastellen. Wir könnten sie zwar durch Aufruf der Funktion **`round(Zahlenwert, Stellenzahl)`** auf die gewünschte Stellenzahl runden, aber jeden einzelnen auszugebenden Zahlenwert in einem Programm durch Aufruf dieser Funktion zu runden, wäre viel zu umständlich. Python besitzt eine viel elegantere und vielseitigere Möglichkeit, Zahlenwerte mit fester Nachkommastellenzahl auszugeben: die Formatierung durch Platzhalter.

Ungewöhnlicherweise gibt es hier in Python zwei verschiedene Verfahren. Die ältere, zur Funktion *printf* in der Programmiersprache C kompatible, Formatierung ist sehr einfach aufgebaut, soll aber in kommenden Python-Versionen nicht mehr unterstützt werden. Sie wird in einem späteren Kapitel (5.21.9) nur noch der Vollständigkeit halber beschrieben. Stattdessen sollten wir die Zeichenkettenmethode **`.format()`** einsetzen, die durch die mit Python 3.6 neu eingeführten F-Strings noch komfortabler geworden ist.

5.21.6 F-Strings

F-Strings sind Zeichenkettenkonstanten, die Platzhalter für andere Konstanten, Variablen oder zusammengesetzte Python-Ausdrücke enthalten. Als Kennzeichnung für solche Platzhalter werden geschweifte Klammern verwendet. Ihren Namen haben F-Strings von dem ihnen vorangestellten Buchstaben „f“.

```
a = "Länge"
b = 11/7
print("Die", a, "beträgt", b, "Meter.")
print(f"Die {a} beträgt {b} Meter.")

Die Länge beträgt 1.5714285714285714 Meter.
Die Länge beträgt 1.5714285714285714 Meter.
```

Die Rundung der Zahlenwerte können wir einfach dadurch festlegen, dass wir innerhalb der geschweiften Klammern, hinter dem auszugebenden Wert, die gewünschte Stellenzahl angeben. Wollen wir beispielsweise die in der Variable **a** enthaltene Gleitkommazahl mit drei Nachkommastellen anzeigen, so schreiben wir **{a:.3f}**. Um anstelle der Nachkommastellen die Zahl der signifikanten Stellen auf drei festzulegen, lassen wir das „f“ innerhalb der geschweiften Klammer fort und schreiben: **{a:.3}**.

Wir können auch die Gesamtbreite der Zahl angeben. Das ist ganz praktisch, wenn wir in einer mehrzeiligen Ausgabe alle Dezimalpunkte ordentlich untereinander sehen wollen. Diese Breite tragen wir direkt hinter dem Doppelpunkt ein:

```
for i in range(7):
    a = 10**i
    b = a/7
    print(f"{a:8}/7 = {b:10.3f}")

       1/7 =      0.143
      10/7 =      1.429
     100/7 =     14.286
    1000/7 =    142.857
   10000/7 =   1428.571
  100000/7 =  14285.714
 1000000/7 = 142857.143
```

Der Dezimalpunkt und ein eventuell vorhandenes Vorzeichen werden dabei als eigenständige Zeichen mitgezählt.

Falls wir innerhalb eines f-Strings geschweifte Klammern verwenden möchten, die nicht als Platzhalter dienen, so schreiben wir sie doppelt: **{{** für eine öffnende und **}}** für eine schließende geschweifte Klammer.

5.21.7 Die Methode .format()

Ganz ähnlich wie bei den F-Strings funktioniert die Formatierung durch die Format-Methode von Zeichenketten. Hier geben wir die Variablennamen und Werte jedoch nicht direkt an, sondern hängen sie in Form einer Parameterliste an. Ob das übersichtlicher oder umständlicher als die Formatierung durch F-Strings ist, mögen Sie für sich entscheiden.

```
a = "Länge"
b = 11/7
print(f"Die {a} beträgt {b:.2f} Meter.")
```

```
print("Die {0} beträgt {1:.2f} Meter.".format(a,b))

Die Länge beträgt 1.57 Meter.
Die Länge beträgt 1.57 Meter.
```

Die Positionsangaben (hier 0 und 1) dürfen weggelassen werden, wenn jede Variable nur einmal in die Zeichenkette eingesetzt wird und die Reihenfolge der Verwendung dieselbe ist wie in der Parameterliste:

```
print("Die {} beträgt {:.2f} Meter.".format(a,b))

Die Länge beträgt 1.57 Meter.
```

5.21.8 Die Formatierungs-Mini-Sprache

Die Formatierungsmöglichkeiten von F-Strings und der .format-Methode gehen weit über die simple Festlegung von Nachkommastellen hinaus. Die Möglichkeiten sind so vielfältig, dass sie gelegentlich als „Mini-Sprache“ bezeichnet werden[1].

Einige Beispiele

Linksbündige, rechtsbündige, vorzeichengetrennte und zentrierte Ausrichtung mit „**<**“, „**>**“, „**=**“ und „**^**“ in einem acht Zeichen breiten Feld:

```
for x in [-10**(i-3) for i in range(10)]:
    print(f"{x:<8} | {x:>8} | {x:=8} | {x:^8}")

-0.001   |   -0.001 | -  0.001 |  -0.001
-0.01    |    -0.01 | -   0.01 |  -0.01
-0.1     |     -0.1 | -    0.1 |   -0.1
-1       |       -1 | -      1 |    -1
-10      |      -10 | -     10 |   -10
```

1 Format Specification Mini-Language: https://docs.python.org/3/library/string.html#format-specification-mini-language

```
-100     |     -100 | -    100 |   -100
-1000    |    -1000 | -   1000 |  -1000
-10000   |   -10000 | -  10000 |  -10000
-100000  |  -100000 | - 100000 | -100000
-1000000 | -1000000 | -1000000 | -1000000
```

Angabe eines Füllzeichens vor dem Ausrichtungszeichen:

```
for x in [-10**(i-3) for i in range(10)]:
    print(f"{x:~<8} | {x:÷>8} | {x:0=8} | {x:*^8}")

-0.001~~ | ÷÷-0.001 | -000.001 | *-0.001*
-0.01~~~ | ÷÷÷-0.01 | -0000.01 | *-0.01**
-0.1~~~~ | ÷÷÷÷-0.1 | -00000.1 | **-0.1**
-1~~~~~~ | ÷÷÷÷÷÷-1 | -0000001 | ***-1***
-10~~~~~ | ÷÷÷÷÷-10 | -0000010 | **-10***
-100~~~~ | ÷÷÷÷-100 | -0000100 | **-100**
-1000~~~ | ÷÷÷-1000 | -0001000 | *-1000**
-10000~~ | ÷÷-10000 | -0010000 | *-10000*
-100000~ | ÷-100000 | -0100000 | -100000*
-1000000 | -1000000 | -1000000 | -1000000
```

Vorangestellte Nullen, Vorzeichen und Leerzeichen:

```
for x in [(i-5)*10**(i-4) for i in range(10)]:
    print(f"{x: 08} | {x:-08} | {x:+08}")

-00.0005 | -00.0005 | -00.0005
-000.004 | -000.004 | -000.004
-0000.03 | -0000.03 | -0000.03
-00000.2 | -00000.2 | -00000.2
-0000001 | -0000001 | -0000001
 0000000 | 00000000 | +0000000
 0000100 | 00000100 | +0000100
 0002000 | 00002000 | +0002000
```

```
0030000 | 00030000 | +0030000
0400000 | 00400000 | +0400000
```

Tausendergruppierung mit Unterstrich, Komma oder gemäß der aktuellen Ländereinstellung des Betriebssystems (achten Sie in der rechten Spalte auf das Dezimalkomma!):

```
import locale
locale.setlocale(locale.LC_ALL, '')

for x in [10**(i-2) for i in range(10)]:
    print(f"{x:10} | {x:10_} | {x:10,} | {x:10n}")

      0.01 |       0.01 |       0.01 |       0,01
       0.1 |        0.1 |        0.1 |        0,1
         1 |          1 |          1 |          1
        10 |         10 |         10 |         10
       100 |        100 |        100 |        100
      1000 |      1_000 |      1,000 |      1.000
     10000 |     10_000 |     10,000 |     10.000
    100000 |    100_000 |    100,000 |    100.000
   1000000 |  1_000_000 |  1,000,000 |  1.000.000
  10000000 | 10_000_000 | 10,000,000 | 10.000.000
```

Ganze Zahlen können wir nicht nur im Dezimalsystem (Basis 10), im Binärsystem (Basis 2), Oktalsystem (Basis 8) oder Hexadezimalsystem (Basis 16) ausgeben, sondern wir können auch angeben, dass für den Platzhalter eines ganzzahligen Wertes das dem Wert entsprechende Unicodezeichen eingesetzt werden soll. All dies geben wir mit einem Buchstaben ganz am Ende des Platzhaltercodes an. Dabei steht der Buchstabe „b“ für eine Darstellung im Binärsystem, „o“ für eine Oktalzahl, „d“ (oder nichts) für eine Zahl im Dezimalsystem, „x“ für eine Hexadezimalzahl mit kleinen Buchstaben für die Ziffern jenseits der 9, „X“ für eine Hexadezimalzahl mit großen Buchstaben und „c“ für ein Unicodezeichen mit der entsprechenden Codeposition.

Das Gruppierungszeichen „_“ ordnet Zahlen des Binärsystems in Vierergruppen an. Der Modifikator „#“ sorgt dafür, dass Hexadezimalzahlen ein „0x“ und Binärzahlen ein „0b“ vorangestellt wird.

```
for i in (65, 66, 67, 216, 8730, 8734):
    print(f"{i:5d} | {i:019_b} | {i:04x} | {i:#06x} | {i:c}")

   65 | 0000_0000_0100_0001 | 0041 | 0x0041 | A
   66 | 0000_0000_0100_0010 | 0042 | 0x0042 | B
   67 | 0000_0000_0100_0011 | 0043 | 0x0043 | C
  216 | 0000_0000_1101_1000 | 00d8 | 0x00d8 | Ø
 8730 | 0010_0010_0001_1010 | 221a | 0x221a | √
 8734 | 0010_0010_0001_1110 | 221e | 0x221e | ∞
```

Für Gleitkommazahlen gibt es ähnlich umfangreiche Darstellungsmöglichkeiten. Mit „e“ oder „E“ geben wir an, dass wir eine Darstellung in Exponentialschreibweise wünschen. Ein „f“ sorgt für eine feste Nachkommastellenzahl; ohne weitere Angabe erhalten wir sechs Nachkommastellen. Der Kennbuchstabe „g“ wählt je nach Zahlenwert und gewünschter Stellenzahl die Dezimal- oder die Exponentialschreibweise. Mit „%“ sehen wir Prozentwerte, bei denen der Zahlenwert mit 100 multipliziert wird und „n“ sorgt wieder für die schon bekannten nationalen Dezimal- und Tausendertrennzeichen.

Feste Nachkommastellen können wir nur bei den Darstellungsarten „f“, „e“ und „%“ wählen. Die Zahl nach dem Punkt im Platzhalter steht bei den Darstellungsarten „g“ und „n“ für die Gesamtstellenzahl.

```
import locale
locale.setlocale(locale.LC_ALL, '')

for x in [10.0**(i-3) for i in range(10)]:
    print(f"{x:10.2f} | {x:8.2e} | {x:6.5g} | {x:6.5n} | "
          f"{x:13.2%}")
```

```
      0.00 | 1.00e-03 |  0.001 |  0,001 |         0.10%
      0.01 | 1.00e-02 |   0.01 |   0,01 |         1.00%
      0.10 | 1.00e-01 |    0.1 |    0,1 |        10.00%
      1.00 | 1.00e+00 |      1 |      1 |       100.00%
     10.00 | 1.00e+01 |     10 |     10 |      1000.00%
    100.00 | 1.00e+02 |    100 |    100 |     10000.00%
   1000.00 | 1.00e+03 |   1000 |  1.000 |    100000.00%
  10000.00 | 1.00e+04 |  10000 | 10.000 |   1000000.00%
 100000.00 | 1.00e+05 |  1e+05 |  1e+05 |  10000000.00%
1000000.00 | 1.00e+06 |  1e+06 |  1e+06 | 100000000.00%
```

Platzhalter dürfen sogar verschachtelt werden. In unserem letzten Beispiel verändern wir in einer Schleife die Nachkommastellenzahl i bei der Ausgabe des Zahlenwertes 1/7:

```
for i in range(1,7):
    print(f"{1/7:.{i}f}")

0.1
0.14
0.143
0.1429
0.14286
0.142857
```

5.21.9 Die alte printf-kompatible Formatierung

In älteren Python-Quelltexten finden wir häufig eine andere Art der formatierten Ausgabe. Diese orientiert sich an der bereits 1972 mit der Sprache C vorgestellten printf-Funktion. Ihre Platzhalter bestehen im einfachsten Fall aus einem Kennbuchstaben mit vorangestelltem Prozentzeichen. Der Kennbuchstabe zeigt an, welcher Inhaltstyp eingesetzt werden kann. Übliche Werte sind „i“ für ganze Zahlen (int), „f“ für Gleitkommazahlen (float) und „s“ für Zeichenketten (str).

Zwischen Prozentzeichen und Kennbuchstabe kann noch die Breite des zu reservierenden Textbereichs angegeben werden, die Ausrichtung (links oder rechts; mit oder ohne führende Nullen) sowie bei Gleitkommazahlen die Anzahl der Nachkommastellen.

Die tatsächlich anstelle der Platzhalter einzusetzenden Werte führen wir in einem Tupel[1] auf, das von der Zeichenkette, welche die Formatangaben enthält, wiederum mit einem Prozentzeichen abgesetzt wird.

```
>>> a = 2/3
>>> b = 5/7
>>> c = a+b

>>> print("%.3f plus %.3f ergibt %.3f." % (a, b, c))
0.667 plus 0.714 ergibt 1.381.
```

Vergleich mit C

Wer sich als alter C-Programmierer[2] an der Syntax dieser Konstruktion stört, kann auch eine eigene printf-Funktion definieren, die so ähnlich wie in C aufgerufen wird:

```
def printf(Maske,*Werte):
    print(Maske%(Werte))
```

Aufrufen lässt diese sich beispielsweise wie folgt:

```
>>> printf("%.3f plus %.3f ergibt %.3f.", a, b, c)
0.667 plus 0.714 ergibt 1.381.
```

1 Ein „Tupel“ ist in Python eine Gruppe von durch Kommas getrennten Werten, die von runden Klammern umschlossen ist – siehe Kapitel 5.15.2 auf Seite 157.

2 … oder alte C-Programmiererin …

Vergleich mit Java

Wer das zu einfach findet, kann sich auch eine zu Java ähnliche Ausgabefunktion zusammenschrauben:

```
>>> class System:
        class out:
            def format(Maske, *Werte):
                print(Maske % Werte)
```

Ergebnis:

```
>>> System.out.format("In %s ist das ähnlich.", "Java")
In Java ist das ähnlich.
```

Übersicht

Die folgende Tabelle zeigt einige Möglichkeiten der Formatierung auf. Die einzusetzenden Werte sind hier weggelassen worden, um die Darstellung übersichtlich zu halten.

Platz-halter	Anwendung	Beispiele	Ergebnis

`%i, %d`	Ganzzahlen (int)	`"|%i|"`
		`"|%4i|"`
		`"|%-4i|"`
		`"|%04i|"`
			`|42|`
			`|  42|`
			`|42  |`
			`|0042|`

`%f`	Gleitkommazahlen (float)	`"|%f|"`
		`"|%.2f|"`
		`"|%6.2f|"`
		`"|%06.2f|"`
			`|1.234568|`
			`|1.23|`
			`|  1.23|`
			`|001.23|`

`%s`	Zeichenkette (str)	`"|%s|"`
		`"|%6s|"`
		`"|%-6s|"`
			`|abc|`
			`|   abc|`
			`|abc   |`

Platz-halter	Anwendung	Beispiele	Ergebnis

`%x, %X`	Ganzzahlen als Hexadezimalzahlen	`"%x"`
		`"%X"`
		`"#%02X%02X%02X"`
			`c0af`
			`C0AF`
			`#0707FF`

`%a`	Reduzierung auf ASCII-Zeichen	`"%s"` `"%a"`	`Größe` `'Gr\\xf6\\xdfe'`
`%c`	Unicode-Zeichen aus Ganzzahl	`"%i"` `"%c"`	`65` `A`
`%e, %E`	Exponentialdarstellung	`"%e"` `"%.2E"`	`1.234568e+08` `1.23E+08`

5.21.10 Kodierung und Dekodierung

Was wir gemeinhin als Zeichenketten wahrnehmen, besteht im Computer nur aus einer Folge von Zahlenwerten. Jedem Buchstaben wird dabei ein bestimmter Zahlenwert zugeordnet. Leider gibt es weltweit dutzende verschiedene Arten, diese Zuordnung vorzunehmen. Python und die meisten modernen Betriebssysteme verwenden die zum Unicode kompatible UTF-8-Kodierung. Manchmal jedoch geraten wir an Daten, die in einer falschen Kodierung vorliegen. Diese müssen wir übersetzen, um sie sinnvoll verwenden zu können. Wir verwenden dazu die Zeichenkettenmethoden **`encode`** und **`decode`**. Die Methode **`decode`** entschlüsselt dabei eine Bytefolge zu einer (hoffentlich) lesbaren Zeichenkette und die Methode **`encode`** verschlüsselt eine Zeichenkette zu einer Bytefolge.

Python zeigt Bytefolgen als Zeichenketten mit vorangestelltem „b“ an. Bytewerte zwischen 32 und 126 werden darin als ASCII-Zeichen dargestellt, die Werte 9, 10 und 13 als Tabulator **`\t`**, Zeilenumbruch **`\n`** und Wagenrücklauf **`\r`** und alle anderen Bytewerte in einer mit **`\x`** beginnenden Hexadezimal-Ersatzdarstellung.

```
>>> b = bytes([65, 66, 67, 32, 195, 182, 195, 164, 195,
               188, 32, 195, 159])
```

```
>>> b
b'ABC \xc3\xb6\xc3\xa4\xc3\xbc \xc3\x9f'

>>> b.decode("utf-8")
'ABC öäü ß'

>>> b.decode("windows-1252")
'ABC Ã¶Ã¤Ã¼ ÃŸ'

>>> "ABC Ã¶Ã¤Ã¼ ÃŸ".encode("windows-1252").decode("utf-8")
'ABC öäü ß'
```

Der 1963 eingeführte und 1968 erweiterte Zeichensatz des ASCII (*american standard code for information interchange*) enthält als druckbare Zeichen lediglich die wenigen in Abb. 79 dargestellten Zeichen.

+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	●

Abb. 79: ASCII-Zeichen

5.21.11 Komprimierung und Verschlüsselung

In Pythons Modul „*codecs*“ finden wir weitere Kodierungsarten, die recht praktisch sind. Wenn wir beispielsweise ein großes Datenpaket speichern wollen, das viele sich wiederholende Abschnitte aufweist, so können wir dieses mit den Kompressionsverfahren „zip“ oder „bz2“ verlustfrei in seiner Größe reduzieren.

Wir bauen uns zum Ausprobieren mal eine 10 Megabyte große Bytefolge, die immer wieder aus den aufsteigenden Zahlenwerten von 0 bis 255 besteht:

```
>>> B = bytes([i%256 for i in range(10_000_000)])
>>> len(B)
10000000
```

Nun laden wir das Modul „codecs“ und komprimieren die Bytefolge:

```
>>> from codecs import encode, decode
>>> B_zip = encode(B, "zip")
>>> B_bz2 = encode(B, "bz2")
```

Die Komprimierung mit dem zweiten Verfahren dauert erkennbar länger. Was haben wir gewonnen?

```
>>> len(B_zip)
39119
>>> len(B_bz2)
12489
```

Die Komprimierung mit dem ZIP-Algorithmus reduziert die Größe um 96 % und das BZ2-Verfahren spart sogar 99 % ein. Nicht schlecht.

Ist wirklich nichts verloren gegangen? Wir packen die komprimierten Bytefolgen wieder aus und vergleichen ihre Inhalte:

```
>>> B_unzip = decode(B_zip, "zip")
>>> B_unbz2 = decode(B_bz2, "bz2")
>>> len(B_unzip)
10000000
>>> len(B_unbz2)
10000000
```

```
>>> B_unzip == B_unbz2 == B
True
```

… sehr schön!

Falls wir einmal etwas anderes als Bytefolgen komprimieren wollen, zum Beispiel eine Liste oder ein Dictionary, so müssen wir diese Objekte zuerst in Bytefolgen umformen. Das Modul „pickle“ besitzt dafür die beiden Funktionen **dumps** und **loads**.

```
>>> from codecs import encode, decode
>>> from pickle import dumps, loads

>>> Riesenliste = ["Test", 1, 2, 3] * 10_000_000

>>> Bytefolge = dumps(Riesenliste)

>>> len(Bytefolge)
80080015
```

Das sind rund 80 Megabyte. Geht das nicht ein bisschen kompakter?

```
>>> Komprimiert = encode(Bytefolge, "bz2")

>>> len(Komprimiert)
5491
```

Die Bytefolge wurde gerade auf 0,7% ihrer Länge eingedampft. Wir schauen wieder, ob auch diesmal nichts verloren gegangen ist:

```
>>> ausgepackte_Liste = loads(decode(Komprimiert, "bz2"))

>>> ausgepackte_Liste == Riesenliste
True
```

Die Kompression um 99,3% war also tatsächlich verlustfrei.

Simple Verschlüsselung

Manchmal ist es angebracht, Informationen vor allzu einfachem Zugriff zu schützen. Geocaching-Fans beispielsweise verschleiern Hinweise auf Rätsellösungen gern mit dem Rot-13-Verfahren. Dabei wird jeder Buchstabe von „A“ bis „Z“ sowie jeder Buchstabe von „a“ bis „z“ durch sein 13 Stellen im Alphabet versetzt liegendes Gegenstück ersetzt. Alle anderen Zeichen bleiben unverändert. Wenden wir diese „Verschlüsselung“ ein zweites Mal an, gelangen wir wieder zum Originaltext. Kodierung und Dekodierung bewirken also in diesem Fall dasselbe.

```
>>> encode("Das soll nicht jeder lesen können.", "rot13")
'Qnf fbyy avpug wrqre yrfra xöaara.'

>>> encode("Qnf fbyy avpug wrqre yrfra xöaara.", "rot13")
'Das soll nicht jeder lesen können.'
>>> decode("Qnf fbyy avpug wrqre yrfra xöaara.", "rot13")
'Das soll nicht jeder lesen können.'
```

5.21.12 Sonderformen von Zeichenketten

B-Strings

Standardmäßig sind Zeichenketten in Python 3 Sequenzen aus Unicode-Zeichen in der Kodierung UTF-8. Gelegentlich ist es jedoch sinnvoll, die Bytes einer Zeichenkette ohne textuelle Interpretation zu verarbeiten. In Python 3 kennzeichnen wir solche undekodierten Byteketten mit einem vorangestellten „b“. In ihnen sind alle 256 möglichen Bytewerte erlaubt, unabhängig davon, ob sie als Zeichen dargestellt werden können oder nicht.

```
>>> S = "Grünanlage\tAntonín Dvořák\nFläche: 2500 m²"
>>> print(S)
Grünanlage    Antonín Dvořák
Fläche: 2500 m²
```

```
>>> S.encode("utf-8")
b'Gr\xc3\xbcnanlage\tAnton\xc3\xadn Dvo\xc5\x99\xc3\xa1k\nFl\
xc3\xa4che: 2500 m\xc2\xb2'
```

Die Zeichenkombination „\x“ zeigt dabei an, dass die folgenden zwei Zeichen den Hexadezimalcode des Bytes bilden, das sich an dieser Stelle des B-Strings befindet. „\xc3\xbc“ steht also für die Hexadezimalzahlen c3 und bc (dezimal 195 und 188), dem UTF-8-Code für das Zeichen „ü“.

B-Strings verwenden wir auch, wenn wir reine Binärdaten verarbeiten, also Daten, die nicht in einem menschenlesbaren Textformat vorliegen, wie beispielsweise Bilder, Audiodateien, ZIP-Archive oder Videos.

U-Strings

In Python 2 war die im vorigen Kapitel beschriebene Zuordnung umgedreht. Zeichenketten waren standardmäßig Bytefolgen und Unicode-Zeichenketten musste ein „u“ vorangestellt werden. Falls Sie in den Tiefen des Internets Codeschnipsel finden, in denen solche U-Strings vorkommen, handelt es sich um alten Python-2-Code.

In Python 3 wird ein vorangestelltes „u“ ignoriert.

```
>>> "Bötchen" == u"Bötchen"
True
```

R-Strings

In „rohen“ R-Zeichenketten findet keine Zeichenersetzung durch den Rückwärtsschrägstrich „\“ statt. Der Rückwärtsschrägstrich und das auf ihn folgende Zeichen werden wie zwei gewöhnliche Zeichen behandelt.

```
>>> S = "Grünanlage\tAntonín Dvořák\nFläche: 2500m²"
>>> R = r"Grünanlage\tAntonín Dvořák\nFläche: 2500m²"
>>> print(S)
Grünanlage    Antonín Dvořák
Fläche: 2500m²
```

```
>>> print(R)
Grünanlage\tAntonín Dvořák\nFläche: 2500m²
```

Es gibt bei der Verwendung von R-Strings eine Einschränkung: Da der Rückwärtsschrägstrich und das auf ihn folgende Zeichen als gewöhnliche Zeichen übernommen werden, darf ein R-String niemals mit einem Rückwärtsschrägstrich enden, da dieser ihm sein schließendes Anführungszeichen „klaut“.

R-Strings sind besonders bei der Textmusterbeschreibung durch die mächtigen Regulären Ausdrücke hilfreich, in denen häufig Rückwärtsschrägstriche vorkommen.

Reguläre Ausdrücke werden aufgrund der ihnen innewohnenden Komplexität in diesem Kurs nicht behandelt.

F-Strings

F-Strings wurden mit Python 3.6 eingeführt und erlauben eine besonders einfache Ausgabe formatierter Zahlenwerte. Sie werden in Kapitel 5.21.6 ausführlich behandelt.

5.22 Dateien lesen und schreiben

5.22.1 Textdateien lesen

Textdateien sind Dateien, deren Inhalt vorzugsweise aus ASCII-Zeichen oder UTF-8-kodierten Unicodezeichen besteht.

Das Öffnen einer Datei mit der Funktion **`open()`** erzeugt ein iterierbares Objekt, das Dateihandle.

Der Funktion **`open()`** muss immer eine Zeichenkette übergeben werden, in der sich der Name der zu öffnenden Datei befindet. Python sucht dann in demselben Verzeichnis nach der Datei, in der sich auch das gerade ausgeführte Programm befindet. Wenn wir dem Dateinamen einen Verzeichnisnamen voranstellen, dann sucht unser Pythonprogramm die Datei in dem entsprechenden Unterverzeichnis seines Programmverzeichnisses. Dieser Verzeichnisname wird auch in Windows mit einem gewöhnlichen Schrägstrich vom Dateinamen abgesetzt. Siehe auch Kapitel 2.3.

Nach dem Lesen einer Datei sollte diese wieder geschlossen werden. Das geschieht entweder explizit durch Aufruf der Dateihandle-Methode **`.close()`** oder implizit, indem wir über das Schlüsselwort **`with`** einen Kontextblock einrichten, dessen Verlassen die Datei automatisch schließt.

```
# Dateizugriff mit explizitem Schließen
datei = open("Test.txt", "r")
print(datei.read())
datei.close()

# Dateizugriff mit implizitem Schließen
with open("Test.txt", "r") as datei:
    print(datei.read())
```

Python kennt vier grundlegende Arten, eine Textdatei zu lesen:

Python-Code	Resultat
`for zeile in datei:`	Liest Zeile für Zeile bis zum Dateiende nacheinander in die Variable **`zeile`** ein.
`zeile = datei.readline()`	Liest genau eine Zeile aus der Datei in eine Zeichenkette.
`zeilen = datei.readlines()`	Liest alle Zeilen einer Datei in eine Liste aus Zeichenketten.
`alles = datei.read()`	Liest die gesamte Datei in eine einzige Zeichenkette.

Beim zeilenweisen Lesen einer Datei werden die Zeilen einschließlich des Zeilenendezeichens **`\n`** gelesen. Unter Umständen befinden sich auch noch Leerzeichen und Tabulatoren am Zeilenende. Die Methode **`.rstrip()`** entfernt die meistens unerwünschten Leer- und Steuerzeichen vom Zeilenende.

```
with open("Liste.txt", "r") as meine_Datei:
    for zeile in meine_Datei:
        print(zeile.rstrip())
```

Gelegentlich haben wir Dateien zu verarbeiten, die in den ersten Zeilen Kommentare oder andere für uns unwichtige Texte enthalten und deren eigentliche Daten erst danach beginnen. Diese Zeilen können wir einfach mit **`next()`** überspringen.

```
with open("Liste.txt") as meine_Datei:
    # vier unwichtige Zeilen überspringen
    for n in range(4):
        next(meine_Datei)
    # den ganzen Rest der Datei einlesen
    for zeile in meine_Datei:
        print(zeile.rstrip())
```

Textdateien, die nicht in der Standardzeichenkodierung des Betriebssystems vorliegen, müssen beim Lesen umkodiert werden, damit Umlaute und Sonderzeichen korrekt dargestellt werden.

Üblicherweise verwenden heutige Betriebssysteme die Zeichenkodierung UTF-8. Unter Microsoft Windows treten jedoch trotz der Empfehlung Microsofts[1], UTF-8 zu verwenden, immer wieder die veralteten Einzelbyte-Kodierungen auf. Hier muss die Zeichenkodierung grundsätzlich immer angegeben werden, um kompatibel zum Rest der Welt zu sein.

```
with open("Liste.txt", "r", encoding="utf-8") as meine_Datei:
    for zeile in meine_Datei:
        print(zeile.rstrip())
```

Falls das Lesen der Textdatei trotz dieser Angabe mit einer Fehlermeldung vom Typ **`UnicodeDecodeError`** abbricht, so liegt die Datei nicht im UTF-8-Format vor. Probieren Sie in dem Fall der Reihe nach einige der vielen weltweit immer noch verwendeten Zeichenkodierungen des vorigen Jahrhunderts aus: "windows-1252", "windows-1251", "windows-1254", "iso-8859-15", "iso-8859-1", "cp850" oder "cp437". Achten Sie besonders auf Umlaute und andere nicht-ASCII-Sonderzeichen.

Irreführenderweise heißt die auf einem beliebigen Windows-PC gerade eingestellte Zeichenkodierung dort immer „ANSI“. Bitte vermeiden Sie diese sinnleere Kodierungsbezeichnung, da sie keiner eindeutigen Kodierung entspricht und Python-Programme, die diese Kodierungsangabe verwenden, nicht auf macOS oder Linux lauffähig sind.

Wenn alle Versuche scheitern, die richtige Kodierung zu finden, ist die Datei möglicherweise gar keine Textdatei, sondern eine Binärdatei (Kapitel 5.22.4).

1 Microsoft: Code Page Identifiers, https://docs.microsoft.com/en-us/windows/win32/intl/code-page-identifiers.

5.22.2 Textdateien schreiben

Um eine zum Schreiben geöffnete Textdatei zeilenweise zu beschreiben, können wir der Print-Funktion das Handle der geöffneten Datei mit dem Parameter **file** übergeben.

```
with open("Liste.txt", "w", encoding="utf-8") as meine_Datei:
    print("Hallo Datei!", file=meine_Datei)
```

Jede zum Schreiben geöffnete Datei verfügt aber auch über eine Methode **write**. Diese schreibt genau eine Zeichenkette ohne darauf folgendes Zeilenwechselzeichen in eine Datei.

```
with open("Liste.txt", "w", encoding="utf-8") as meine_Datei:
    meine_Datei.write("Hallo Datei!")
```

Vorsicht! Das Öffnen einer Datei zum Schreiben vernichtet auf der Stelle sämtliche zuvor in der Datei vorhandenen Inhalte.

5.22.3 Textdateien fortsetzen

Wenn wir Zeichenketten an bereits vorhandene Textdateien anhängen wollen, so öffnen wir diese nicht im Modus „**w**“ (wie „write“), sondern im Modus „**a**“ (wie „append“).

```
with open("Liste.txt", "a", encoding="utf-8") as meine_Datei:
    meine_Datei.write("Dies wird an die Datei angehängt.")
```

5.22.4 Binärdateien

Als Binärdateien bezeichnen wir alle Dateien, deren Inhalt etwas anderes als lesbarer Text ist. Das können beispielsweise Bilddateien, Videos, ZIP-Archive, „Worddateien“ (also Textverarbeitungsdateien) oder Tonaufzeichnungen sein. Auch komplette Python-Objekte wie Listen oder Dictionarys können wir in Binärdateien speichern.

Binärdateien können wie Textdateien mit dem Methoden **read** und **write** gelesen und beschrieben werden. Beim Öffnen der Dateien müssen wir lediglich den Modus („**r**“, „**w**“ oder „**a**“) noch um den Buchstaben „**b**“ ergänzen, um Python mitzuteilen, dass die Inhalte keiner Zeichenkodierung entsprechen und aus der Datei daher „nackte“ Bytefolgen gelesen und in sie geschrieben werden.

5.22.5 Pickle

Um Python-Objekte als Ganzes in Dateien zu schreiben oder aus ihnen zu lesen, verwenden wir das Modul **pickle**[1]. Dieses enthält die beiden Funktionen **dump** und **load**, mit denen komplette Python-Objekte in Binärdateien geschrieben und aus ihnen gelesen werden können.

```
>>> import pickle

>>> meineListe = [1, 2, 3, "Test"]

>>> with open("Datei.dat","wb") as datei:
        pickle.dump(meineListe, datei)

>>> with open("Datei.dat","rb") as datei:
        neueListe = pickle.load(datei)

>>> neueListe
[1, 2, 3, 'Test']
```

Das Lesen und Schreiben durch Pickle ist extrem schnell und wird in der Regel nur durch die Festplattenhardware beschränkt.

1 Wenn Python eine deutschsprachige Entwicklung wäre, hieße diese Funktion jetzt möglicherweise „einmachen“. Das gedankliche Bild, ein Objekt durch „Einmachen“ haltbar aufbewahren zu können, ist jedenfalls recht anschaulich. Es gibt sogar ein weiteres Python-Modul, das dieses Bild noch weiter strapaziert und den Einmachgläsern ein Regal spendiert. Es heißt … shelve.

Standardmäßig kann die Funktion **dump** Objekte bis zu einer Größe von rund 4,3 Gigabyte in eine Datei schreiben. Wenn größere Objekte geschrieben werden sollen, ist der Parameter **protocol** auf den Wert **4** (oder einen negativen Wert) zu setzen.

Protokoll-version	Besonderheit
0	Das Format der ersten Python-Versionen. Die von **dump** geschriebene Datei ist hier noch eine menschenlesbare Textdatei.
1, 2	Binärformate der alten Python-2-Versionen.
3	Standard-Binärformat seit Python 3. Unterstützt Objekte bis 4 GiB (2^{32} Byte) Größe.
4	Verbessertes Format seit Python 3.4. Erlaubt schnellere Zugriffe und sehr große Objekte (2^{64} Byte).

Das folgende Programm erzeugt ein 5 Gigabyte großes Objekt, schreibt es in eine Datei und liest diese Datei wieder ein. Dazwischen merkt es sich die jeweils aktuelle Systemzeit in den Variablen **t0**, **t1** und **t2**. Die Differenzen zwischen den gemessenen Zeiten zeigen, wie lang die Ausführung der einzelnen Befehle dauerte.

```
from pickle import load, dump
from time import time

dickesding = b"x" * 5_000_000_000
print("Geschwindigkeitstest mit", len(dickesding), "Bytes")

t0 = time()
with open("Speedtest.dat", "wb") as datei:
    dump(dickesding, datei, protocol=4)
t1 = time()
with open("Speedtest.dat", "rb") as datei:
    dickesding = load(datei)
t2 = time()
```

```
print(f"{t1-t0:7.3f} Sekunden, um zu schreiben")
print(f"{t2-t1:7.3f} Sekunden, um wieder zu lesen")
```

Das Ergebnis lässt erstaunliche Festplattengeschwindigkeiten vermuten.

```
Geschwindigkeitstest mit 5000000000 Bytes
 13.794 Sekunden, um zu schreiben
  4.340 Sekunden, um wieder zu lesen
```

Der Trick des Betriebssystems (hier: Linux) besteht darin, das Schreiben schon für vollzogen zu erklären, wenn es tatsächlich erst alle zu sichernden Daten von unserem Programm entgegengenommen hat und eigentlich noch fleißig dabei ist, diese im Hintergrund zur Festplatte zu schicken. Auch beim Lesen wird es nicht den tatsächlichen Inhalt der Festplatte ausgeben, solange es noch „gecachte“ Daten der letzten Schreib- oder Lesezugriffe im RAM weiß. Bei Rechnern mit wenig RAM dauern Dateizugriffe daher oft viel länger als auf Rechnern mit ausreichend großem Speicher, der sich vom Betriebssystem als Festplattencache nutzen lässt.

5.22.6 Das aktuelle Arbeitsverzeichnis

In den vorangegangenen Beispielen haben wir beim Öffnen der Dateien lediglich einen Dateinamen ohne Angabe eines Laufwerksbuchstabens oder Verzeichnisnamens verwendet. Python sucht die zu lesenden Dateien dann in demselben Verzeichnis, in dem sich auch das gerade ausgeführte Programm befindet und legt neue Dateien in ebendiesem Verzeichnis an.

Nachdem in IDLE ein Python-Programm ausgeführt wurde, entspricht das Arbeitsverzeichnis der IDLE-Shell ebenfalls dem Verzeichnis, in dem dieses Programm zuvor gespeichert wurde.

Wenn Sie Dateibefehle nicht aus einem Programm heraus aufrufen, sondern direkt in die IDLE-Shell eintippen, so ist das aktuelle Arbeitsverzeichnis je nach Betriebssystem oder Installation unterschiedlich. Unter Linux wird in der Regel das Home-Verzeichnis des aktuell angemeldeten Useraccounts als Arbeitsverzeichnis verwendet, unter Windows dagegen

oft ein Unterverzeichnis von „C:\PROGRAM FILES\“ oder „C:\WINDOWS\SYSTEM32\“, für das Sie vermutlich keine Schreibrechte besitzen und darauf mit der Fehlermeldung „PermissionError“ hingewiesen werden (Abb. 80).

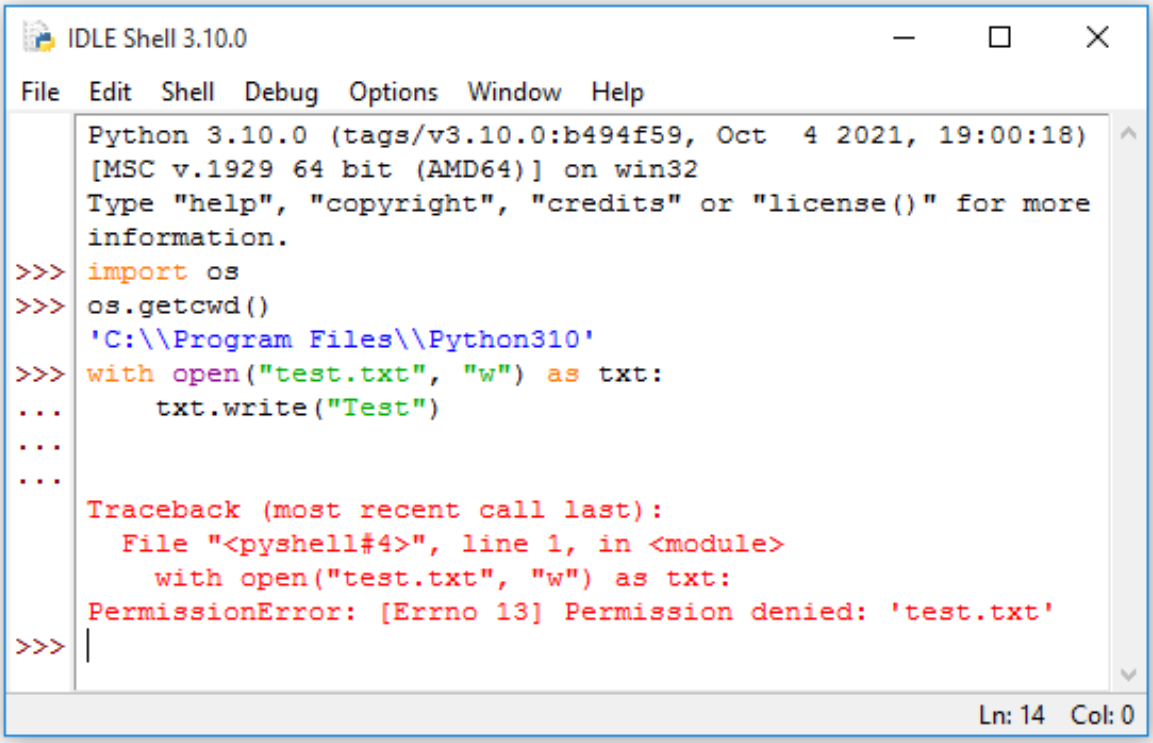


Abb. 80: Schreibrechte unter Windows

Um das aktuelle Arbeitsverzeichnis der IDLE-Shell herauszufinden, können Sie dort folgende zwei Zeilen eingeben:

```
import os
os.getcwd()
```

Mithilfe der Funktion **os.chdir()** lässt sich das aktuelle Arbeitsverzeichnis auch wechseln. Unabhängig vom Betriebssystem verwendet Python als Trennzeichen zwischen Verzeichnisnamen den gewöhnlichen Schrägstrich „/“. Falls Sie den windowstypischen Rückwärtsschrägstrich bevorzugen, müssen Sie ihn jeweils doppelt schreiben: „\\“.

Mit dem Befehl

```
os.chdir("/home/meinname/Desktop")
```

wechseln Sie beispielsweise unter Linux auf den Desktop der Anwenderin oder des Anwenders „meinname“ und

```
os.chdir("C:\\USERS\\meinname\\DESKTOP")
```

macht dasselbe unter Microsoft Windows.

Um unabhängig vom Betriebssystem den Desktop zum aktuellen Arbeitsverzeichnis zu erklären, können Sie folgende Konstruktion verwenden:

```
os.chdir(os.path.expanduser("~/Desktop"))
```

5.23 Diagramme mit Matplotlib

Mit der Bibliothek **matplotlib** können wir auf einfache Weise ansprechende Diagramme erzeugen.

Matplotlib[1] gehört nicht zum Standardumfang von Python, kann jedoch schnell mit dem Konsolenbefehl „**pip3 install matplotlib**“ nachinstalliert werden. Tipps zur Installation finden Sie in Kapitel 5.1.1.

5.23.1 Ein schnelles x-y-Diagramm

Im einfachsten Fall brauchen wir nur ein paar x- und y-Werte in einer Liste, um ein Diagramm zu erzeugen:

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4, 5]
y = [0, 1, 4, 9, 16, 25]

plt.plot(x, y)
plt.show()
```

Ohne weiteren Aufwand entsteht so schon ein ansehnliches Diagramm (Abb. 81).

Dass das Modul **matplotlib.pyplot** beim Import traditionell in **plt** umbenannt wird, hat den einfachen Grund, dass **plt** schneller zu tippen und zu lesen ist.

1 Der Name steht für „matrix plot library“ und wird deshalb nicht mit „th“ geschrieben.

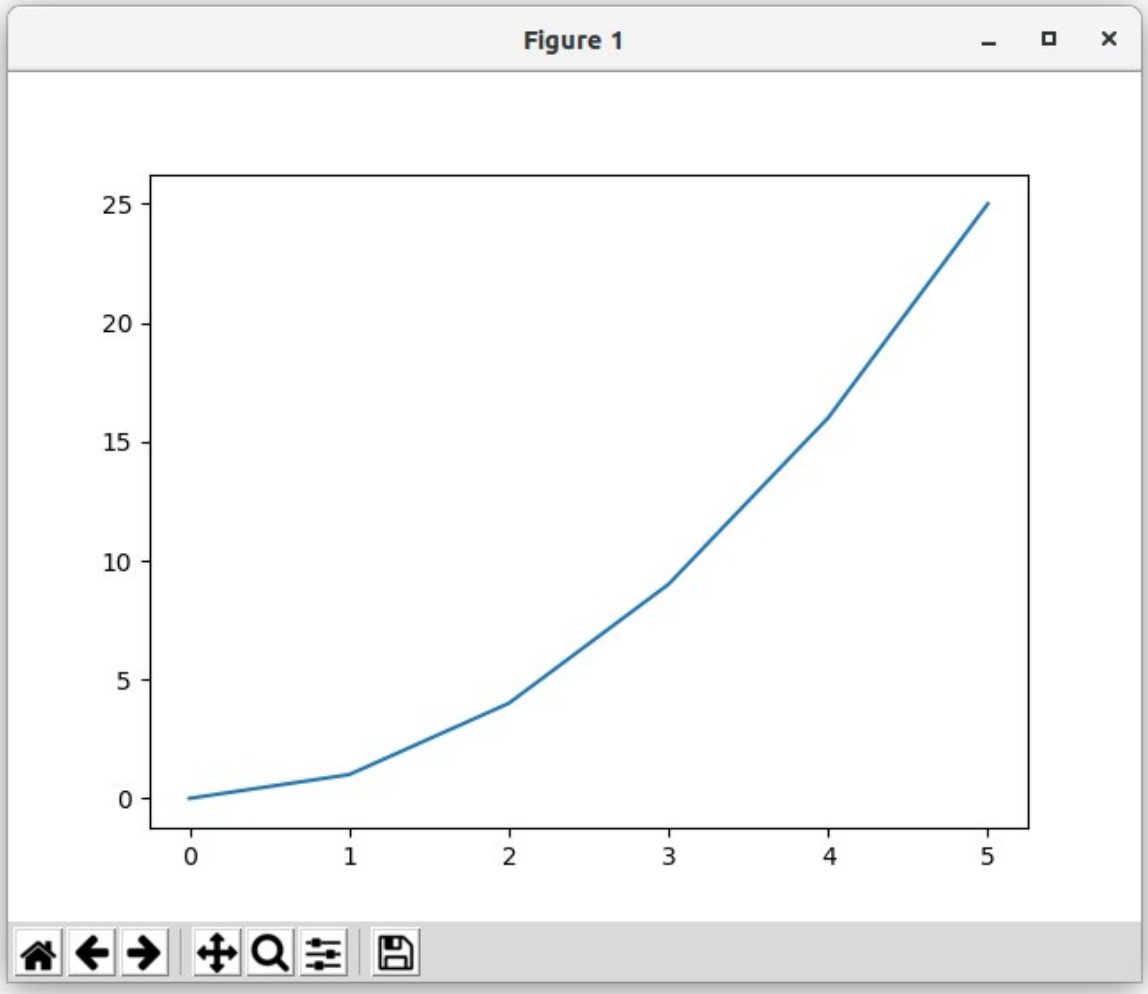
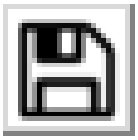
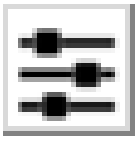


Abb. 81: Eines der einfachsten Matplotlib-Diagramme

Am unteren Rand des Matplotlib-Fenster sehen wir sieben Icons.



Mit dem Diskettensymbol ganz rechts in der Icon-Leiste kann die aktuelle Ansicht in verschiedenen Grafikformaten, zum Beispiel PDF, PNG oder SVG gespeichert werden, um beispielsweise in Textverarbeitungsdateien eingebunden zu werden.



Das Schiebereglersymbol erlaubt es, die Ränder und, bei mehreren Diagrammen pro Fenster, die Abstände zwischen den Diagrammen nachträglich zu verändern. Die Angaben sind relativ zur aktuellen Fensterhöhe und -breite.

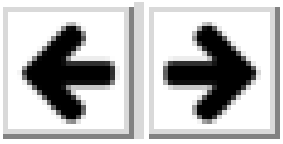
Nach Anklicken des Lupensymbols können wir mit der linken Maustaste ein Rechteck in der Diagrammfläche aufziehen, welches anschließend den enthaltenen Ausschnitt auf die gesamte Diagrammfläche vergrößert (Zoom-Funktion).

Ziehen wir das Rechteck dagegen mit der rechten Maustaste auf, so wird die gesamte sichtbare Diagrammfläche auf das Rechteck verkleinert.

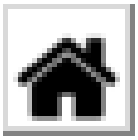
Durch Festhalten der Tasten x und y wird das Rechteck auf die gesamte Höhe oder Breite der Diagrammfläche erweitert, sodass sich die Größenänderung nur auf die ausgewählte Achse auswirkt.

Nach Anklicken des Pfeilkreuz-Icons können wir das Diagramm mit der linken Maustaste innerhalb der Diagrammfläche verschieben (Pan-Funktion).

Mit der rechten Maustaste können wir die Größe des Diagramms frei in x- und y-Richtung verändern. Der beim Klicken unter dem Mauszeiger befindliche Punkt des Diagramms wird zum Fixpunkt, um den herum die Größenänderung stattfindet. Durch Festhalten der Taste Strg fixieren wir das Seitenverhältnis, um die Grafik nicht zu verzerren.



Jede Zoom- oder Verschiebeaktion erzeugt eine neue Ansicht. Mit dem Pfeil-Icons können wir zwischen den verschiedenen Ansichten vor- und zurückblättern.



Ein Klick auf das Häuschen setzt das Diagramm wieder auf seine Anfangsdarstellung zurück.

Zusätzliche Funktionen können wir über die Tastatur aufrufen.

Taste	Funktion
F	Schaltet die Vollbilddarstellung ein/aus.
G	Schaltet die Gitterdarstellung zwischen vertikalen, horizontalen, allen und keinen Gitterlinien um.
K	Wechselt zwischen logarithmischen und linearen x-Werten.

Taste	Funktion
L	Wechselt zwischen logarithmischen und linearen y-Werten.
O	Ruft die Zoom-Funktion auf.
P	Ruft die Pan-Funktion auf.
Q	Schließt das Diagramm.
S	Speichert das Diagramm als Datei.
← →	Wechselt zwischen den bisher erfolgten Zoom- und Pan-Darstellungen

5.23.2 Ein schönes x-y-Diagramm

Die Möglichkeit, das Aussehen von Diagrammen wiederholbar zu beeinflussen, hilft uns, wenn wir Texte mit zahlreichen Diagrammen, wie zum Beispiel Forschungsberichte, Gutachten oder Bachelorarbeiten verfassen. In einer Tabellenkalkulationen kann man zwar auch nach einigem Herumprobieren das eine oder andere hübsche Diagramm erzeugen, diese Arbeit dann jedoch dutzende oder hunderte Male exakt wiederholen zu müssen, ist menschenunwürdig.

In einem Pythonprogramm definieren wir dagegen einmalig das Aussehen eines Matplotlib-Diagramms und können anschließend in einer Schleife hunderte Diagramme derselben Größe und Gestalt als Grafikdatei auf die Festplatte schreiben.

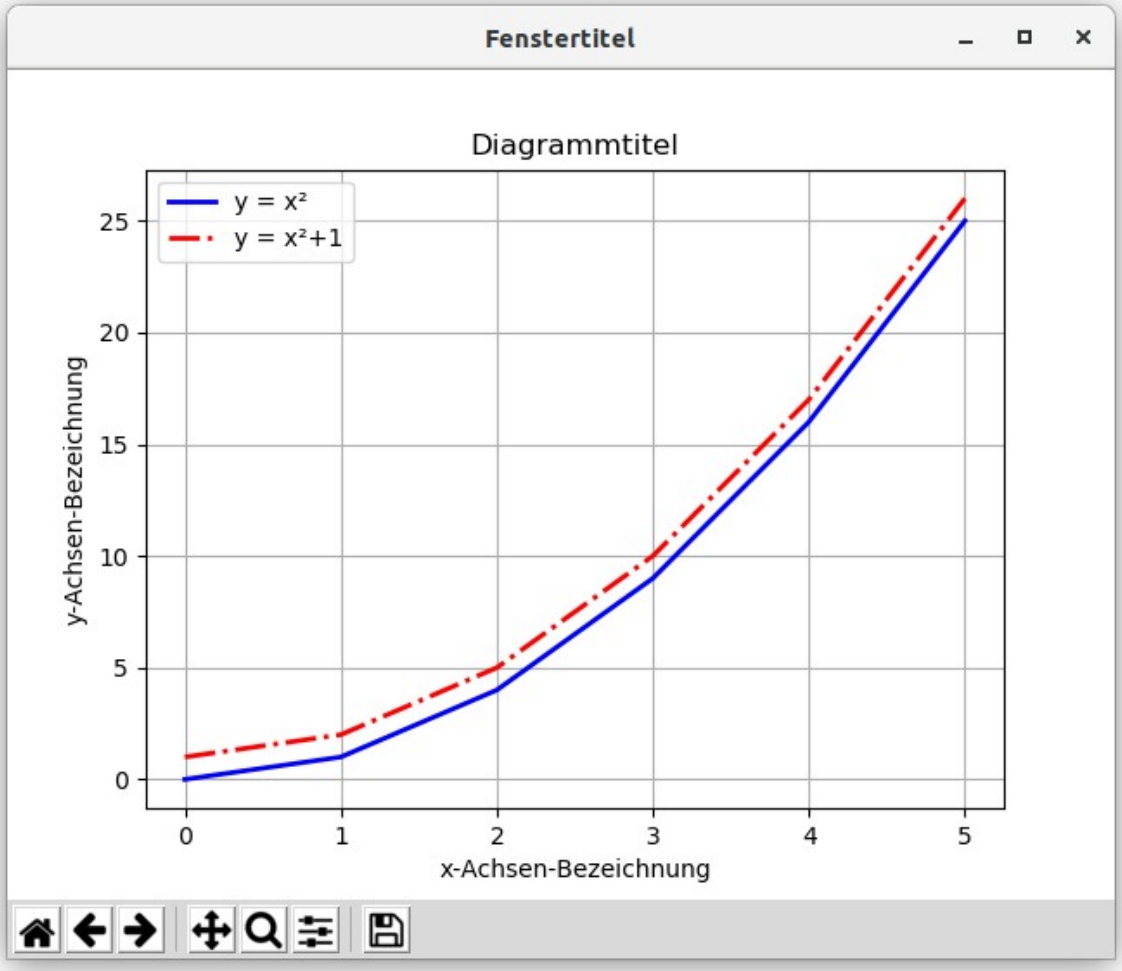


Abb. 82: Verbessertes Matplotlib-Diagramm

Wir werfen einen Blick in das erzeugende Programm zu Abb. 82. Zunächst werden die Listen mit den darzustellenden Daten gefüllt. Im Beispiel sind das feste Werte. Tatsächlich würde unser Programm die Werte zuvor berechnen.

Da wir zwei Graphen darstellen wollen, benötigen wir neben der Liste mit den x-Werten zwei Listen mit y-Werten:

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4, 5]
y1 = [0, 1, 4, 9, 16, 25]
y2 = [1, 2, 5, 10, 17, 26]
```

Die Werte aus der Liste **y1** sollen mit einer blauen Linie, die 2 Einheiten breit und durchgezogen ist, gezeichnet werden. Der Linie wird der Legendentext „**y = x²**“ zugeordnet:

```
plt.plot(x, y1,
         color="blue", linewidth=2, linestyle="-",
         label="y = x²")
```

Die Einheit für die Linienbreiten in Matplotlib ist der typographische Punkt, dieser entspricht 1/72 Zoll oder rund 0,35 mm. Auf alten Röhrenmonitoren entspricht das ziemlich genau einem Bildschirmpixel.

Für die Werte aus der Liste **y2** wählen wir eine rote Linie, die 2 Einheiten breit und strichpunktiert ist. Dieser Linie wird der Legendentext „**y = x²+1**“ zugeordnet:

```
plt.plot(x, y2,
         color="red", linewidth=2, linestyle="dashdot",
         label="y = x²+1")
```

Eine Liste von englischsprachigen Farbnamen finden Sie im Anhang dieses Skripts in Kapitel 7.2. Anstelle der Farbnamen können Sie für den Parameter **color** auch den RGB-Code der jeweiligen Farbwerte verwenden, also beispielsweise **"#0000FF"** anstelle von **"blue"** oder **"#F5F5DC"** anstelle von **"beige"**.

Die verfügbaren Linienstile für den Parameter **linestyle** sind **"solid"** oder **"-"** für durchgezogene Linien, **"dashed"** oder **"--"** für gestrichelte Linien, **"dotted"** oder **":"** für gepunktete Linien und **"dashdot"** oder **"-."** für strichpunktierte Linien.

In der Diagrammfläche soll ein hellgraues Raster angezeigt werden:

```
plt.grid(color="lightgrey")
```

Die Legende mit den beiden zuvor festgelegten Legendentexten (label) soll an einer passenden Stelle angezeigt werden:

```
plt.legend()
```

Oberhalb des Diagramms soll eine Überschrift angezeigt werden. Auch die beiden Achsen erhalten passende Bezeichnungen:

```
plt.title("Diagrammtitel")
plt.ylabel("y-Achsen-Bezeichnung")
plt.xlabel("x-Achsen-Bezeichnung")
```

Wer unbedingt möchte, kann auch dem Bildschirmfenster des Diagramms einen neuen Fenstertitel geben:

```
plt.gcf().canvas.set_window_title("Fenstertitel")
```

Nachdem wir alle gewünschten Einstellungen am Diagramm vorgenommen haben, schreiben wir es in eine Grafikdatei im besonders vorteilhaften SVG-Format:

```
plt.savefig("Diagramm.svg")
```

Zum Schluss stellen wir das fertige Diagramm auf dem Bildschirm dar:

```
plt.show()
```

Wenn wir in einem Programm nacheinander mehrere Diagrammdateien durch **`savefig()`** anlegen, so kann es störend sein, danach jedes Mal das mit **`plt.show()`** erzeugte Bildschirmfenster zu schließen, damit das Programm weiterläuft. Manche Programme erzeugen hunderte von Grafiken. Wenn wir den Aufruf aber einfach fortlassen, wird der Zeichenbereich nicht zurückgesetzt und alle folgenden Diagramme werden ebenfalls in die bestehende Zeichenfläche hineingezeichnet. Wenn wir **`plt.show()`** jedoch einfach durch **`plt.close()`** ersetzen, so setzt dieses nach dem Exportieren die Grafik zurück ohne dazu das Programm anzuhalten.

5.23.3 Streudiagramme

Streudiagramme (Scatterplots) stellen die x-y-Koordinaten zweier übergebener Listen als einzelne Punkte dar.

Einfache Streudiagramme können wir mit der uns schon bekannten Funktion **`plt.plot()`** umsetzen, indem wir dieser die beiden zusätzlichen Parameter **`marker`** und **`markersize`** übergeben.

Dem Parameter **`marker`** weisen wir eines der folgenden Zeichen zur Markierung der Datenpunkte zu: **`"o"`** für Kreise, **`"v"`**, **`"^"`**, **`"<"`** und **`">"`** für Dreiecke mit einer Ecke nach unten, oben, links und rechts, **`"8"`** für Achtecke, **`"s"`** für Quadrate, **`"p"`** für Fünfecke, **`"*"`** für Sterne, **`"h"`** und **`"H"`** für auf einer Spitze oder einer Seite stehende Sechsecke, **`"D"`** und **`"d"`** für breite oder schmale Rauten, **`"P"`** für Pluszeichen und **`"X"`** für Kreuzchen.

Die Einheit für die Größenangabe **`markersize`** ist wie oben bei **`linewidth`** der typographische Punkt. 28 typographische Punkte entsprechen rund einem Zentimeter.

Wenn wir keine Linien zwischen den Markern zeichnen möchten, setzen wir den Parameter **`linestyle`** auf eine leere Zeichenkette.

```
import matplotlib.pyplot as plt

x = [0, 0, 1, 2, 2]
y = [0, 2, 4, 2, 0]

plt.plot(x, y, marker="o", markersize=28, linestyle="")
plt.grid()
plt.show()
```

Der damit erzeugte Scatterplot sollte so aussehen wie in Abb. 83 gezeigt.

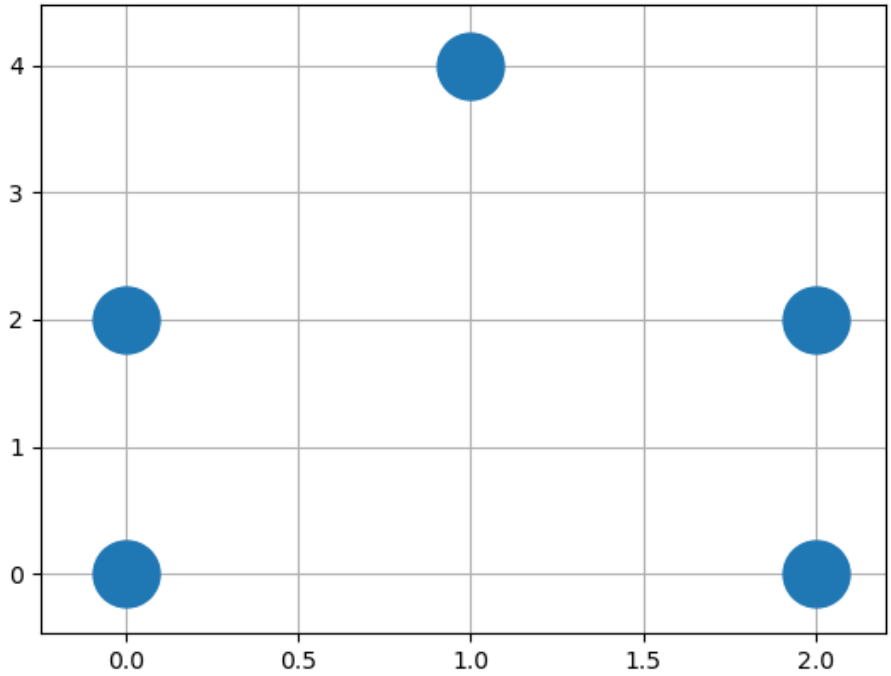


Abb. 83: Plot mit Markern

Noch leistungsfähiger ist die speziell für Streudiagramme gedachte Funktion **plt.scatter()**, denn bei dieser müssen wir die Punktgröße und Punktfarbe nicht für alle Punkte einheitlich vorgeben, sondern können sie für jeden einzelnen Punkt mithilfe einer weiteren Liste festlegen.

```
import matplotlib.pyplot as plt

x = [0, 0, 1, 2, 2]
y = [0, 2, 4, 2, 0]
fläche = [4000, 3000, 2000, 5000, 6500]
farbe = ["red", "green", "blue", "brown", "magenta"]

plt.scatter(x, y, s=fläche, c=farbe)
plt.grid()
plt.show()
```

Das Maß für die Punktgröße in Abb. 84 ist hier nicht der Durchmesser der Marker, sondern die von ihnen ausgefüllte Fläche in Vielfachen von 0,1244 mm² bzw. 1/5184 Quadratzoll, weshalb die Zahlenwerte in der Liste **fläche** ungewöhnlich groß erscheinen.

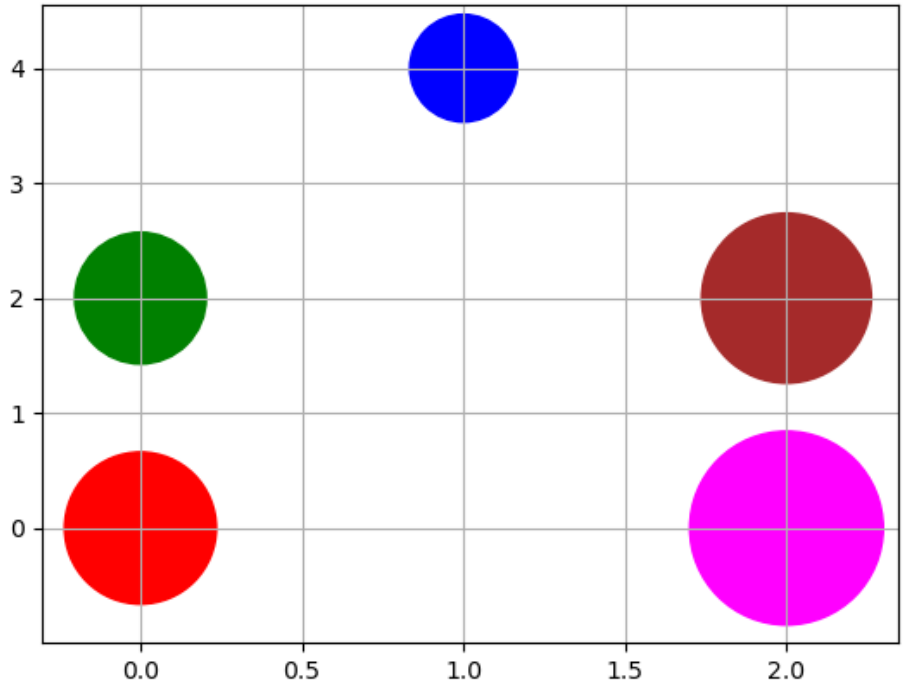


Abb. 84: Scatterplot mit Flächen- und Farblisten

5.23.4 Text

Mit **`plt.text(x, y, s)`** positionieren wir einen einzelnen Text **`s`** auf der Diagrammfläche am Einfügepunkt (x,y). Standardmäßig bezeichnen **`x`** und **`y`** die Koordinaten der unteren linken Ecke des darzustellenden Textes.

Mithilfe zahlreicher Parameter können wir die Formatierung des Textes beeinflussen. Nach dem Matplotlib-Import erhalten wir die komplette Liste durch den Funktionsaufruf **`help(plt.text)`**[1]. Hier eine kleine Auswahl:

`rotate=w` dreht den Text im Gegenuhrzeigersinn im Winkel w [Grad] um den Einfügepunkt.

`color=f` oder **`c=f`** weist dem Text die Farbe f zu.

`size=g` weist dem Text eine relative Größenangabe zu. Erlaubt sind für g die Werte **`"small"`**, **`"medium"`** und **`"large"`**, wobei **`small`** und **`large`** noch die Verstärker **`x-`** und **`xx-`** vorangestellt werden können.

1 Ausführlich erklärt werden die Einstellungen auf https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.text.html

ha=crl und **va=tbcb** legen die horizontale und vertikale Ausrichtung relativ zum Einfügepunkt fest. Für **ha** sind die Werte **"left"**, **"right"** und **"center"** erlaubt und **va** können wir die Werte **"top"**, **"bottom"**, **"center"** und **"baseline"** zuweisen. Der Wert **"bottom"** bezieht sich dabei auf die Unterkante des Textes einschließlich Unterlängen und **"baseline"** auf die Grundlinie des Textes, also in der Regel die Unterkante der Großbuchstaben.

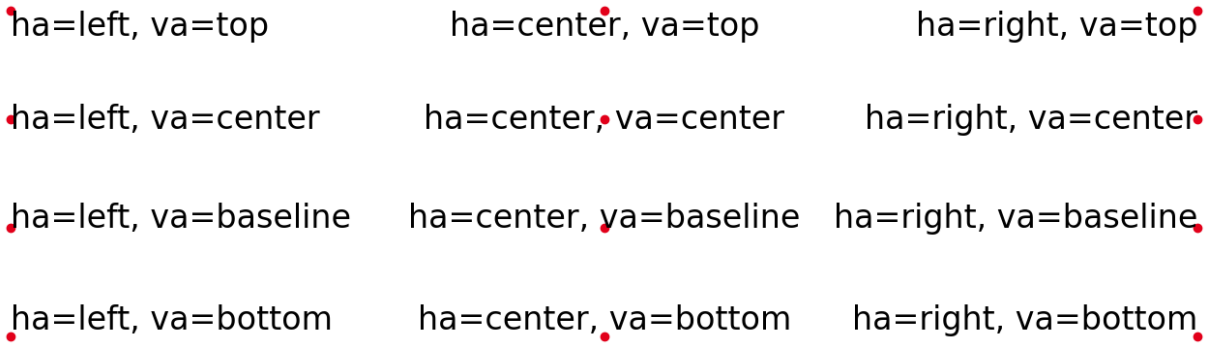


Abb. 85: Textausrichtung mit Matplotlib

5.23.5 gefüllte Flächen

Ausgefüllte Flächen erzeugen wir durch **plt.fill()**, indem wir dieser Funktion Listen umlaufender Randpunktkoordinaten übergeben, welche die auszufüllende Fläche aufspannen (Abb. 86).

```
import matplotlib.pyplot as plt

x = [0, 0, 1, 2, 2]
y = [0, 2, 4, 2, 0]

plt.fill(x, y, color="firebrick")
plt.grid()
plt.show()
```

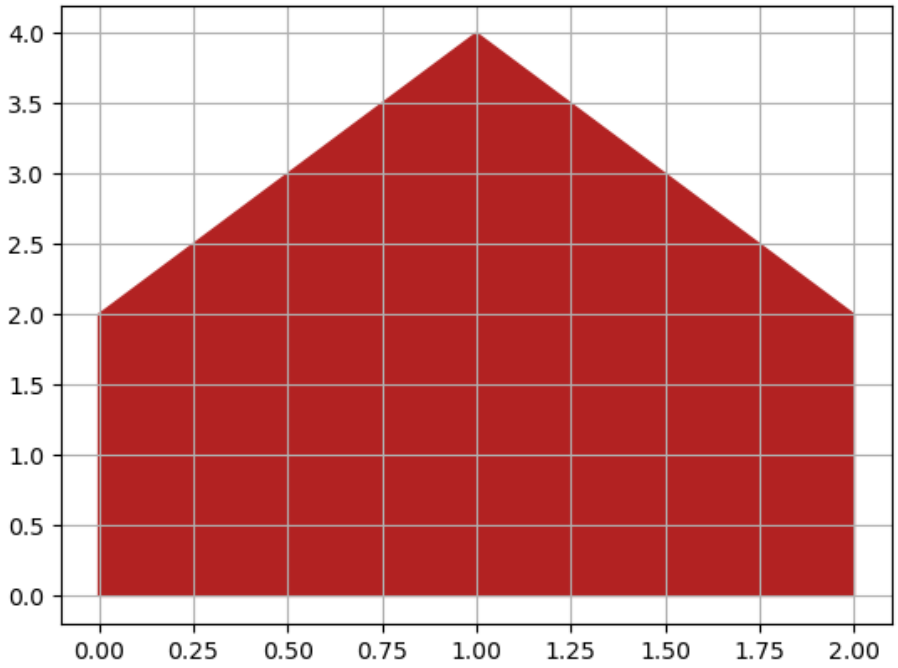


Abb. 86: Flächenfüllung mit plt.fill()

5.23.6 Zeichenreihenfolge

Alle Grafikelemente in Matplotlib werden nach einem festen Schema übereinander angeordnet. Wenn wir die dadurch festgelegte Zeichenreihenfolge ändern wollen, müssen wir den Wert des Parameters **zorder** der erzeugten Grafikobjekte ändern. Je höher dieser Wert ist, desto weiter oben (oder vorne) werden die Elemente angeordnet.

Standardmäßig besteht folgende Ordnung: Ganz oben, über allen anderen Elementen, werden Legenden angeordnet (**zorder=5**), darunter Text (**zorder=3**), gefolgt von Linienelementen (**zorder=2**) und Flächen (**zorder=1**). Zuunterst finden wir schließlich eingebundene Bilder (**zorder=0**).

```
import matplotlib.pyplot as plt

x = [0, 0, 1, 2, 2]
y = [0, 2, 4, 2, 0]

plt.grid()
```

```
plt.plot(x, y, marker="o", markersize=28, linestyle=" ")
plt.plot(x, y, color="black", linewidth=5)
plt.fill(x, y, color="firebrick")
plt.show()
```

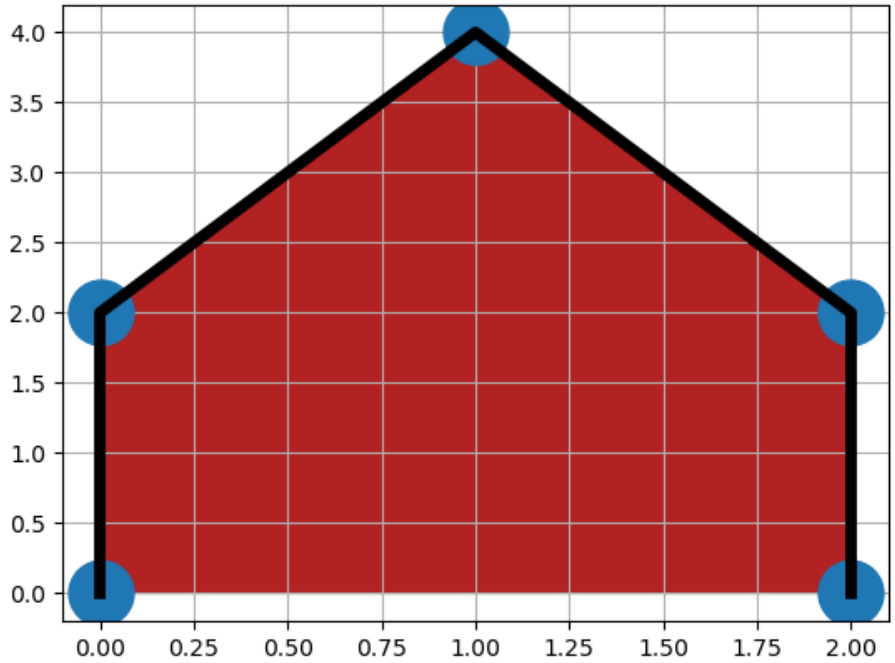


Abb. 87: Unbeeinflusste Anzeigereihenfolge

In dem Codebeispiel oben ist es ziemlich egal, in welcher Reihenfolge die Zeichenfunktionen aufgerufen werden, das Ergebnis sieht immer aus wie in Abb. 87.

Um die rote Fläche über das Gitter zu heben, die schwarze Umrandung darüber zu zeichnen und die blauen Marker ganz nach oben zu legen, ergänzen wir das Programm um mehrere **zorder**-Parameter und erhalten die Grafik in Abb. 88.

```
import matplotlib.pyplot as plt

x = [0, 0, 1, 2, 2]
y = [0, 2, 4, 2, 0]

plt.grid()
```

```
plt.plot(x, y,
         marker="o", markersize=28, linestyle=" ", zorder=9)
plt.plot(x, y, color="black", linewidth=5, zorder=8)
plt.fill(x, y, color="firebrick", zorder=7)
plt.show()
```

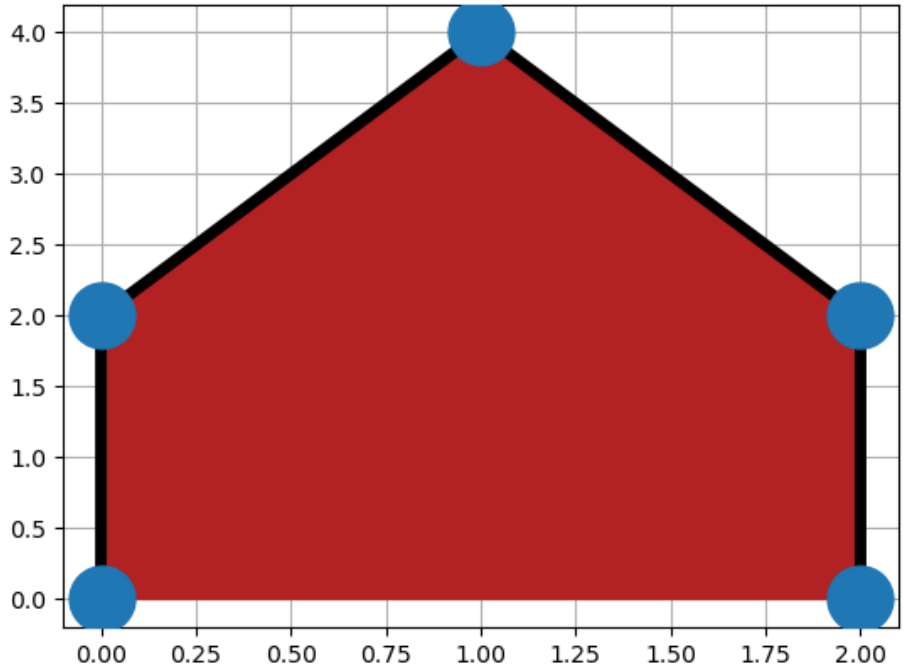


Abb. 88: Einfluss von zorder

5.23.7 3D-Diagramme

Nur wenig komplizierter ist es, 3D-Daten darzustellen, beispielsweise eine Geländeoberfläche, die durch eine „Wolke“ aus 3D-Punkten festgelegt werden soll.

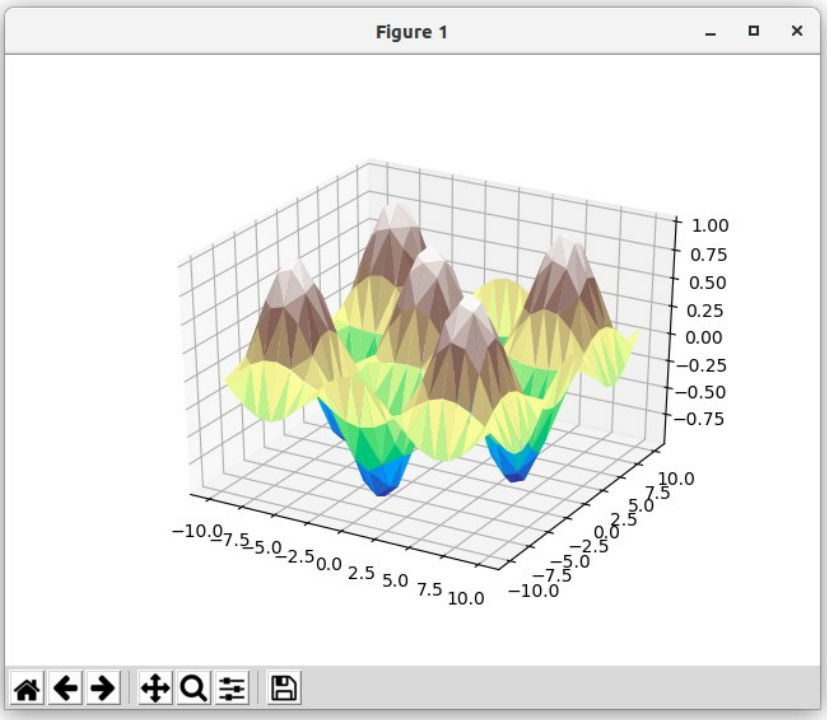


Abb. 89: x-y-z-Oberfläche mit Terrain-Farbgebung

Die 3D-Koordinaten würden wir üblicherweise aus Vermessungsdaten[1] lesen. Wir berechnen für unser Beispiel stattdessen einen kleinen „Eierkarton“ mit der Kosinusfunktion.

```
from math import cos

x = []
y = []
z = []

for xi in range(-10,11):
    for yi in range(-10,11):
        zi = cos(xi/2) * cos(yi/2)
        x.append(xi)
        y.append(yi)
        z.append(zi)
```

1 Ein Beispiel finden Sie auf https://bauforum.wirklichewelt.de/index.php?id=11643

Wir importieren wieder Matplotlib und danach(!) zusätzlich das Modul Axes3D. Die Reihenfolge ist wichtig, da beim Laden von Axes3D einige Voreinstellungen von Matplotlib überschrieben werden.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

Zunächst erzeugen wir ein leeres 3D-Achsen-Objekt, das uns als „Leinwand“ dient.

```
ax = plt.axes(projection="3d")
```

Diesem übergeben wir die zuvor erzeugten 3D-Koordinaten. Wir können die zu erzeugende Fläche auf unterschiedliche Arten einfärben[1]. Für unser Beispiel wählen wir die Farbpalette „terrain“, die Farben ähnlich einer Landkarte verwendet.

```
ax.plot_trisurf(x, y, z, cmap="terrain")
```

Das war schon alles. Wir stellen das Diagramm auf dem Bildschirm dar.

```
plt.show()
```

Die Grafik ist interaktiv. Durch Klicken und Ziehen mit der linken Maustaste können wir sie räumlich drehen und von allen Seiten betrachten. Vertikale Bewegungen mit gedrückter rechter Maustaste vergrößern und verkleinern die Darstellung.

1 Eine Liste der verfügbaren Farbpaletten finden Sie auf https://matplotlib.org/stable/tutorials/colors/colormaps.html

5.24 Grafik mit Tkinter

Mit dem Modul **`tkinter`** besitzt Python vielfältige Möglichkeiten, Grafiken auf dem Bildschirm auszugeben.

Tkinter ist dabei keine speziell für Python entwickelte Grafiklösung, vielmehr basiert es auf der schon 1988 von John Ousterhout an der Berkeley-Universität in Kalifornien entwickelten Grafik-Befehlssprache TCL (*tool command language*) und einer darauf aufbauenden Werkzeugsammlung (engl. *toolkit*), für die in vielen Programmiersprachen Schnittstellen (engl. *interfaces*) existieren. Tkinter ist dementsprechend das „*toolkit interface*“ für Python. Die von uns verwendeten Python-Funktionen werden vom Modul **`tkinter`** in TCL-Befehle umgewandelt und an den TCL-Interpreter geschickt, welcher dann für die Erzeugung der Grafik sorgt.

Im Rahmen dieses Textes können wir nur einen kleinen Streifzug durch die Welt der Grafikprogrammierung unternehmen, um zumindest die Grundlagen dieses interessanten Themenbereiches kennenzulernen.

Es ist jedoch durchaus möglich, auch ganze Anwendungsprogramme mit interaktiven grafischen Benutzungsoberflächen (*graphical user interface*, GUI) in Python zu programmieren.

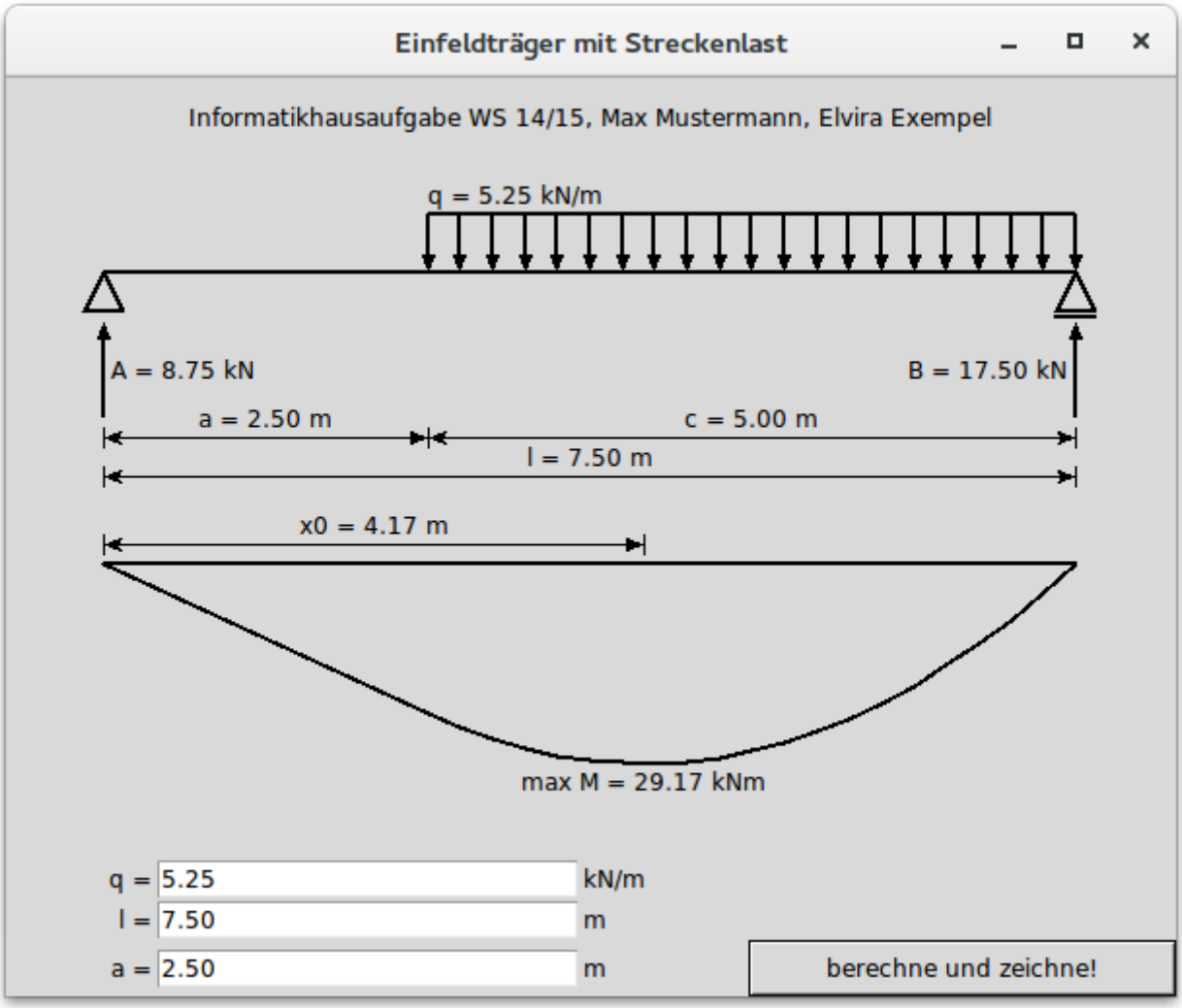


Abb. 90: GUI-Programm aus dem ersten Semester 2014/2015

Eines der ambitioniertesten Projekte dieser Art ist wahrscheinlich das Projekt PythonCAD[1], welches ein zu AutoCAD kompatibles CAD-Programm in Python realisiert, das unter Windows, Linux und macOS läuft.

5.24.1 Das Hauptfenster

Mit der Funktion **`Tk()`** laden wir den TCL-Interpreter und erzeugen ein leeres Bildschirmfenster, in dem wir anschließend alle anderen Grafikobjekte anordnen können.

1 http://sourceforge.net/projects/pythoncad/

Die Funktion **mainloop()** ist die „Hauptschleife“ des Programms. Hier werden im Hintergrund ständig Tastatureingaben und Mausaktionen abgefragt und an die Objekte im Hauptfenster weitergereicht, damit diese dazu passende Aktionen auslösen können. Siehe dazu auch Kapitel 5.25.1 („EVA und die Events“).

```
from tkinter import Tk, mainloop
Hauptfenster = Tk()
mainloop()
```

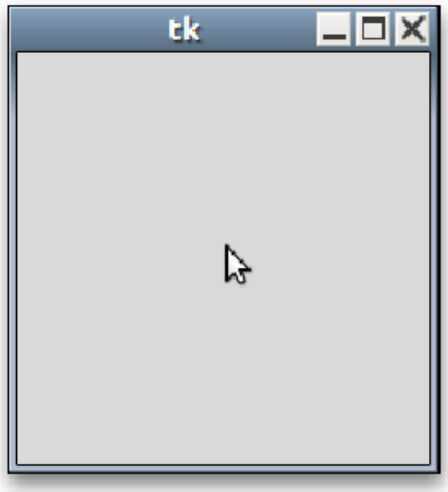


Abb. 91: Das Tk-Hauptfenster

Standardmäßig trägt das Fenster den Titel „tk“. Wir können diese Voreinstellung mit der Tk-Methode **.title()** auf einen informativeren Wert ändern.

Das Fenster passt seine Größe automatisch an den Inhalt an, darum müssen uns nicht darum kümmern, welche genauen Abmessungen es haben muss. Falls wir uns jedoch ganz sicher sind, dass es unbedingt nötig ist, die Fenstergröße zu fixieren, können wir direkt nach der Erzeugung des Fensters mithilfe seiner Methode **.geometry()** eine Größe festlegen.

Da Python die Größenangabe nicht selbst auswertet, sondern diese einfach als Zeichenkette an den TCL-Interpreter durchreicht, sieht der Funktionsparameter auf den ersten Blick etwas seltsam aus. Um ein 400 Pixel breites und 200 Pixel hohes Fenster zu erschaffen, schreiben wir nämlich nicht **.geometry(400,200)**, sondern **.geometry("400x200")**.

```
from tkinter import Tk, mainloop
Hauptfenster = Tk()
Hauptfenster.title("Mein Statikprogramm")
Hauptfenster.geometry("400x200")
mainloop()
```

In Abb. 92 ist das Ergebnis zu sehen. Die darübergelegten Maßbänder zeigen, dass sich die Größenangaben auf die innere Fensterfläche beziehen, nicht auf das gesamte Fenster mit Rahmen und Titelleiste.

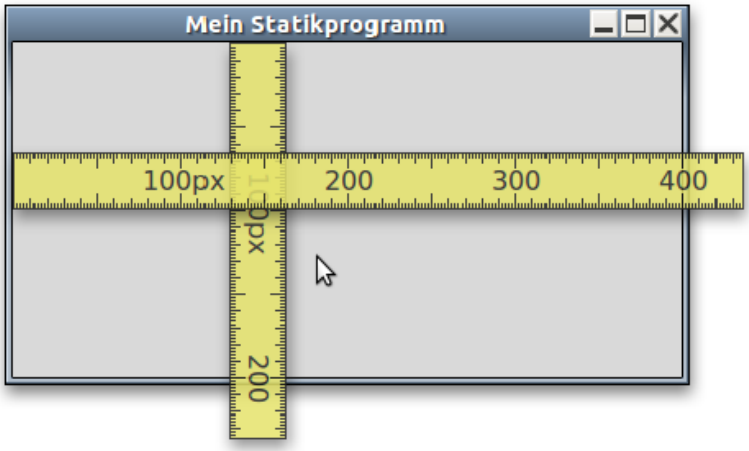


Abb. 92: Tk-Fenster mit festgelegter Größe und Überschrift.

5.24.2 untergeordnete Fenster

Ein Python-Programm kann gleichzeitig mit mehreren Fenstern arbeiten.

Zusätzliche Fenster erzeugen wir mit der Tk-Funktion **Toplevel()**. Im Gegensatz zu **Tk()** startet diese keinen eigenen TCL-Interpreter.

Jedes Fenster wird sinnvollerweise einer Variable zugeordnet. Sowohl **Tk()** als auch **Toplevel()** geben dazu eine Referenz auf das erzeugte Fenster zurück. Beim Erzeugen von Grafikobjekten geben wir diese Variable als ersten Parameter an, um das neue Objekt eindeutig einem bestimmten Fenster zuzuordnen. In dem folgenden Beispielprogramm greifen die beiden Textfelder (Label) gezielt auf das Hauptfenster und das Unterfenster zu.

```
from tkinter import Tk, Toplevel, Label, mainloop
Hauptfenster = Tk()
Hauptfenster.title("Hauptfenster")
Unterfenster = Toplevel()
Unterfenster.title("Unterfenster")
Label(Hauptfenster,
      text="-"*50+"\n"
      +"Dies ist das Hauptfenster"+"\n"
      +"unseres Programms"+"\n"
      +"-"*50
      ).pack()
Label(Unterfenster,
      text="-"*50+"\n"
      +"Dies ist ein Unterfenster"+"\n"
      +"unseres Programms"+"\n"
      +"-"*50
      ).pack()
mainloop()
```

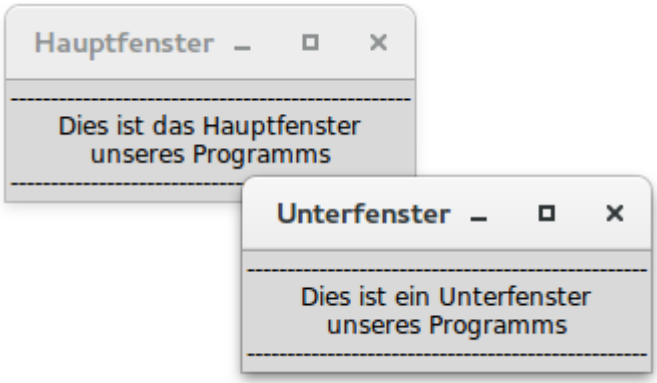


Abb. 93: Hauptfenster und Unterfenster

Solange ein Python-Programm nur ein einziges Grafikfenster verwendet, ist die Angabe eines Zielfensters nicht notwendig. In den folgenden Kapiteln wird daher nicht mehr weiter darauf eingegangen.

Der Rahmen des Bildschirmfensters und die Gestaltung der Knöpfe zum Minimieren, Maximieren und Schließen des Fensters werden vom Betriebssystem geliefert. Auf das Aussehen dieser Komponenten haben unsere Programme keinen Einfluss. Auch die Standardschriftarten werden in der Regel vom Betriebssystem vorgegeben, daher kann es bei unterschiedlichen Betriebssystemversionen stets zu leichten Abweichungen in der Bildschirmdarstellung kommen.

Abb. 94 zeigt dasselbe Programm unter sechs verschiedenen Betriebssystemversionen. Oben ist Ubuntu Linux mit verschiedenen Desktopkonfigurationen zu sehen und unten wurden die Microsoft-Windows-Versionen Windows XP, Windows 2000 und Windows 7 verwendet.

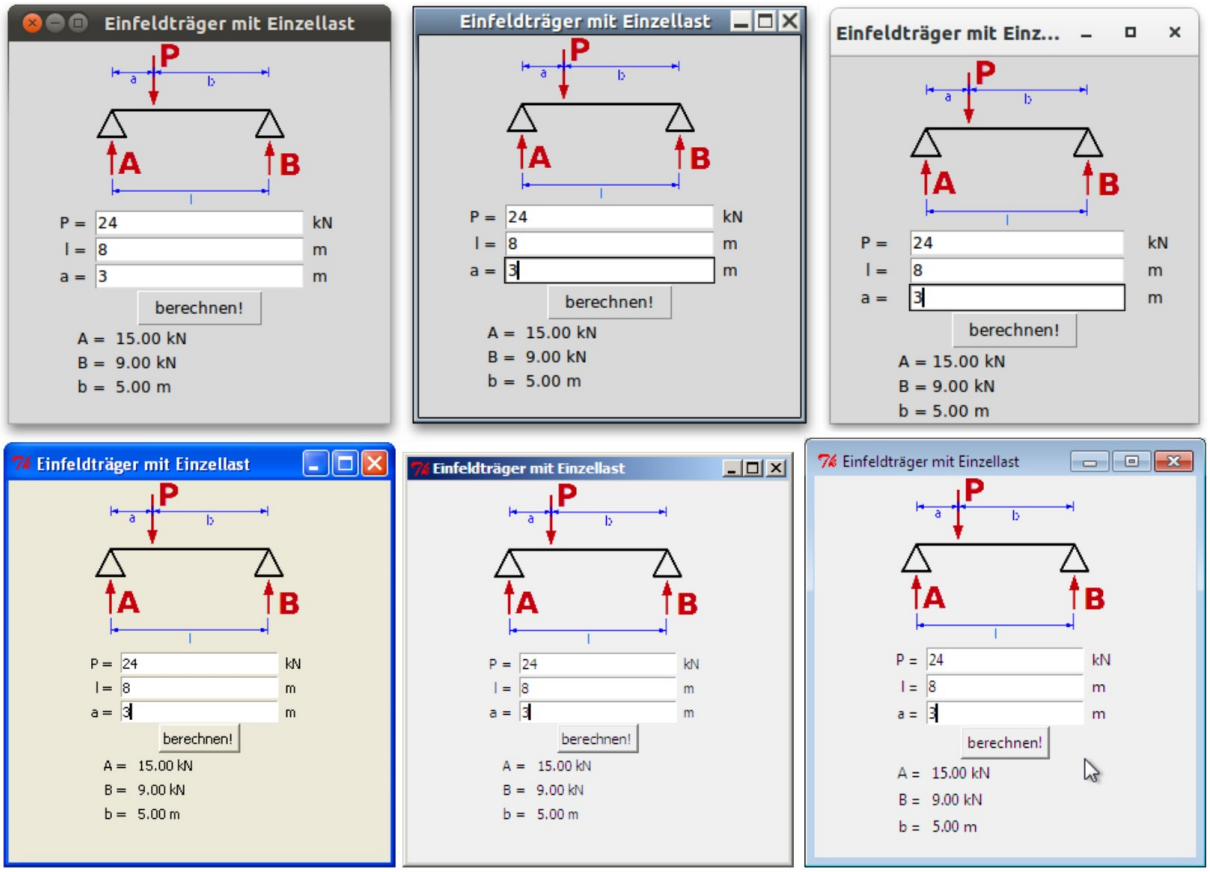


Abb. 94: Fensterdekorationen

5.24.3 Canvas – die Leinwand

In den folgenden Kapiteln werden wir einige Grafikfunktionen kennenlernen. Als „Leinwand“ (englisch: *canvas*) für unsere Grafiken dient uns ein rechteckiger Bereich auf dem Bildschirm, das Canvas-Objekt.

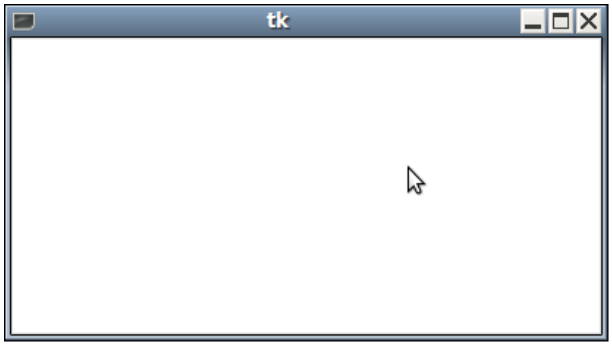


Abb. 95: Die leere Leinwand

Um ein Canvas-Objekt zu erzeugen, rufen wir die Tkinter-Funktion **`Canvas()`** auf und speichern das von ihr zurückgegebene Objekt in einer Variable.

Als Argumente der erzeugenden Funktion können wir angeben, wie hoch (**`height`**) und wie breit (**`width`**) die Leinwand sein soll, welche Hintergrundfarbe (**`bg`**) gewünscht ist und, falls notwendig, in welchem Bildschirmfenster sie angeordnet werden soll.

```
# Grafikbibliothek importieren
from tkinter import Canvas, mainloop

# Zeichenfläche einrichten
C = Canvas(width=400, height=200, bg="white")

# Zeichenfläche im Programmfenster anordnen
C.pack()
```

```
# Auf Eingaben warten
mainloop()
```

Die Methode .**pack()** ordnet die Zeichenfläche dabei in einem zuvor definierten Bildschirmfenster an. Wenn noch kein Tk-Fenster existiert, wird automatisch ein neues erzeugt, in das die Leinwand genau hineinpasst (Abb. 95).

Die „Hauptschleife“ **mainloop()** verwenden wir hier vorläufig nur, damit sich das Grafikfenster nicht sofort wieder schließt. In Programmen mit grafischen Oberflächen wird hier auf Aktionen der Benutzerin oder des Benutzers gewartet, die dann gegebenenfalls bestimmte Funktionen auslösen. Siehe dazu auch Kapitel 5.25.1 („EVA und die Events“).

5.24.4 Koordinaten der Canvas

Abb. 96: Das tk-Koordinatensystem

Die Canvas verwendet ein ebenes Koordinatensystem mit dem Nullpunkt in der oberen linken Ecke. Die Standardmaßeinheit ist das Pixel, also der einzelne Bildpunkt auf dem Monitor.

Je nach Auflösung des Bildschirms ist ein Pixel etwa zwischen 0,08 und 0,35 Millimetern groß. Nach Pixelzahl skalierte Grafiken können daher stark in der Größe variieren. Um Grafiken unabhängig von der Monitorauflösung darzustellen, können Größenangaben auch in Zentimetern, Millimetern, Zoll (25,4 mm) oder typografischen Punkten (1/72 Zoll, rund 0,35 mm) erfolgen. Den Maßzahlen wird dazu einer der Buchstaben „c“ (cm), „m“ (mm), „i“ (Zoll) oder „p“ (Punkt) angehängt.

Damit die Skalierung stimmt, muss das Betriebssystem genau wissen, wie groß die Bildschirmfläche des an den Rechner angeschlossenen Monitors ist. Häufig weiß es das jedoch entweder gar nicht oder nur in sehr grober Näherung, sodass es zu Abweichungen bei Längenangaben kommen kann.

5.24.5 Koordinatentransformationen

Funktionsgraphen, Karteninformationen und technische Zeichnungen müssen wir in aller Regel skalieren, damit sie auf dem Bildschirm in angemessener Größe wiedergegeben werden. Die Aufgabe ist es, einen rechteckigen Bereich eines x-y-Koordinatensystems, bei dem eine Achse nach rechts und die andere Achse nach oben zeigt, auf ein Bildschirmkoordinatensystem abzubilden, bei dem eine Achse nach rechts und die andere Achse nach unten zeigt.

Wenn wir, um Verwechslungen zu vermeiden, die Bildschirmkoordinaten u und v nennen, ergibt sich folgender Zusammenhang:

$$u = u_{min} + \frac{(x - x_{min}) \cdot (u_{max} - u_{min})}{x_{max} - x_{min}}$$

und

$$v = v_{min} + \frac{(y - y_{max}) \cdot (v_{max} - v_{min})}{y_{min} - y_{max}}$$

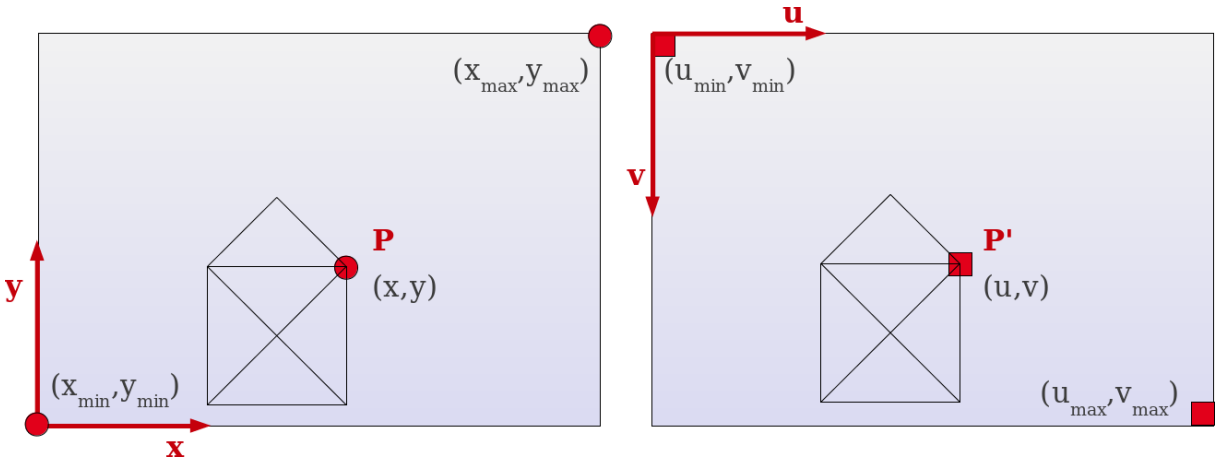


Abb. 97: Koordinatentransformation

Wir können diese Formeln ein wenig vereinfachen, wenn wir davon ausgehen, dass die obere linke Ecke des Zielkoordinatensystems (unsere Leinwand) immer die Koordinate (0,0) hat. Dann gilt:

$$u=\frac{(x-x_{min})\cdot u_{max}}{x_{max}-x_{min}} \quad \text{und} \quad v=\frac{(y-y_{max})\cdot v_{max}}{y_{min}-y_{max}}$$

5.24.6 Linien und Linienzüge

Mit der Methode **`.create_line()`** des Canvas-Objektes erzeugen wir Linien und Linienzüge. Als Argumente geben wir eine beliebige Zahl von Koordinatenpaaren an. Im einfachsten Fall rufen wir die Methode mit zwei Koordinatenpaaren auf.

Heißt unser Canvas-Objekt beispielsweise **`C`**, so erzeugt

```
C.create_line(100,150,250,300)
```

eine Linie von Punkt (100,150) zu Punkt (250,300). Die Linie ist schwarz und ein Pixel breit.

Um andere Farben und Breiten zu erhalten, verwenden wir die Attribute **`width`** und **`fill`**, denen wir eine Breite in Pixeln und eine Farbe zuordnen. Die Farbe definieren wir entweder über ihren (englischen) Farbnamen oder über einen RGB-Code.

Eine rote Linie mit fünf Pixeln Dicke wird beispielsweise durch den Befehl

```
C.create_line(100,150, 250,300, width=5, fill="red")
```

erzeugt.

Eine Liste von gültigen Farbnamen finden Sie in Kapitel 7.2 im Anhang.

Anstatt die Koordinaten als einzelne Argumente zu übergeben, können wir sie auch in eine Liste oder ein Tupel schreiben. Die Anzahl der Koordinatenpaare ist beliebig, solange mindestens ein Start- und ein Endpunkt vorhanden sind.

```
Punkte = (100,150, 250,300, 100,300, 250,150)
C.create_line(Punkte,width=5,fill="red")
```

Die Canvas-ID

Wenn wir die Befehle zum Erzeugen von Grafikobjekten direkt in der Python-Shell verwenden, stellen wir fest, dass jeder Aufruf von **`.create_line()`** eine fortlaufende Zahl erzeugt. Diese Zahl ist die Identifikationsnummer der Linie, oder kurz: ihre „ID“.

Die Canvas verwaltet eine Liste aller durch ihre Methoden erzeugten Objekte und erlaubt es, diese nachträglich einzeln oder in Gruppen unter anderem zu löschen, zu verschieben oder zu skalieren.

```
>>> from tkinter import Canvas
>>>
>>> C = Canvas(width=400,height=400,bg="white")
>>> C.pack()
>>> Punkte = (100,150, 250,300, 100,300, 250,150)
>>> C.create_line(Punkte,width=5,fill="red")
1
>>> C.update()
>>> |
```

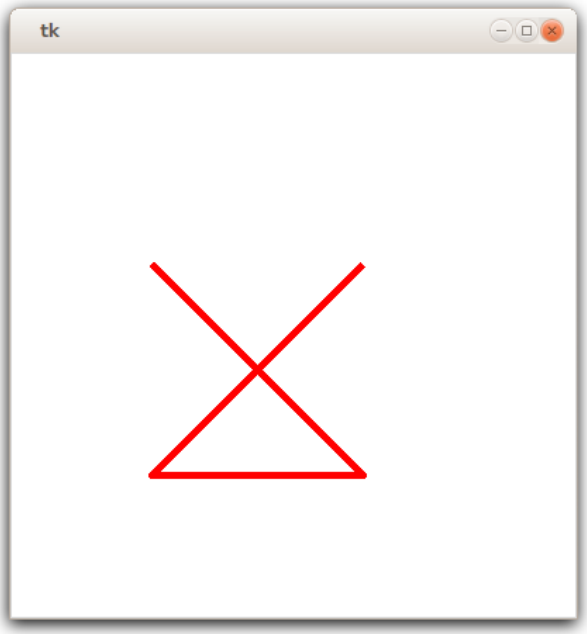


Abb. 98: Linienzug mit Breite und Farbe

Falls die verwendete Python-3-Version die eingetippten Zeichenbefehle scheinbar nicht ausgeführt, kann die Methode **.update()** aufgerufen werden, um die Leinwand zu aktualisieren.

5.24.7 Pfeilspitzen

Wir können sowohl den Anfangs- als auch den Endpunkt einer Linie oder eines Linienzuges mit einer Pfeilspitze versehen. Je nachdem, ob der erste, der letzte oder beide Endpunkte als Pfeilspitze dargestellt werden sollen, weisen wir dem Attribut **arrow** den Text **"first"**, **"last"** oder **"both"** zu. Wahlweise können wir auch eine der drei vordefinierten Konstanten **FIRST**, **LAST** oder **BOTH** verwenden.

```
C.create_line( 50,50,150,150, arrow="first")
C.create_line(100,50,200,150, arrow="last")
C.create_line(150,50,250,150, arrow="both")
```

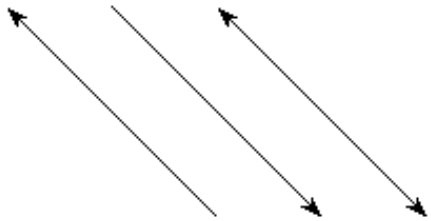


Abb. 99: Pfeilspitzen am Anfang und/oder am Ende von Linien

5.24.8 Gestrichelte Linien

Das Attribut **`dash`** erlaubt es uns, Linien auf genau definierte Art zu stricheln. Es besteht aus einer Liste von Zahlen, die abwechselnd für gezeichnete und leere Linienabschnitte einer Sequenz stehen. Die Liste **`(10,5,3,5)`** steht beispielsweise für eine strichpunktierte Linie, bei der immer wieder auf eine 10 Pixel lange Linie eine 5 Pixel breite Lücke folgt, danach eine 3 Pixel lange Linie (der „Punkt“) und zum Abschluss wieder eine 5 Pixel breite Lücke.

```
from tkinter import Canvas
C = Canvas(width=400, height=200, bg="white")
C.pack()
C.create_line(20,20, 380,20, width=3)
C.create_line(20,60, 380,60, width=3, dash=(3,7))
C.create_line(20,100, 380,100, width=3, dash=(10,))
C.create_line(20,140, 380,140, width=3, dash=(10,5,3,5))
```

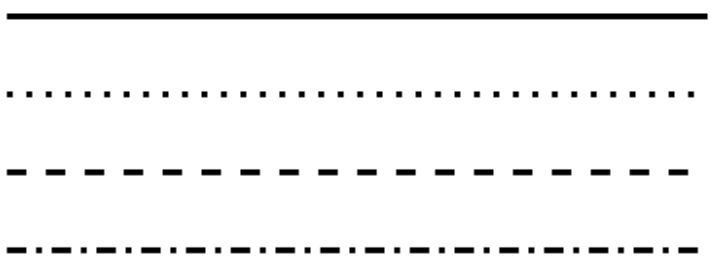


Abb. 100: Gestrichelte Linien

Bei einfachen gestrichelten Linien, deren Lücken zwischen den Strichen genauso lang sind wie die Teilstriche selbst, kann die Angabe der Lückengröße entfallen.

5.24.9 Splines (Kurvenlinien)

Splines sind Kurvenzüge, deren Teilstücke aus Parabeln bestehen. Sie lassen sich ähnlich wie Linienzüge über eine Liste von Punkten definieren. Dabei werden jedoch nur der erste und letzte Punkt tatsächlich vom Spline berührt, alle anderen Punkte sind nur Tangentenschnittpunkte der einzelnen Parabeln, deren Anfangs- und Endpunkte jeweils in der Mitte der einzelnen Abschnitte des ursprünglichen Linienzuges liegen.

Einen Linienzug können wir in ein Spline umwandeln, indem wir dem Attribut **`smooth`** den Wert **`1`** oder **`True`** zuweisen.

```
punkte = (20,20, 100,20, 20,100, 250,150,
          150,20, 380,20, 250,180, 380,180)    # Eckpunkte
C.create_line(punkte, width=3, fill="tomato")  # Linienzug
C.create_line(punkte, width=3, smooth = True)  # Spline
```

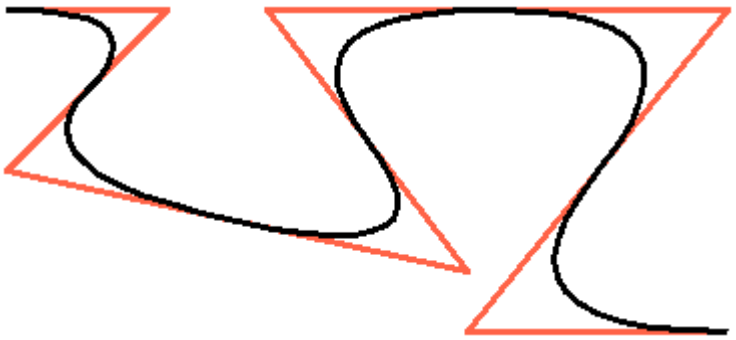


Abb. 101: Linienzug und Spline

5.24.10 Geschlossene Polygone

Geschlossene Linienzüge (Polygone) werden ganz ähnlich wie die offenen Linienzüge erzeugt. Der Unterschied besteht darin, dass wir die umschlossene Fläche mit einer Farbe füllen können.

Diese Farbe weisen wir in der Methode **.create_polygon()** dem Attribut **fill** zu. Die Linienfarbe des Polygonzugs wird über das Attribut **outline** definiert und durch **width** können wir ihm eine Breite zuweisen.

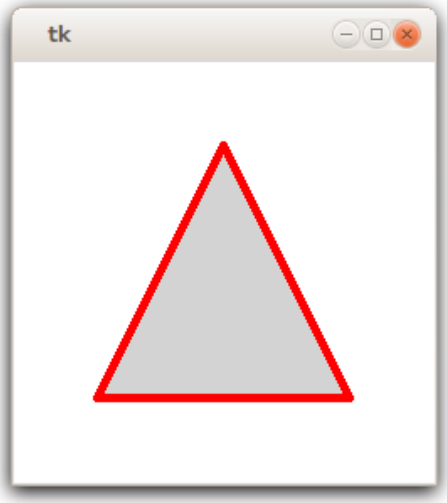


Abb. 102: Dreieck als geschlossenes Polygon

Das folgende Codebeispiel erzeugt das Dreieck in Abbildung 102.

```
from tkinter import Canvas, mainloop
C = Canvas(width=250, height=250, bg="white")
C.pack()
punkte = (50,200, 200,200, 125,50)
C.create_polygon(punkte,
                 fill="light grey", outline="red",
                 width=5)
mainloop()
```

Wenn wir das Attribut **outline** für die Randfarbe nicht angeben, wird kein Rand um das Polygon gezeichnet. Eine eventuell vorhandene Breitenangabe **width** wird dann ignoriert.

5.24.11 Rechtecke und Ellipsen

Mit nur zwei Punkten können wir sowohl Rechtecke als auch Ellipsen definieren.

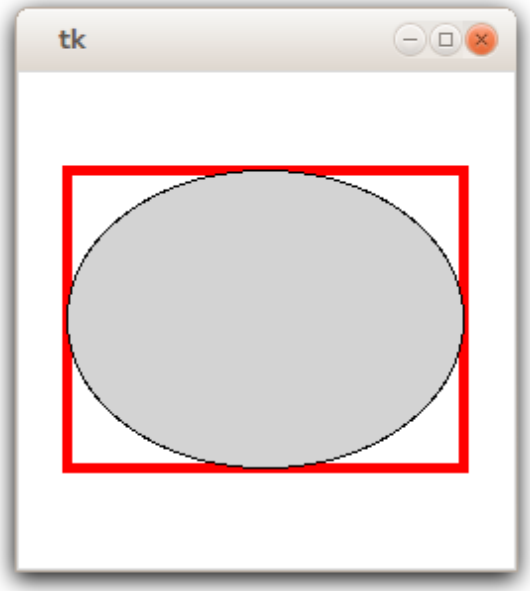


Abb. 103: Rechteck und Ellipse

Beim Rechteck sind zwei diagonal gegenüberliegende Punkte anzugeben, zwischen denen das (stets achsenparallele) Rechteck aufgespannt wird.

```
C.create_rectangle(25,50, 225,200, outline="red", width=5)
```

Ellipsen (sie heißen hier Ovale) werden durch das sie umhüllende Rechteck festgelegt.

```
C.create_oval(25,50, 225,200, fill="light grey")
```

Wenn die Randfarbe und -breite nicht angegeben werden, wird ein schwarzer Rand mit einem Pixel Breite um das Rechteck oder die Ellipse gezeichnet.

5.24.12 Kreise

Eine eigene Methode zum Zeichnen von Kreisen ist im Canvas-Objekt nicht vorhanden. Kreise sind auch nur Ovale, deren umhüllendes Rechteck zufällig ein Quadrat ist.

Wenn wir nicht für jeden neuen Kreis die Eckpunkte des umhüllenden Quadrats angeben wollen, sondern diesen lieber ganz klassisch über Mittelpunkt und Radius konstruieren, können wir dazu diese selbstgeschriebene Funktion verwenden:

```
def Kreis(C, x, y, r, width=None, fill=None, outline=None):
    return C.create_oval(x-r, y-r, x+r, y+r,
                         width=width,
                         fill=fill,
                         outline=outline)
```

`C` ist dabei die zuvor definierte Canvas, **`x`** und **`y`** sind die Koordinaten des Mittelpunktes und **`r`** ist der Radius des Kreises. Zusätzlich können noch die Attribute **`width`**, **`fill`** und **`outline`** an die erzeugende Methode **`.create_oval()`** durchgereicht werden.

5.24.13 Text

Text können wir mit der Canvas-Methode **`create_text()`** in beliebiger Größe, Farbe und Schriftart an jeder Stelle der Canvas ausgeben.

Im einfachsten Fall benötigen wir nur eine Koordinate und eine Zeichenkette. Der Text der Zeichenkette wird dann in Höhe und Breite zentriert am Einfügepunkt ausgegeben.

```
C.create_text(100, 150, text="Hallo Welt!")
```

Soll der Einfügepunkt nicht mittig im Text liegen, sondern beispielsweise unten links, so ist der untere linke Punkt des Textes als Ankerpunkt zu definieren.

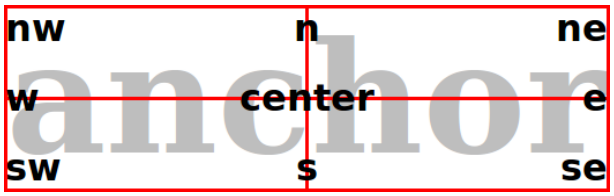


Abb. 104: Die Ankerpunkte eines Canvas-Textes

Diese Ankerpunkte orientieren sich an den Himmelsrichtungen. Anstelle von „unten links“ sagen wir dann „Südwest“ beziehungsweise „sw“:

```
C.create_text(100, 150, text="Hallo Welt!", anchor="sw")
```

Schriftart, Auszeichnung und Schriftgröße

Wenn uns die standardmäßig verwendete Schriftart zu klein oder zu freudlos ist, können wir den zu erzeugenden Text typografisch gestalten. Dazu weisen wir dem Attribut **`font`** ein Tupel zu, welches den Namen einer Schriftart, die Schriftgröße (in typografischen Punkten) und gegebenenfalls die gewünschten Auszeichnungen wie fett (*bold*) oder kursiv (*italic*) enthält.

Die Textfarbe wird über das Attribut **`fill`** geändert.

```
C.create_text(100, 150,
              text="Hallo Welt!",
              anchor="sw",
              font=("Arial", 24, "bold"),
              fill="red")
```

Dass die Schriftgröße nicht wie alle anderen Koordinatenangaben bisher in Pixeln, sondern in typografischen Punkten (1/72 Zoll) angegebenen wird, ist der Konvention geschuldet, dass in dieser Einheit die gesamte Typografiebranche arbeitet. Wer unbedingt möchte, kann die Schriftgröße auch in Pixeln angegeben. Der Größenangabe ist dann ein Minuszeichen voranzustellen.

5.25 GUI – Grafische Benutzungsoberflächen

Wollen wir menschenfreundliche Programme schreiben, die komplett über die grafische Benutzungsoberfläche (*graphical user interface*, GUI) eines eigenen Applikationsfensters bedienbar sind, benötigen wir verschiedene Arten von Ein- und Ausgabefeldern sowie Aktionsknöpfe (Buttons). Diese GUI-Objekte werden auch „Widgets“ genannt.

5.25.1 EVA und die Events

Hatten unsere Programme bisher den klassischen EVA-Aufbau „Eingabe – Verarbeitung – Ausgabe“, so betreten wir mit grafischen Benutzungsoberflächen eine ganz neue Welt. Nun arbeiten unsere Programme nicht mehr von Anfang bis Ende in einer zuvor festgelegten Reihenfolge, sondern sie reagieren auf äußere Ereignisse (engl. *events*). Das können Mausklicks sein, Tastatureingaben, das Bewegen von Schiebereglern, die Auswahl aus einem Menü und viele weitere Aktionen.

Für jede Aktion, auf die wir reagieren wollen, schreiben wir dazu eine kleine Funktion, die sich um dieses Ereignis kümmert. Diese Ereigniskümmerer werden auch im Deutschen meistens mit dem englischsprachigen Ausdruck *event handler* (Eventhandler) bezeichnet. An dieser Stelle kommt auch die von uns schon verwendete Tkinter-Funktion **`mainloop()`** wieder ins Spiel. In ihr wartet die Applikation geduldig auf Events und ruft bei deren Eintreten die passenden Eventhandler auf.

Beispiel für einen Eventhandler

Wir bauen uns unseren ersten eigenen Eventhandler und verbinden dazu einmal das Ereignis „linke Maustaste wurde gedrückt“, das in Tk den Namen **`"<Button-1>"`** trägt, mit einer Funktion, der wir den Namen **`wo_ist_die_Maus`** geben.

Als erklärtem Eventhandler wird dieser Funktion beim Aufruf ein Objekt übergeben, in dessen Attributen einige Informationen über das Ereignis enthalten sind. Traditionell gibt man diesem Objekt den Namen **`event`**. Die aktuellen Mauskoordinaten befinden sich dann in den Attributen **`event.x`** und **`event.y`**.

Unser Eventhandler soll nun nichts weiter machen, als diese Mauskoordinaten durch Aufruf der Funktion **print()** auszugeben.

```
from tkinter import Tk, mainloop

def wo_ist_die_Maus(event):
    print("Maus geklickt bei Koordinate", event.x, event.y)

T = Tk()

T.bind("<Button-1>", wo_ist_die_Maus)

mainloop()
```

Bitte achten Sie darauf, dass in der vorletzten Zeile nur der Funktionsname **wo_ist_die_Maus** ganz ohne Klammern steht, nicht der Funktionsaufruf **wo_ist_die_Maus()** mit Klammern!

Ein Programmlauf sieht dann beispielsweise so aus:

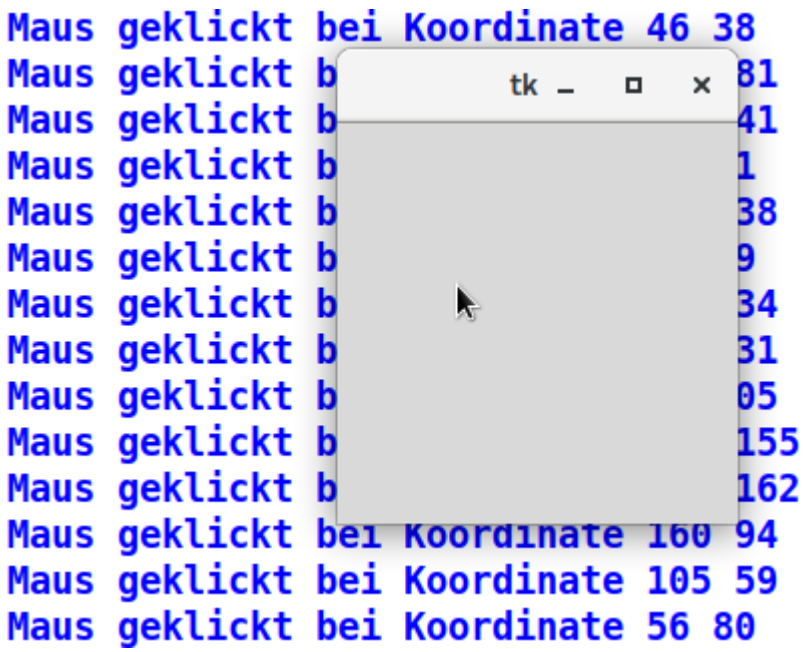


Abb. 105: Wo ist die Maus?

Die folgende Tabelle zählt einige Beispiele für Eventnamen auf, mit denen durch die Methode **.bind(***Eventname***)** eigene Eventhandler an ein Widget oder Tk-Fenster gebunden werden können.

Eventname	Bedeutung
<Button-1>	Linke Maustaste gedrückt (event.x, event.y)
<Button-2>	Mittlere Maustaste gedrückt (event.x, event.y)
<Button-3>	Rechte Maustaste gedrückt (event.x, event.y)
<Motion>	Bewegung der Maus (event.x, event.y)
<B1-Motion>	Bewegung mit gedrückter linker Maustaste (event.x, event.y)
<ButtonRelease-1>	Linke Maustaste losgelassen (event.x, event.y)
<Double-Button-1>	Linker Doppelklick (event.x, event.y)
<Enter>	Mauszeiger betritt Widget
<Leave>	Mauszeiger verlässt Widget
<Return>	Die Eingabetaste wurde gedrückt.
<KP_Enter>	Die Entertaste im Zehnerblock wurde gedrückt.
<Key>	Irgendeine Taste wurde gedrückt (event.char).
j	Die Taste „j“ wurde gedrückt.
<Up>	Die Pfeiltaste „nach oben“ wurde gedrückt. Die anderen Pfeiltastenereignisse heißen entsprechend <Down>, <Left> und <Right>.
<Shift-Up>	Die Shifttaste und die Pfeiltaste „nach oben“ wurden gemeinsam gedrückt. Der Modifikator „Shift“ kann auch Mausereignissen vorangestellt werden. <Shift-Button-3> ist dann ein Rechtsklick bei gedrückter Umschalttaste ⇧. Andere Modifikatoren sind beispielsweise „Alt“ und „Control“ (Strg).

Eventname	Bedeutung
<Prior>	Die Taste „Bild↑“ wurde gedrückt. Das Gegenstück für „Bild↓“ heißt <Next>. Andere Sondertasten sind beispielsweise <Escape> (Esc), <F1> … <F12>, <Insert> (Einfg), <Delete> (Entf), <BackSpace> (Rückschrittaste/Löschtaste), <Tab>, <Home> (Pos1) oder <End> (Ende).
<Configure>	Dieses Ereignis wird ausgelöst, wenn sich die Größe eines Widgets verändert hat, beispielsweise, weil das Fenster, in dem sich das Widget befindet, maximiert wurde. Eine Canvas könnte nun beispielsweise die auf ihr enthaltenen Grafikobjekte an die neue Größe anpassen. Diese wird in den Event-Attributen event.width und event.height übergeben.
<FocusIn>	Das Widget hat den Eingabefokus erhalten. In grafischen Oberflächen wird das aktive Eingabefeld üblicherweise durch ein Rähmchen oder eine andere Hervorhebungsart gekennzeichnet. Das Gegenstück dazu ist <FocusOut>.

Je nachdem, welcher Ereignistyp ausgelöst wurde, enthält das Event-Objekt eine Auswahl der folgenden Attribute:

Event-Attribut	Ereignistyp	Bedeutung
`widget`	alle	Instanz des auslösenden Widget-Objekts
`x, y`	Maus, Configure	Bei Mausevents: aktuelle Mauskoordinaten relativ zur linken oberen Widget-Ecke Bei Configure-Events: Koordinaten der linken oberen Widget-Ecke relativ zur linken oberen Bildschirmecke
`x_root, y_root`	Maus	Die aktuellen Mauskoordinaten relativ zur linken oberen Bildschirmecke
`char`	Tastatur	Die gedrückte Taste als Zeichen (z. B. "a")
`keysym`	Tastatur	Der Name der gedrückten Taste (z. B. "Return")
`keysym_num`	Tastatur	Die Codeposition des erzeugten Zeichens im ASCII bzw. Unicode, z. B. 65 für das Zeichen "A".
`keycode`	Tastatur	Der Tastencode der gedrückten Taste. Dieser ist unabhängig von der Tastenbeschriftung. Die Taste "y" liegt beispielsweise auf deutschen Tastaturen links vom "x" und hat dort den Tastencode 52. Auf englischen Tastaturen liegt das "y" zwischen "t" und "u" und hat den Tastencode 29.
`num`	Maus	Die gedrückte Maustaste (1, 2, 3). Weitere Maustasten werden uneinheitlich gezählt. Unter Linux werden Scrollradbewegungen beispielsweise als Maustasten 4 und 5 behandelt.

Event-Attribut	Ereignistyp	Bedeutung
`delta`	Maus	In einer perfekten Welt würde dieser Wert angeben, wie weit das Scrollrad der Maus gedreht wurde. Tatsächlich ist dieser Wert unter Linux immer null (siehe num), unter macOS gibt er an, um wie viele Rastschritte das Scrollrad gedreht wurde und unter Windows den 120-fachen Wert davon.
`width, height`	Configure	Neue Breite und Höhe des Widgets
`type`	alle	Der Ereignistyp, z.B. "ButtonPress", "KeyPress" oder "Configure"
`serial`	alle	laufende Nummer des Events
`state`	Maus, Tastatur	Integerzahl, deren einzelne Bits angeben, welche Taste beim Auslösen eines Ereignisses bereits gedrückt (oder aktiv) war. Lässt sich als Summe folgender Werte interpretieren: **`1:`** Shifttaste („Umschalttaste“) **`2:`** Feststelltaste aktiv **`4:`** Strg-Taste **`8:`** Linke Alt-Taste **`16:`** Num aktiv (Ziffernblock) **`128:`** Rechte Alt-Taste **`256:`** Linke Maustaste **`512:`** Mittlere Maustaste **`1024:`** Rechte Maustaste Bei Mausereignissen enthält **`state`** nicht den aktuellen Mausknopfdruck. Dadurch lässt sich zum Beispiel herausfinden, ob ein Doppelklick der linken Maustaste bei gedrückter rechter Maustaste erfolgte.

Event-Attribut	Ereignistyp	Bedeutung
`time`	alle	Zähler für fortlaufende Millisekunden. Damit lässt sich beispielsweise die Geschwindigkeit eines Doppelklicks oder die zeitliche Exaktheit eines rhythmischen Tastenanschlags messen.
`send_event`	alle	Dieses Attribut hat üblicherweise den Wert **`False`**.

5.25.2 Anordnung der GUI-Elemente

Alle Widgets, also alle Buttons, Ein- und Ausgabetextfelder, die Canvas und alle weiteren GUI-Elemente, sollten wir möglichst so im Programmfenster anordnen, dass Anwenderinnen und Anwender unseres Programms dessen wesentliche Funktionen mühelos und möglichst intuitiv aufrufen können.

Für Entwicklerinnen und Entwickler komplexer Programmoberflächen lohnt sich die Einarbeitung in ein GUI-Design-Programm wie beispielsweise Glade (Abb. 106), mit dem sich Tkinter-Benutzungsoberflächen gestalten und als Datei speichern lassen. Allerdings gibt es durchaus leistungsfähigere grafische Oberflächen als das mit Python mitgelieferte Tkinter. Wenn Sie wirklich ernsthafte Anwendungsentwicklung im Sinn haben, wäre dies der richtige Zeitpunkt, auf https://wiki.python.org/moin/GuiProgramming Ausschau nach passenden Werkzeugen zu halten.

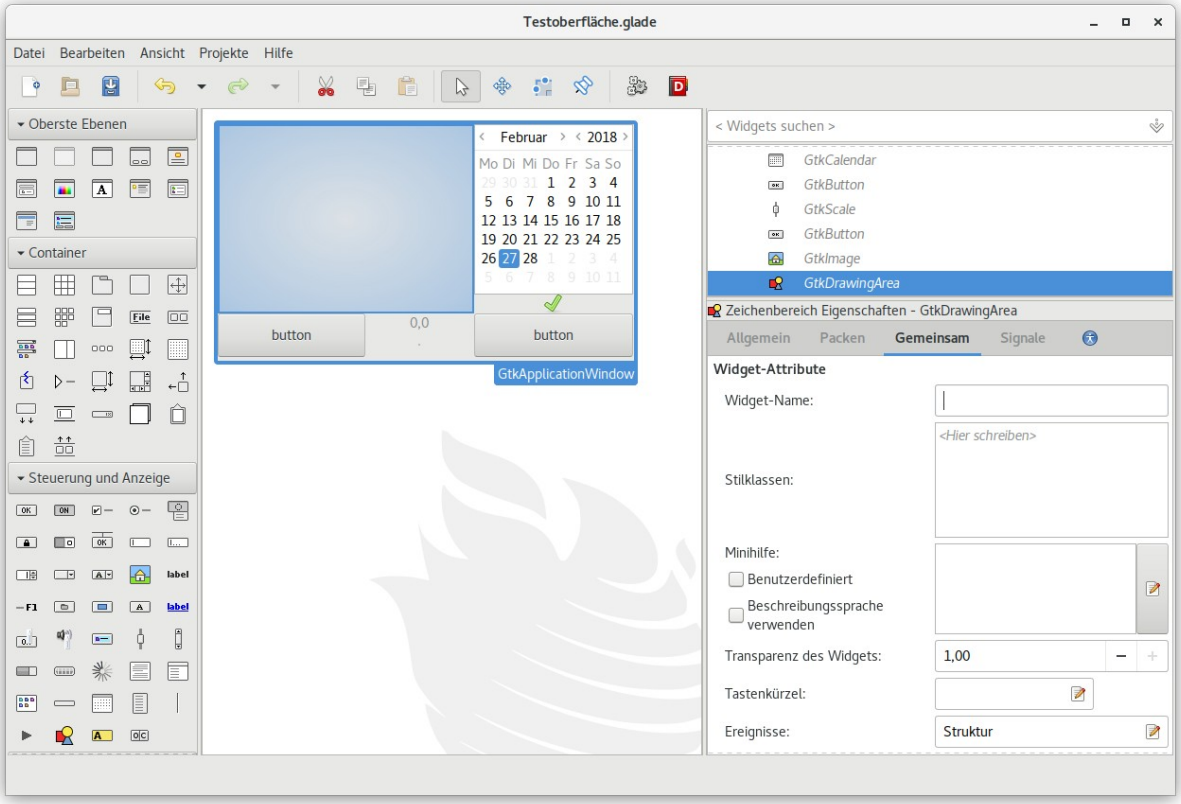


Abb. 106: Glade

Für einfache Layouts kleiner Programme, wie wir sie üblicherweise schreiben, schießen diese Lösungen jedoch übers Ziel hinaus. Mit wenigen Zeilen Quelltext kommen wir bei der Gestaltung unserer Programmoberflächen üblicherweise aus.

5.25.3 Die drei Geometriemanager

Tkinter bietet drei verschiedene Geometriemanager an, um Widgets in einem Bildschirmfenster anzuordnen. Sie werden über die Methoden **`.pack()`**, **`.place()`** und **`.grid()`** angesprochen.

Pack

Mit **`.pack()`** werden neue Elemente mittig an eine der vier Seiten der verfügbaren Fläche „gepackt“. Das Attribut **`side`** gibt an, an welchem Rand des noch unbenutzten Bereichs das neue Widget Platz für sich reservieren soll (**`left`**, **`right`**, **`top`** oder **`bottom`**). Fehlt die Angabe, wird **`side="top"`** angenommen. Widgets, die später dazukommen, werden dann darunter angeordnet.

An „seinem“ Rand kann das Widget über das Attribut **`align`** zusätzlich ausgerichtet werden. Wie an vielen Stellen in Tkinter werden hier die Himmelsrichtungen (**`n`**, **`s`**, **`e`** und **`w`**) zu Hilfe genommen.

Der horizontale und vertikale Abstand des Widgets zu anderen Elementen lässt sich mit den Attributen **`padx`** und **`pady`** festlegen. Der Abstand des Widgetrands zu seinem Inhalt wird durch **`ipadx`** und **`ipady`** definiert.

Standardmäßig belegt jedes Widget nur soviel Platz, wie es unbedingt benötigt. Wir können jedoch angeben, dass es sämtlichen verfügbaren Platz in horizontaler, vertikaler oder beiden Richtungen ausfüllen soll. Dazu setzen wir das Attribut **`fill`** auf den Wert **`"x"`**, **`"y"`** oder **`"both"`**. Setzen wir zusätzlich das Attribut **`expand`** auf den Wert **`True`**, so wird bei Größenänderungen des Applikationsfenster der entstehende Platz gleichmäßig auf alle Widgets mit diesem Attribut verteilt.

Die Vordergrundfarbe (Textfarbe) und die Hintergrundfarbe des Widgets können wir mit den Attributen **`foreground`** und **`background`** beeinflussen. Hier sind sowohl die englischsprachigen Farbnamen aus dem Anhang (Kapitel 7.2) als auch hexadezimale Farbcodes möglich.

Durch die Möglichkeit, Widgets durch Rahmen zu Gruppen zusammenzufassen, die dann gemeinsam angeordnet werden können, lassen sich auch komplexe hierarchische Layoutvorstellungen mit überschaubarem Aufwand realisieren. Der besondere Charme der Pack-Methode besteht allerdings gerade darin, diese ganzen Angaben gar nicht machen zu müssen, sondern mal eben schnell ein paar Widgets in ein Applikationsfenster packen zu können:

Abb. 107: Pack

```
C = Canvas(width=400, height=200, bg="white")
C.pack()
B1 = Button(text="zeichne einen Baum")
B1.pack()
B2 = Button(text="lösche den Baum")
B2.pack()
```

Place

Die Methode **.place()** erlaubt es UI-Designern, Widgets millimetergenau (oder pixelgenau) zu platzieren. Damit lassen sich theoretisch sehr anspruchsvolle Layouts realisieren; allerdings ist die damit verbundene Pixelzählerei enorm aufwendig, solange man nicht spezielle Layoutsoftware verwendet. Die Freude über ein gelungenes Layout währt bei dieser Methode zudem oft nur kurz. Da unterschiedliche Betriebssysteme und unterschiedliche Bildschirmauflösungen üblicherweise auch unterschiedliche Schriftgrößen mit sich bringen, ist ein pixelgenaues Layout eine hinterhältige Falle, die man möglicherweise erst bemerkt, wenn man sein in langer Nachtschicht feinjustiertes Designerschmuckstück voller Stolz weitergibt und nur betretene Blicke erntet. Wir vermeiden diese Gestaltungsmethode.

Grid

Ein gutes Verhältnis von Aufwand und Ergebnis erzielen wir mit der dritten Methode, die uns Tkinter zur Verfügung stellt. Mithilfe von **.grid()** weisen wir jedem Widget eine Zeilen- und Spaltenposition in einer flexiblen Tabelle zu, die sich automatisch an die Größe der in ihr plazierten Elemente anpasst. Mit wenigen Programmzeilen können wir so ansprechende Layouts realisieren.

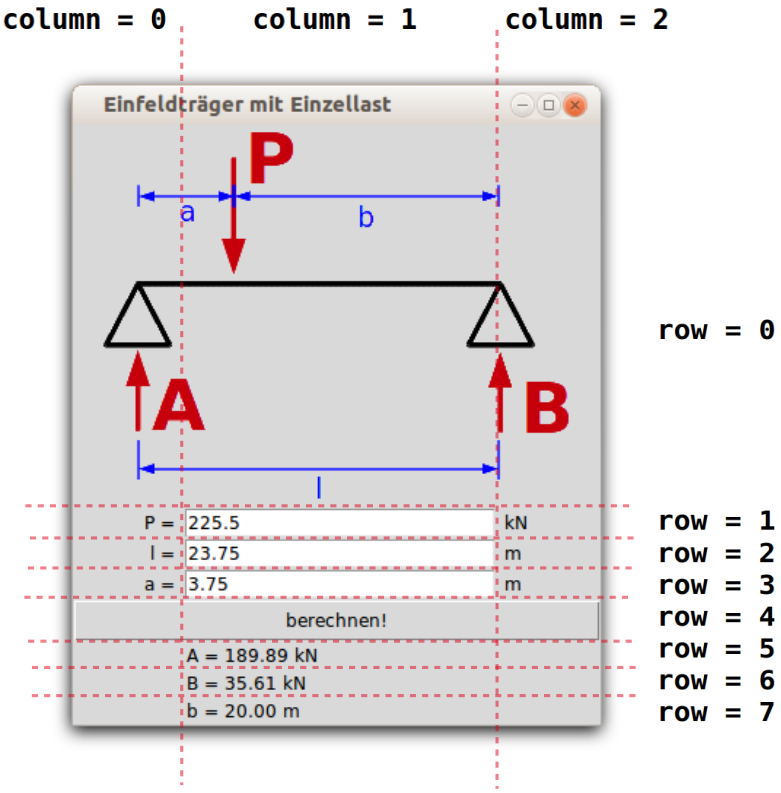


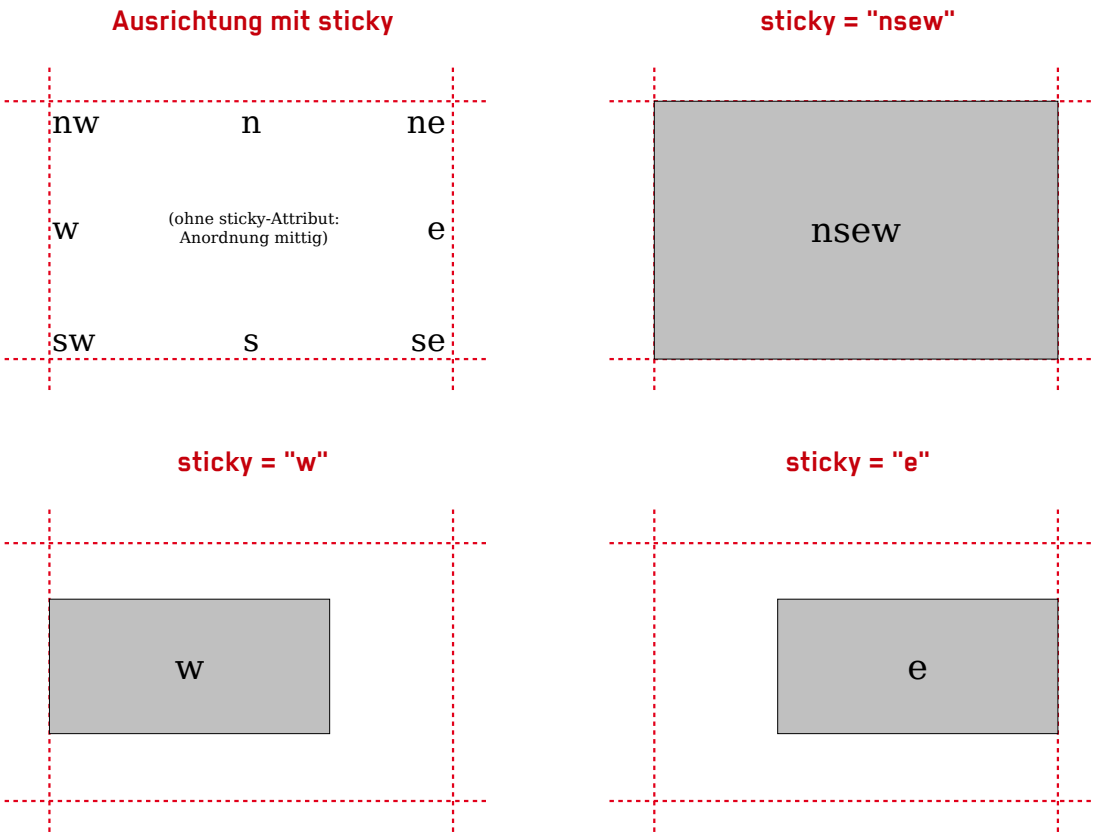
Abb. 108: Grid

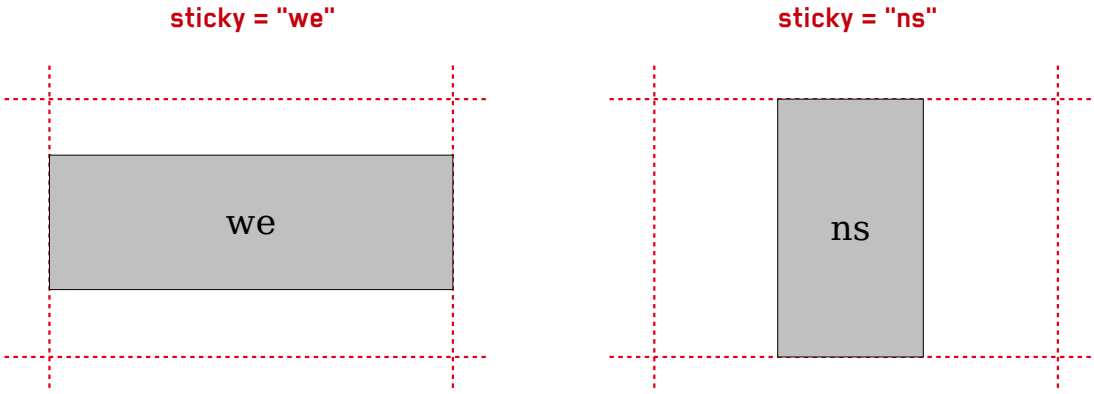
Ein paar Auszüge aus dem Quelltext der in Abb. 108 gezeigten Applikation sollen die Verwendung der Grid-Methode zur Anordnung von Widgets deutlich machen:

```
# Die Grafik stammt aus einer Datei.
bild = PhotoImage(file = "Einfeldträger.gif")
# Sie wird einem Label zugeordnet.
B = Label(image=bild)
```

```
# Dieses Bild-Label belegt drei Spalten im Grid.
B.grid(row=0, column=0, columnspan=3)
# Der Text „P = “ wird ebenfalls einem Label zugeordnet.
L1 = Label(text="P = ")
# Dieses wird rechtsbündig in Zeile 1, Spalte 0 plaziert.
L1.grid(row=1, column=0, sticky="e")
# Rechts daneben ist ein Eingabefeld, das Spalte 1 ausfüllt.
Eingabefeld_P = Entry()
Eingabefeld_P.grid(row=1, column=1, sticky="we")
# Der Button soll, wie die Grafik, dreispaltig sein.
B1 = Button(text="berechnen!")
B1.grid(row=4, column=0, columnspan=3, sticky="nsew")
```

Mit dem Parameter **sticky** legen wir fest, wie ein Widget in seinem Rasterfeld ausgerichtet wird. Die vier Buchstaben **"n"**, **"s"**, **"e"** und **"w"** stehen dabei für die vier Himmelsrichtungen auf einer Landkarte. Sie können beliebig kombiniert werden.





Die verschiedenen Geometriemanager sollten nicht gemeinsam in einem Programmfenster verwendet werden. Vor allem die Kombination von **pack** und **grid** führt dazu, dass Tkinter daran verzweifelt, gleichzeitig die Wünsche beider Manager zu erfüllen und sich „aufhängt“.

5.25.4 GUI-Widgets

Tkinter verfügt über eine Vielzahl von Widget-Typen, von denen hier nur eine Auswahl vorgestellt wird. Mit den in den folgenden Abschnitten beschriebenen Tkinter-Widgets kommen wir jedoch für die meisten einfachen GUI-Anwendungen recht gut aus[1].

Die in diesem Kapitel beschriebenen Beispiele setzen voraus, dass zuvor die verwendeten Funktionen aus dem Modul **tkinter** geladen wurden. Da wir recht viele dieser Funktionen einbinden, importieren wir sie der Einfachheit halber gleich alle.

```
from tkinter import *
```

1 Wer sich tiefer mit der Materie beschäftigen möchte, findet auf http://effbot.org/tkinterbook/tkinter-classes.htm und https://docs.python.org/3.6/library/tkinter.ttk.html eine ausführliche Beschreibung aller von Tkinter gebotenen Möglichkeiten.

Taste: Button

Der Button ist wohl das wichtigste Element einer grafischen Benutzungsoberfläche. Er enthält einen Beschriftungstext und beim Anklicken mit der Maus oder beim Aktivieren mit der Tastatur wird eine frei wählbare argumentlose Funktion aufgerufen.

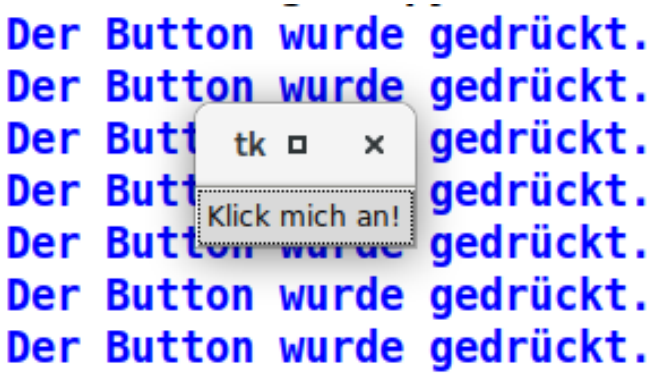


Abb. 109: Button

```
from tkinter import *

def meine_Funktion():
    print("Der Button wurde gedrückt.")

T = Tk()

B = Button(text="Klick mich an!", command=meine_Funktion)
B.pack()

mainloop()
```

Beschriftung: Label

Ein Label ist ein Widget, das einen kurzen Text und/oder oder ein Bild enthält. Das Bild wird als Objekt der Klasse tkinter.PhotoImage erwartet und kann beispielsweise den Inhalt einer GIF-Datei aufnehmen.
Falls sowohl ein Text als auch ein Bild angegeben werden, wird der Text standardmäßig nur dann ausgegeben, wenn das Bildobjekt nicht darstellbar ist.

Mit dem Attribut **`font`** lassen sich Schriftart, Größe und Auszeichnung wählen.

Die Ausrichtung des Textes wird über das Attribut **`anchor`** festgelegt. Die „Ankerpunkte“ orientieren sich an den Himmelsrichtungen.

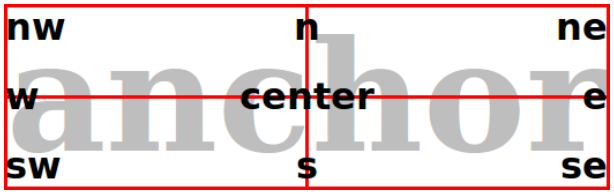


Abb. 110: Die Ankerpunkte eines Label-Textes

Unser Beispiel ordnet zwei Label nebeneinander in einem Fenster an.

Abb. 111: Text- und Image-Label

Das linke Label in Abb. 111 enthält einen zweizeiligen Text, der kursiv (*italic*) in der Schriftart Arial Black in 16 Punkt Schriftgröße gesetzt wird. Der Text ist blau und in beiden Achsen zentriert. Das Label soll 20 Pixel Abstand zu anderen Elementen lassen.

Im rechten Label sehen wir ein klassisches Gemälde. Falls es ein Problem beim Laden der Bilddatei gegeben hätte, wäre statt des Bildes ein hilfreicher Hinweistext ausgegeben worden.

```
L1 = Label(text="Ein Bild sagt mehr\nals tausend Worte.",
           font="ArialBlack 16 italic",
           foreground="blue",
           anchor="center")
L1.pack(side="left", padx=20)

try:
    Bild = PhotoImage(file="Monalisa.gif")
except:
    Bild = None

L2 = Label(image=Bild,
           text="Die Bilddatei Monalisa.gif fehlt!")
L2.pack(side="right")
```

Eine Variation des Label-Widgets ist das Message-Widget. Es wird genauso verwendet, gibt längere Texte jedoch automatisch mehrzeilig aus.

Eingabefeld: Entry

In grafischen Oberflächen verwenden wir für kurze, einzeilige Eingaben anstelle der Input-Funktion Eingabefelder, welche in Tkinter **`Entry`** heißen. Den Inhalt des Eingabefeld-Objekts erhalten wir mit dessen Methode **`.get()`** als Zeichenkette.

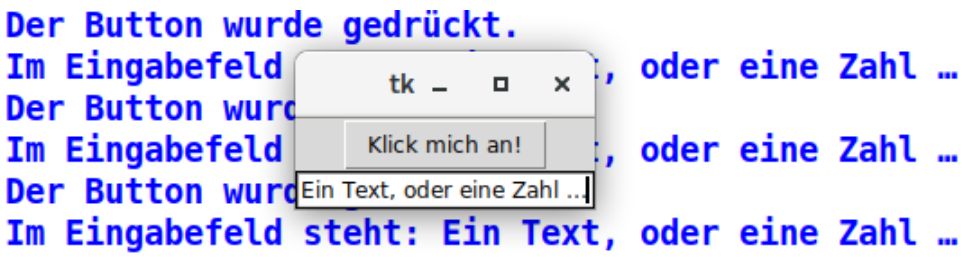


Abb. 112: Entry

```
from tkinter import *

def meine_Funktion():
    print("Der Button wurde gedrückt.")
    print("Im Eingabefeld steht:",E.get())

T = Tk()
B = Button(text="Klick mich an!", command=meine_Funktion)
B.pack()

E = Entry()
E.pack()

mainloop()
```

Die größte Umstellung gegenüber Konsolenprogrammen mit **print** und **input** ist der Verzicht auf Argumente bei der Funktion, welche die Eingaben auswertet. Diese ist nun selbst dafür verantwortlich, sich mit der Methode **.get()** die benötigten Eingabewerte aus den einzelnen Eingabefeldern der Benutzungsoberfläche zusammenzusuchen.

Falls wir mehr als ein Eingabefeld haben, sollten wir ihnen aussagekräftigere Namen als nur **E** geben. Ein vorangestelltes „E“, das darauf hinweist, dass es sich bei der entsprechenden Variable um ein Eingabefeld-Objekt handelt, schadet jedoch nicht.

```
E_Kraft = Entry()
E_Kraft.pack()
E_Hebelarm = Entry()
E_Hebelarm.pack()
```

Üblicherweise ordnet man jedem Eingabefeld mindestens ein Beschriftungsfeld zu, damit erkennbar ist, welcher Wert dort einzugeben ist.

Schieberegler: Scale

Mit Schiebereglern können wir Größen schnell mit der Maus ändern. Vor allem, wenn es nicht um exakte Eingaben, sondern eher um qualitative Größen geht, ist diese Eingabeformen sehr benutzungsfreundlich. Das Widget **`Scale`** stellt solche Schieberegler dar. Sie können horizontal oder vertikal angeordnet werden. Die Endwerte, die Schrittweite und die Art der Beschriftung sind wählbar.

```
Schieberegler steht auf: 0
Schieberegler steht auf: 0
Schieberegler steht auf: 0
Schieberegler steht auf: 12
Schieberegler steht auf: 40
Schieberegler steht auf: 76
Schieberegler steht auf: 100
Schieberegler steht auf: 50
```

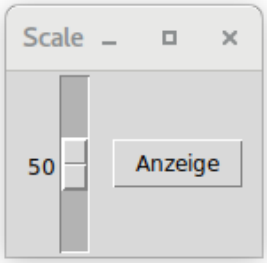


Abb. 113: Scale

In seiner einfachsten Form ist der Schieberegler senkrecht orientiert, der obere(!) Wert ist 0 und der untere 100. Der aktuelle Wert wird neben dem Schieber angezeigt. Mit der Methode **`.get()`** erhalten wir den Integer-Wert der aktuellen Einstellung.

```
from tkinter import *

def Anzeige():
    print("Schieberegler steht auf:",S.get())

T = Tk()
T.title("Scale")

S = Scale()
S.pack(side="left")

B = Button(text="Anzeige", command=Anzeige)
B.pack(side="right",padx=10,pady=10)
mainloop()
```

Wollen wir einen anderen Wertebereich abdecken, verwenden wir dazu die beiden Argumente **from_** und **to**. Beachten Sie, dass der Argumentname **from_** mit einem Unterstrich endet, um einen Konflikt mit dem Python-Schlüsselwort **from** zu vermeiden.

Um den oberen Wert auf 100 zu setzen und den unteren Wert auf 0, schreiben wir:

```
S = Scale(from_=100, to=0)
```

Die Schrittweite kann auch kleiner als 1 sein. Sie wird durch das Argument **resolution** festgelegt. Die mit **.get()** abgefragten Werte sind dann Gleitkommazahlen.

Wir können festlegen, dass bei der Bewegung des Schiebers ein Event ausgelöst wird, sodass unser Programm unmittelbar auf den neuen Schieberwert reagieren kann. Dazu weisen wir dem Argument **command** den Namen einer Funktion zu, die als Eventhandler dient.

Die Orientierung legen wir mit dem Argument **orient** fest. Gültige Werte sind **"h"**, **"horizontal"**, **"v"** und **"vertical"**.

In regelmäßigen Abständen kann die Skala des Schiebereglers beschriftet werden. Den Abstand der Zahlenwerte bestimmen wir mit dem Argument **tickinterval**.

Die Länge des Schiebereglers legen wir mit dem Argument **length** fest. Standardmäßig ist ein Schieberegler 100 Pixel lang.

Über das Argument **label** fügen wir eine die Funktion erläuternde Beschriftung hinzu.

Ein Scale-Widget kann also auch so aussehen:

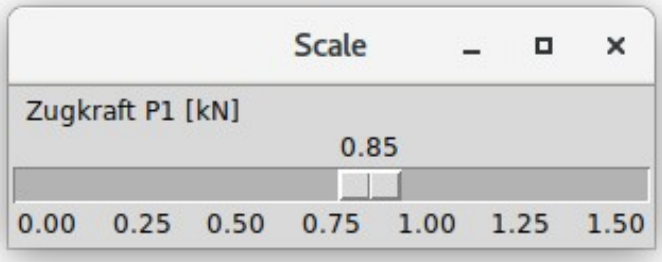


Abb. 114: Horizontales Scale-Widget

Der vollständige Quelltext zum obigen Beispiel:

```
from tkinter import *

def Anzeige(event=None):
    print("Schieberegler steht auf:",S.get())

T = Tk()
T.title("Scale")

S = Scale(from_=0, to=1.5,
          resolution=0.01,
          command=Anzeige,
          orient="h",
          tickinterval=0.25,
          length=300,
          label="Zugkraft P1 [kN]")
S.pack()

mainloop()
```

Rahmen: Frame

Der **Frame** ist ein Widget, das andere Widgets aufnehmen kann. Um die Zuordnung eines Widgets zu einem Frame kenntlich zu machen, geben wir den Namen dieses übergeordneten Frames als erstes Argument des Widgets an.

Standardmäßig sind Frames unsichtbar. Für unser Beispiel statten wir einen solchen „Positionierungsrahmen“ mit einem dicken gelben Rand aus, damit wir ihn sehen können.

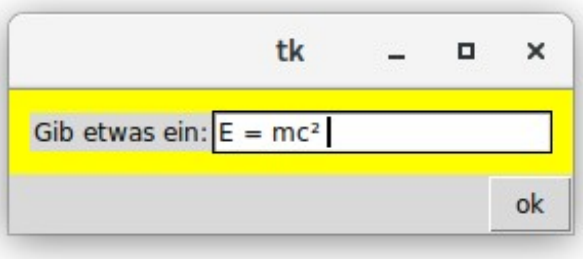


Abb. 115: Frame

Quelltextauszug:

```
F = Frame(background="yellow", borderwidth=10)
F.pack()

L = Label(F, text="Gib etwas ein:")
L.pack(side="left")

E = Entry(F)
E.pack()

B = Button(text="ok", command=meine_Funktion)
B.pack(side="right")
```

Beschrifteter Rahmen: LabelFrame

Ein LabelFrame wird wie ein Frame eingesetzt und verfügt zusätzlich über einen Rahmen mit Beschriftung.

Die Beschriftung des LabelFrames befindet sich üblicherweise auf der linken Seite des oberen Randes. Das Attribut **`labelanchor`** kann verwendet werden, um eine andere Position zu wählen:

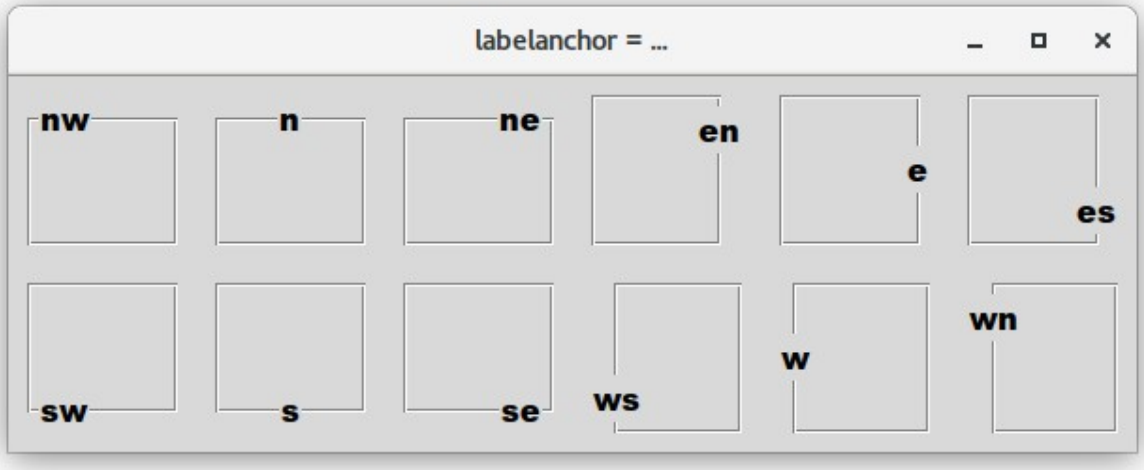


Abb. 116: Anordnung der LabelFrame-Beschriftung

LabelFrames und Frames können kombiniert werden, um Widgets ästhetisch ansprechend anzuordnen. Der Abstand zum Inhalt lässt sich mit den LabelFrame-Attributen **`padx`** und **`pady`** pixelgenau einstellen. Um den äußeren Abstand zu beeinflussen, verwenden wir im nächsten Schritt namensgleiche Attribute bei der Anordnung durch **`pack()`**.

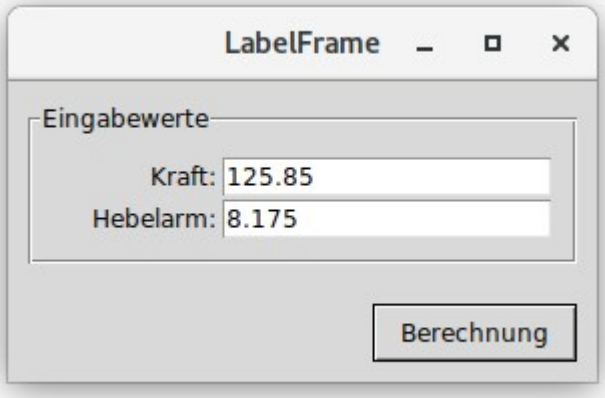


Abb. 117: LabelFrame

Zusätzlich zum sichtbaren **LabelFrame** wurden hier noch zwei gewöhnliche Frames eingesetzt, um die zeilenweise Anordnung der Label und Eingabefelder zu vereinfachen. Die Breitenangabe der Label (**width=10**) hat hier den Zweck, die Elemente sauber ausgerichtet untereinander anzuordnen.

```
from tkinter import *

T = Tk()
T.title("LabelFrame")

L = LabelFrame(text="Eingabewerte", padx=10, pady=10)
L.pack(padx=10, pady=10)

F1 = Frame(L)
F1.pack()

L1 = Label(F1, text="Kraft:", anchor="e", width=10)
L1.pack(side="left")
E_Kraft = Entry(F1)
E_Kraft.pack()

F2 = Frame(L)
F2.pack()
```

```
L2 = Label(F2, text="Hebelarm:", anchor="e", width=10)
L2.pack(side="left")
E_Hebelarm = Entry(F2)
E_Hebelarm.pack()

B = Button(text="Berechnung")
B.pack(side="right", padx=10, pady=10)

mainloop()
```

An dieser Stelle sollten wir einmal über unnötige Tipparbeit reden. Dem Quelltext grafischer Benutzungsoberflächen haftet der Ruf an, einen Hang zu ausufernder Länge und ermüdenden Wiederholungen zu haben. Tatsächlich liegt das oft nur an mangelnder Überlegung. Schauen wir uns den letzten Quelltext noch einmal an! Die Programmabschnitte für die beiden Eingabezeilen sind jeweils ein halbes Dutzend Zeilen lang und nahezu identisch. Anstatt nun für jede weitere Eingabezeile wieder sechs Zeilen Code einzutippen, schreiben wir lieber eine Funktion, die diese immergleichen Zeilen aufnimmt. Wir nennen sie der Einfachheit halber „Eingabezeile“. Als Argument erhält sie den Text des Labels und einen Hinweis auf das übergeordnete Widget (falls vorhanden). Ihr Rückgabewert soll das Entry-Objekt der Eingabezeile sein. Anstelle von sechs Codezeilen benötigen wir dank der neuen Funktion zukünftig nur noch eine:

```
from tkinter import *

def Eingabezeile(root=None, text="Eingabe"):
    F = Frame(root)
    F.pack(anchor="e")
    E = Entry(F)
    E.pack(side="right")
    L = Label(F, text=f"{text}: ", anchor="e")
    L.pack()
    return E
```

```
T = Tk()

L = LabelFrame(text="Eingabewerte")
L.pack()

E_Kraft = Eingabezeile(L,"Kraft")
E_Hebelarm = Eingabezeile(L,"Hebelarm")

B = Button(text="Berechnung")
B.pack(side="right")

mainloop()
```

Haben Sie gemerkt, dass diese Variante ohne die width-Angabe für den Text links vom Eingabefeld auskommt? Was wurde geändert?

Schiebefenster: PanedWindow

Frames und LabelFrames passen ihre Größe automatisch an ihre Inhalte an. Mit den Attributen **`fill`** und **`resize`** können wir bei Verwendung des Pack-Geometriemanagers festlegen, dass Widgets immer den gesamten verfügbaren Platz ausnutzen und sich Größenänderungen des Programmfensters oder des übergeordneten Elements anpassen.

Gelegentlich ist es jedoch sinnvoll, der Anwenderin und dem Anwender die Flächenaufteilung zu überlassen, damit diese beispielsweise einer Grafik oder einem Textbereich vorübergehend mehr Raum zuteilen können.

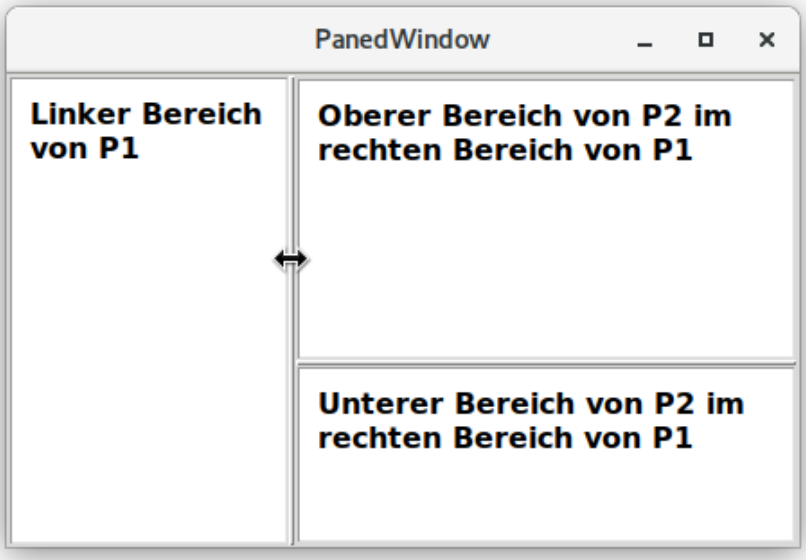


Abb. 118: PanedWindow

```
from tkinter import *

def Textpanel(root,text):
    T = Text(root, wrap="word",
             font="sans 12 bold",
             padx=10, pady=10)
    T.pack(fill="both", expand=True)
    T.insert(1.0, text)
    return T
```

```
T = Tk()
T.title("PanedWindow")

P1 = PanedWindow(
    T, orient="horizontal", sashrelief="raised")
P1.pack(fill="both", expand=True)

L_links = Textpanel(
    P1, text="Linker Bereich von P1")
P1.add(L_links)

P2 = PanedWindow(
    P1, orient="vertical", sashrelief="raised")
P1.add(P2)

L_oben = Textpanel(
    P2,
    text="Oberer Bereich von P2 im rechten Bereich von P1")
P2.add(L_oben)

L_unten = Textpanel(
    P2,
    text="Unterer Bereich von P2 im rechten Bereich von P1")
P2.add(L_unten)

mainloop()
```

Das Attribut **sashrelief** legt fest, wie der verschiebliche Trennbalken zwischen den PanedWindow-Bereichen aussehen soll. Mögliche Werte sind **"raised"** (erhaben) **"flat"** (ohne 3D-Effekt) und **"sunk"** (versenkt). Die Standardvorgabe für **sashrelief** ist **"flat"**.

Ankreuzkästchen: Checkbutton

Um unsere Anwenderinnen und Anwender einfache Ja/Nein-Entscheidungen mithilfe eines gesetzten oder nicht gesetzten Kreuzchens oder Häkchens ausführen zu lassen, verwenden wir Checkbuttons.

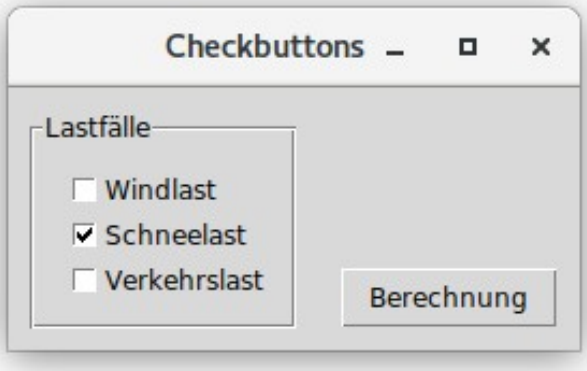


Abb. 119: Checkbuttons

Interessanterweise wird die Information darüber, ob ein Kästchen angekreuzt wurde, nicht im Widget selbst gespeichert, sondern in einem von uns separat anzulegenden Tkinter-IntVar-Objekt, dessen Wert mit **`.set()`** geschrieben und mit **`.get()`** gelesen werden kann.

Eine naheliegende Idee wäre es daher, für jeden Button einen mehrzeiligen Programmblock wie den im folgenden Quelltext gelb markierten anzulegen.

```
from tkinter import *

def Auswertung():
    print(CWv.get(), CSv.get(), CVv.get())

T = Tk()
T.title("Checkbuttons")

L = LabelFrame(text="Lastfälle", padx=10, pady=10)
L.pack(padx=10, pady=10, side="left")
```

```
CWv = IntVar()
CWv.set(0)  # Kästchen leer
CW = Checkbutton(L, text="Windlast", variable=CWv)
CW.pack(anchor="w")

CSv = IntVar()
CSv.set(1)  # Kästchen angekreuzt
CS = Checkbutton(L, text="Schneelast", variable=CSv)
CS.pack(anchor="w")

CVv = IntVar()
CV = Checkbutton(L, text="Verkehrslast", variable=CVv)
CV.pack(anchor="w")

B = Button(text="Berechnung", command=Auswertung)
B.pack(side="bottom", padx=10, pady=10)

mainloop()
```

Das *kann* man so machen; es geht aber auch deutlich übersichtlicher. Unsere neue Funktion **Checkbox()** erledigt im folgenden Programm alle wiederkehrenden Arbeiten. Sie legt das benötigte IntVar-Objekt an, weist ihm einen Anfangswert zu, zeichnet den Checkbutton und gibt das IntVar-Objekt zurück, das über den Zustand des Checkbuttons Auskunft gibt. Anschließend benötigt jeder Aufruf der Funktion **Checkbox** nur noch eine einzige Programmzeile.

```
from tkinter import *

def Auswertung():
    print(CW.get(), CS.get(), CV.get())
```

```
def Checkbox(root=None, text="", wert=0):
    IV = IntVar()
    IV.set(wert)
    C = Checkbutton(root, text=text, variable=IV)
    C.pack(anchor="w")
    return IV

T = Tk()
T.title("Checkbuttons")

L = LabelFrame(text="Lastfälle", padx=10, pady=10)
L.pack(padx=10, pady=10, side="left")

CW = Checkbox(L, "Windlast")
CS = Checkbox(L, "Schneelast", 1)
CV = Checkbox(L, "Verkehrslast")

B = Button(text="Berechnung", command=Auswertung)
B.pack(side="bottom", padx=10, pady=10)

mainloop()
```

Radiobutton

Der **Radiobutton** ist ein naher Verwandter des Checkbuttons. Während jedoch in einer Gruppe von Checkbuttons beliebig viele angekreuzt werden dürfen, ist in einer Gruppe von Radiobuttons immer nur einer aktiv. Damit erklärt sich auch der Name dieses Widgets, denn beim Radio lässt sich mit den Stationstasten ebenfalls zu jedem Zeitpunkt nur ein einzelner Sender wählen.

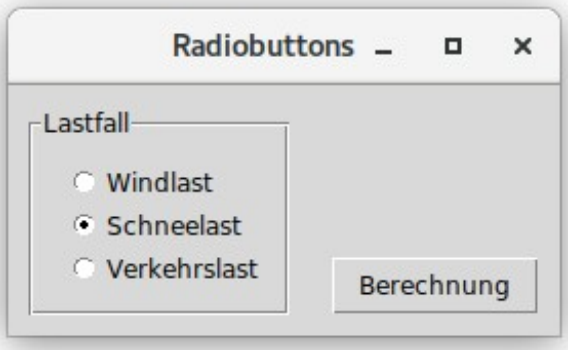


Abb. 120: Radiobuttons

Während Checkbuttons stets eine eigene Variable besitzen, die ihren Ankreuzzustand speichert, teilen sich zusammengehörende Radiobuttons eine gemeinsame Variable. Diese enthält dann den Wert des Attributs **value** des gerade gewählten Buttons.

Weil Radiobuttons immer rudelweise auftreten, schreiben wir uns eine eigene Funktion, welche die Konfiguration jedes einzelnen Radiobuttons für uns übernimmt. Unsere Funktion „Radiobuttons“ verarbeitet daher gleich eine ganze Liste von Zeichenketten, aus denen sie eine Radiotastenleiste für uns zusammenstellt. Diese Aufzählung darf beliebig lang sein.

```
from tkinter import *

def Auswertung():
    print(RB.get())
```

```
def Radiobuttons(root=None, *textliste, gewählt=None):
    TV = StringVar()
    TV.set(gewählt)
    for t in textliste:
        R = Radiobutton(root, variable=TV, text=t, value=t)
        R.pack(anchor="w")
    return TV

T = Tk()
T.title("Radiobuttons")

L = LabelFrame(text="Lastfall", padx=10, pady=10)
L.pack(padx=10, pady=10, side="left")

RB = Radiobuttons(L, "Windlast", "Schneelast",
                  "Verkehrslast", gewählt="Schneelast")

B = Button(text="Berechnung", command=Auswertung)
B.pack(side="bottom", padx=10, pady=10)

mainloop()
```

Der Wert, den ein Radiobutton zurückgibt und sein Beschriftungstext können unterschiedlich sein. Oft wird ein Programm jedoch lesbarer, wenn wir den Argumenten **value** und **text** denselben Wert zuweisen.

Menubutton

Etwas platzsparender als die Auswahl aus einer Liste von Radiobuttons ist die Verwendung eines in Tkinter **`Menubutton`** genannten Ausklappmenüs. Hier werden die verschiedenen Auswahlpunkte erst nach dem ersten Anklicken sichtbar.

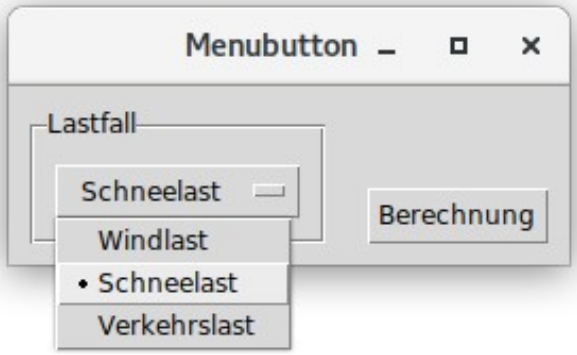


Abb. 121: ttk-Menubutton

Der Menubutton kann deutlich mehr, als nur eine kleine Auswahlliste darzustellen. Tatsächlich ist er in der Lage, komplette Menüs mit Checkbuttons, Radiobuttons, Kommandoschaltflächen und sogar Untermenüs aufnehmen. Leider wird seine Verwendung durch das Verweben mit dem Tkinter-Menu-Objekt etwas sperrig.

Unter manchen Betriebssystemen ist der Menubutton in Tkinter auf den ersten Blick nicht von einem gewöhnlichen Button zu unterscheiden. Durch Überschreiben der Funktion Menubutton mit der gleichnamigen Funktion aus dem Modul tkinter.ttk erhalten wir eine visuell ansprechendere Darstellung. Zusätzlich verfrachten wir auch hier wieder allen von unserem eigentlichen Programm ablenkenden Code in eine eigene Funktion, der wir in unserem Hauptprogramm nur mitteilen müssen, in welchem übergeordneten Widget sie untergebracht werden soll und welcher Menüpunkt voreingestellt sein soll.

```
from tkinter import *
from tkinter.ttk import Menubutton
```

```
def Auswertung():
    print(MB.get())

def Buttonmenu(root=None, *textliste, gewählt=None):
    V = StringVar()
    V.set(gewählt)
    M = Menubutton(root, textvariable=V)
    M.pack()
    M.menu = Menu(M, tearoff=0)
    M["menu"] = M.menu
    for t in textliste:
        M.menu.add_radiobutton(variable=V, label=t, value=t)
    return V

T = Tk()
T.title("Menubutton")

L = LabelFrame(text="Lastfall", padx=10, pady=10)
L.pack(padx=10, pady=10, side="left")

MB = Buttonmenu(L, "Windlast", "Schneelast", "Verkehrslast",
                gewählt="Schneelast")

B = Button(text="Berechnung", command=Auswertung)
B.pack(side="bottom", padx=10, pady=10)

mainloop()
```

Unser kleiner Widgetzoospaziergang ist damit zu Ende. Wir konnten dabei leider nur einen kleinen Teil der umfangreichen Tkinter-Bibliothek kennenlernen. Wenn Sie meinen, dass ein wichtiges Thema hier unbedingt noch aufgenommen werden sollte, melden Sie sich bei mir!

5.26 Webserver

Bisher haben wir nur Programme geschrieben, bei denen die Bedienung auf demselben Rechner stattfindet, auf dem das jeweilige Programm läuft. Wir können aber auch über das Internet auf Programme zugreifen, die auf weit entfernten Rechnern laufen. Diese Programme heißen Webserver. Sie bekommen ihre Eingaben klassischerweise aus HTML-Formularen und erzeugen wiederum HTML- und Grafikdateien, die über das Internet zurück zu den Anwenderinnen und Anwendern geschickt werden.

Viele große Websites wie Youtube oder Dropbox bestehen zu einem guten Teil aus Python-Skripten.

Um auf dem eigenen PC einen einfachen Webserver zu starten, genügt eine einzige Zeile im Terminal:

```
python -m http.server
```

Der Server meldet sich mit der verwendeten Portadresse, unter der er auf dem gastgebenden Rechner (dem sogenannten „Host“) zu erreichen ist. In der Regel wird Port 8000 verwendet.

Starten wir nun auf demselben Rechner einen Webbrowser mit der URL **http://localhost:8000**, so erhalten wir eine Antwort von unserem Webserver. Standardmäßig ist das der Inhalt der Datei index.html. Wenn diese Datei nicht existiert, wird der Inhalt des aktuellen Verzeichnisses ausgegeben.

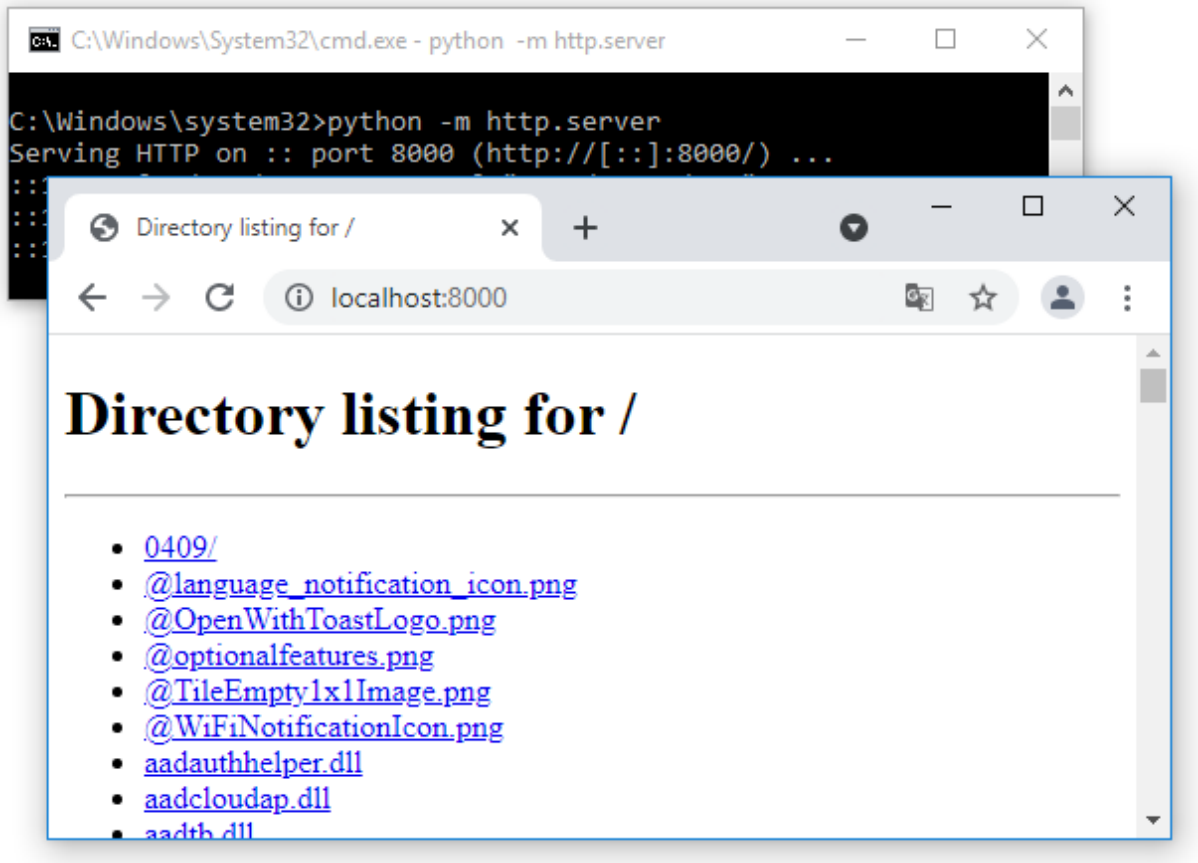


Abb. 122: Webserver unter Windows 10

5.26.1 Zeichenkodierung

Webbrowser unterstützen zwar alle Unicode-Zeichen einschließlich Emojis, trotzdem sollten Sonderzeichen, die nicht auf der Tastatur zu finden sind, derzeit noch mit Vorsicht eingesetzt werden.

Python verwendet Unicode-Zeichen nur, wenn das Betriebssystem dies auch erkennbar unterstützt. Auf manchen Webservern kann Python jedoch nicht ermitteln, ob diese Unterstützung vorhanden ist, und gibt stattdessen eine Fehlermeldung aus.

Wenn Ihr Python-Programm auch als CGI-Skript auf unzureichend konfigurierten Webservern laufen soll, sollte es daher möglichst nur ASCII-Zeichen ausgeben[1].

1 Siehe dazu „Umlaute und Unicode-Zeichen in Python-CGI-Skripten“ (https://baufo-rum.wirklichewelt.de/index.php?id=11668)

5.27 Logische Aussagen

Logische Aussagen sind Aussagen, die objektiv bestimmbar entweder eindeutig wahr oder eindeutig falsch sind. „4 ist größer als 3“ ist beispielsweise eine logische Aussage, da sie eindeutig als wahr oder falsch erkannt werden kann. „Der Hut ist schön“ ist dagegen keine logische Aussage, sondern eine subjektive Meinungsäußerung.

Für die beiden Wahrheitswerte „wahr“ und „falsch“ verwendet Python die beiden Schlüsselwörter **`True`** und **`False`**. Sie sind vom Typ **`bool`**.

5.27.1 Wahrheitswerte anderer Datentypen

In bedingten Schleifen und Fallunterscheidungen dürfen wir anstelle logischer Aussagen auch boolesche Werte und sogar Werte anderer Datentypen verwenden.

Gleichwertig zu wahren Aussagen	Gleichwertig zu falschen Aussagen
Der boolesche Wert **`True`**	Der boolesche Wert **`False`** Der Wert **`None`**
Zahlenwerte ungleich null	Die Zahl 0
Sequenzen mit mindestens einem Element	Leere Sequenzen

Die if-Abfragen in der linken und rechten Spalte der folgender Tabelle sind daher funktionsgleich:

kurze Schreibweise	redundante Schreibweise

`if b:`	`if b == True:`
	`if b != False:`
	`if b != 0:`
	`if b > "":`
	`if len(b) > 0:`

kurze Schreibweise	redundante Schreibweise

`if not b:`	`if b != True:`
	`if b == False:`
	`if b == 0:`
	`if b == "":`
	`if len(b) == 0:`
	`if b == []:`

5.27.2 Vergleichsoperatoren

Um logische Aussagen über das Größenverhältnis zweier Werte zu treffen, verwenden wir sechs verschiedene Operatoren:

ist gleich	`==`
ist ungleich	`!=`
ist kleiner als	`<`
ist kleiner oder gleich	`<=`
ist größer als	`>`
ist größer oder gleich	`>=`

5.27.3 Logische Aussagen über Gleitkommazahlen

Während ganze Zahlen immer mit ihrem exakten Wert gespeichert und verarbeitet werden, sind Gleitkommazahlen sehr oft nur Näherungswerte und unterliegen Rundungsfehlern. Intern werden Gleitkommazahlen in Python, wie auch in vielen anderen Programmiersprachen, als Produkt einer ganzen Zahl mit einer Zweierpotenz verarbeitet und erst bei der Ausgabe in eine Dezimalzahl mit einer bestimmten Anzahl von signifikanten Stellen umgewandelt. Die letzte Stelle ist dabei manchmal erstaunlich ungenau.

```
>>> 3 + 4 == 7
True
>>> 3/2 + 4/2 == 7/2
True
>>> 3/3 + 4/3 == 7/3
False
```

Schauen wir einmal genauer hin, was da bei der letzten Eingabe passiert ist und werten wir beide Seiten einzeln aus:

```
>>> 3/3 + 4/3
2.333333333333333
>>> 7/3
2.3333333333333335
```

Der Unterschied ist zwar äußerst gering, aber es *gibt* einen Unterschied. Wir sollten uns daher davor hüten, Gleitkommazahlen jemals auf Gleichheit zu testen.

Auch das Aufaddieren von Gleitkommazahlen bis zum Erreichen eines bestimmten Grenzwertes führt oft zu Überraschungen. Wer sichergehen will, dass ein selbstgeschriebenes Programm sich wie erwartet verhält, sollte besser auf Ähnlichkeit anstatt auf Gleichheit testen.

Das Mathematikmodul **math** enthält dazu die Funktion **isclose(a,b)**:

```
>>> a = 3/3 + 4/3
>>> a == 7/3
False
>>> isclose(a, 7/3)
True
```

5.27.4 Boolesche Algebra

Logische Ausdrücke lassen sich miteinander verknüpfen. So entsteht ein neuer logischer Ausdruck, der ebenfalls wieder den Wahrheitswert **True** oder **False** besitzt.

Der englische Mathematiker George Boole stellte dazu die Regeln der später nach ihm benannten booleschen Algebra auf. Ihm zu Ehren wurde die Klasse der Wahrheitswerte **True** und **False** in Python **bool** genannt.

```
>>> type(True)
<class 'bool'>
```

Die drei wichtigsten Operatoren zur Verknüpfung logischer Aussagen sind „und“, „oder“ sowie „nicht“ (in Python: **and**, **or** und **not**).

Die Konjunktion: and

Die Verknüpfung zweier logischer Aussagen mit **and** ergibt eine neue logische Aussage, die nur dann wahr ist, wenn beide Einzelaussagen wahr sind.

and	True	False
True	True	False
False	False	False

*Abb. 123: Wahrheitstabelle **and***

Dies entspricht dem üblichen Sprachgebrauch: „Wenn ich draußen bin und es regnet, dann spanne ich meinen Schirm auf.“ Der Schirm bleibt also in drei von vier Fällen geschlossen: Wenn ich drinnen bin und es regnet, wenn ich drinnen bin und es nicht regnet sowie wenn ich draußen bin und es nicht regnet.

Die Disjunktion: or

Die Verknüpfung zweier logischer Aussagen mit **`or`** ergibt eine neue logische Aussage, die nur dann falsch ist, wenn beide Einzelaussagen falsch sind. Wenn entweder die eine oder die andere oder beide Aussagen wahr sind, wird die Gesamtaussage ebenfalls wahr.

or	True	False
True	True	True
False	True	False

*Abb. 124: Wahrheitstabelle **or***

Eine gewisse Aufmerksamkeit ist angebracht, da das **`or`** durch die dritte Kombinationsmöglichkeit (beides wahr) nicht dem in Konversationen üblichen unterscheidenden Sprachgebrauch des Wortes „oder“ entspricht.

Die Antwort „ja“ auf die Frage „möchtest Du Kaffee oder Tee?“ mag aus boolescher Sicht korrekt sein (wenn der/die Befragte beides mag), verstört aber in der Regel die meisten Fragenden.[1]

1 Leider erlaubt es die offene Lizenz dieses Buches nicht, urheberrechtlich beschränktes Material einzubinden, sonst hätte ich längst mal Sydney Padua gefragt, ob ich den wundervollen Comic „Mr. Boole Comes to Tea“ aus ihrem Buch „The Thrilling Adventures of Lovelace and Babbage: The (Mostly) True Story of the First Computer“ verwenden darf. Immerhin findet sich eine Leseprobe auf Google Books: https://goo.gl/466Hfu

Die Negation: not

Der Wahrheitswert einer logischen Aussage wird durch Voranstellen von **`not`** negiert.

Der Ausdruck **`not True`** ergibt **`False`** und umgekehrt.

Vorsicht: das entspricht ganz und gar nicht dem in zwischenmenschlichen Konversationen bei Fragen üblichen Sprachgebrauch!

„Möchtest Du keinen Broccoli?“ – „Nein!“

Logische Konsequenz: ein Teller voll gesunden dampfenden Broccolis. Die logisch richtige Antwort für keinen Broccoli wäre „ja“ gewesen. Ein logisches „doch“ gibt es nicht.

Die Kontravalenz: ^

Die Kontravalenz oder exklusiv-oder-Verknüpfung ergibt immer dann den Wert **`True`**, wenn entweder die eine oder die andere Aussage wahr ist, aber nicht beide gleichzeitig.

^	True	False
True	False	True
False	True	False

Abb. 125: Wahrheitstabelle **`^`**

Diese logische Verknüpfung wird in manchen anderen Programmiersprachen mit dem Operator **`xor`** vorgenommen. In Python ist dazu, genau wie in den Programmiersprachen C oder Java, das Zirkumflex **`^`** vorgesehen, das verwirrenderweise anderenorts, zum Beispiel in der Programmiersprache BASIC und vielen Tabellenkalkulationen, als Potenzierungsoperator verwendet wird.

Prioritäten

Wie in der „Zahlenalgebra“ haben auch in der Booleschen Algebra die einzelnen Operatoren unterschiedliche Prioritäten. Bei zusammengesetzten logischen Ausdrücken besitzt **`not`** die höchste, **`and`** eine mittlere und **`or`** die geringste Priorität.

Im Zweifelsfall empfiehlt es sich, Klammern zu verwenden.

Umkehrung logischer Aussagen

Bei der Umkehrung logischer Aussagen ist Vorsicht geboten. So lautet die Negierung von **`a and b`** nicht etwa **`not a and not b`** oder gar **`not a and b`**, sondern **`not a or not b`**. Wer sich da unsicher fühlt, sollte ebenfalls besser Klammern verwenden und **`not(a and b)`** schreiben.

Boolesche Variablen

Wahrheitswerte können in Python in Variablen gespeichert werden. Diese Variablen, die entweder den Wert **`True`** oder den Wert **`False`** haben, nennen wir boolesche Variablen.

```
>>> A = 3 > 4
>>> A
False

>>> B = 7 == 7
>>> B
True
```

Beachten Sie bitte den Unterschied zwischen dem Zuweisungsoperator **`=`** („wird zu“) und dem Vergleichsoperator **`==`** („ist gleich“)!

Der Variable **`B`** wird der Wahrheitswert der Aussage **`7 == 7`**, also **`True`**, zugewiesen.

5.27.5 Venn-Diagramme

John Venn, ein Mathematiker an der Universität zu Cambridge, stellte 1880 eine Diagrammform vor, mit der sich auf sehr übersichtliche Weise logische Aussagen formulieren und überprüfen lassen.

Im einfachsten Fall bestehen Venn-Diagramme aus zwei sich überschneidenden Kreisen A und B, die jeweils einer logischen Aussage zugeordnet sind.

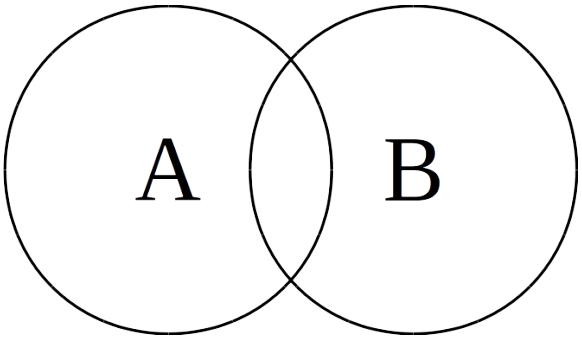


Abb. 126: Venn-Diagramm mit zwei Aussagen A und B

Alle Elemente, auf die Aussage A zutrifft, werden in Kreis A versammelt und alle Elemente, auf die Aussage B zutrifft, finden wir in Kreis B wieder. Im Schnittbereich sollten nur diejenigen Elemente zu finden sein, auf die sowohl Aussage A als auch Aussage B zutrifft.

In einem Pythonprogramm würde für diese Schnittmenge gelten:

`A and B == True`.

Die folgende Abbildung zeigt einige ausgewählte Möglichkeiten, logische Aussagen zu den Mengen A und B zu formulieren, um exakt eine bestimmte Teilmenge zu erhalten.

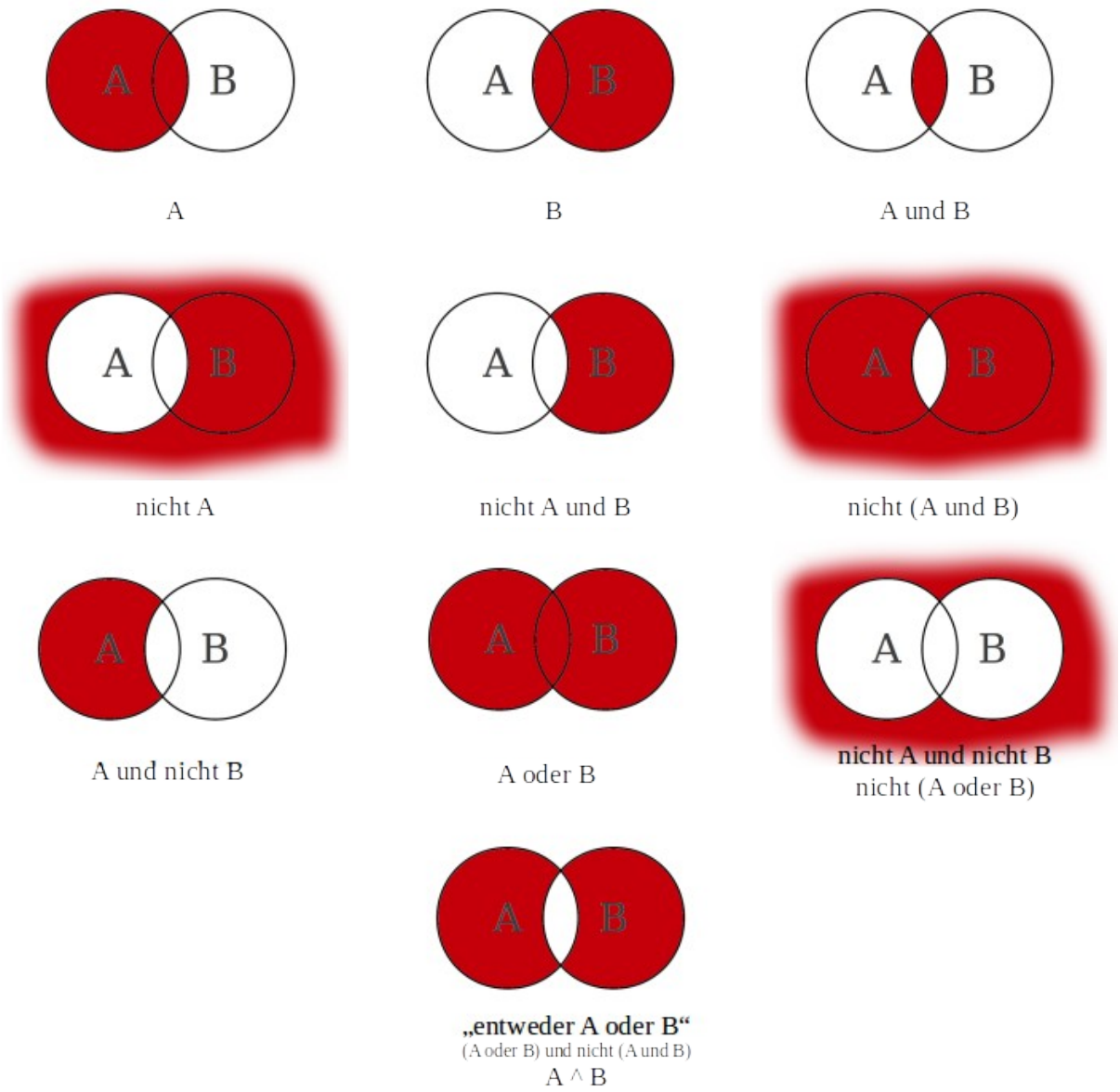


Abb. 127: Venn-Diagramme und logische Aussagen

6 Datenspeicherung und Zahlensysteme

6.1 Bits und Bytes

Wenn Sie dieses Buch als Python-Lehrbuch von vorne nach hinten durcharbeiten, sind Sie nun schon fast durch und haben immer noch nichts über die elementaren Grundbegriffe der Datenspeicherung gehört: Bit und Byte.

Das liegt vor allem daran, dass wir uns bisher vor allem auf einer „höheren Ebene“ bewegt haben, auf der es ziemlich unwichtig ist, wie Daten tatsächlich gespeichert werden. Den ganzen Ärger, den uns eine fehlerhafte Zeichenkodierung bei der Verarbeitung von Textdateien einbringt, können wir aber nur dann wirklich vermeiden, wenn wir einen tiefen Blick in den Kaninchenbau unter der Oberfläche werfen.

6.1.1 Das Bit

Das Bit ist die kleinste in der EDV verwendete Informationseinheit. Es kennt nur zwei definierte Zustände; wir können sie beispielsweise „0“ und „1“ nennen. Diese zwei Zustände können wir elektrisch recht einfach umsetzen, indem wir auf einer Leitung eine Spannung anlegen oder eben nicht.

Um ein Bit speichern zu können, müssen wir ein Medium finden, das von einem Zustand in einen anderen überführt werden kann. Wir können beispielsweise einen Knoten in eine Schnur knüpfen, ein Loch in ein Stück Karton stanzen, einen schwarzen Punkt auf ein weißes Blatt Papier malen (Abb. 128), einen Bereich einer magnetischen Oberfläche umpolen, mit einem Laser einen Spiegel matt schießen oder ein paar Moleküle elektrisch aufladen. Der Phantasie sind hier keine Grenzen gesetzt. Hauptsache, wir finden unser Bit später wieder und können zuverlässig und am besten auch noch möglichst schnell wieder herausfinden, ob wir dort eine „0“ oder eine „1“ gespeichert haben.

Abb. 128: QR-Code

6.1.2 Das Byte

Wollen wir größere Zahlen als 0 und 1 verarbeiten, müssen wir mehrere Bits kombinieren. Mit zwei Bits lassen sich schon vier verschiedene Zahlenwerte darstellen. Sind beide Bits null, stellen sie die Zahl Null dar. Eine Eins entsteht dadurch, dass eines der beiden Bits den Wert 1 annimmt. Für die Zwei wird das andere Bit auf 1 gesetzt und die Drei wird dadurch repräsentiert, dass beide Bits den Wert 1 annehmen.

Mit jedem zusätzlichen Bit lässt sich der darstellbare Zahlenbereich verdoppeln. Mit acht Bits lassen sich schon 2^8, also 256 verschiedene Zustände unterscheiden.

Das reicht beispielsweise, um allen Groß- und Kleinbuchstaben des Alphabets, den wichtigsten Satzzeichen und einigen Sonderzeichen (§$%&) eine eindeutige Codenummer zuzuweisen und dadurch Texte als Folge von Acht-Bit-Zahlenwerten speichern zu können.

In der weltweit verbreitetsten Zeichenkodierung ASCII (*american standard code for information interchange*) wird zum Beispiel dem großen „A“ die Codeposition 65 zugewiesen, dem kleinen „a“ die Position 97 und dem Leerzeichen „ “ die Position 32.

Die Zeichenkette

Hallo Bochum

wird in der ASCII-Kodierung zur Zahlenfolge

72 97 108 108 111 32 66 111 99 104 117 109.

Im Speicher des Computers werden diese Zahlen als Gruppen von jeweils acht Nullen und Einsen abgelegt:

```
H 01001000  72
a 01100001  97
l 01101100 108
l 01101100 108
o 01101111 111
, 00101100  44
  00100000  32
B 01000010  66
o 01101111 111
c 01100011  99
h 01101000 104
u 01110101 117
m 01101101 109
! 00100001  33
```

Abb. 129: ASCII-Zeichen als Bits und Bytes

Da die Kombination aus 8 Bits so wichtig für die elektronische Datenverarbeitung geworden ist, hat sie einen eigenen Namen bekommen. Wir nennen sie „Byte“. Ein Byte kann 256 verschiedene Werte annehmen.

Je mehr Bits zur Verfügung stehen, desto größer sind die verarbeitbaren Zahlenwerte. Mit 16 Bits lassen sich schon Zahlen von 0 bis 65535 darstellen, mit 32 Bits sind rund vier Milliarden unterschiedliche Kombinationen möglich und mit 64 Bits können alle ganzen Zahlen von 0 bis ungefähr $1{,}84 \times 10^{19}$ unterschieden werden.

Der 32-Bit-Beschränkung begegnen wir derzeit (2022) gelegentlich noch bei älteren PCs. Manche alte 32-Bit-Betriebssysteme können nicht einmal 4 GiB[1] RAM nutzen, ein mit dem Dateisystem FAT32 formatierter USB-Stick kann keine Dateien größer als 4294967295 Byte speichern und es gibt weltweit keine freien IP4-Adressbereiche mehr, um neue Geräte ans Internet anzuschließen, weil alle Kombinationen aus 4 Bytes schon vergeben sind.[2]

6.1.3 Das Hexadezimalsystem

Jede 8-Bit-Zahl können wir im uns vertrauten Dezimalsystem mit ein bis drei Dezimalziffern darstellen: 0 bis 255.

Mit einem kleinen Trick lässt sich eine Ziffer einsparen, indem wir anstelle des Zehnersystems das Sechzehnersystem verwenden. Zu den zehn Ziffern 0 bis 9 kommen dann die sechs Buchstaben A bis F mit den dezimalen Ziffernwerten 10 bis 15.

$0_{10} = 0_{16}$	$4_{10} = 4_{16}$	$8_{10} = 8_{16}$	$12_{10} = C_{16}$
$1_{10} = 1_{16}$	$5_{10} = 5_{16}$	$9_{10} = 9_{16}$	$13_{10} = D_{16}$
$2_{10} = 2_{16}$	$6_{10} = 6_{16}$	$10_{10} = A_{16}$	$14_{10} = E_{16}$
$3_{10} = 3_{16}$	$7_{10} = 7_{16}$	$11_{10} = B_{16}$	$15_{10} = F_{16}$

Zahlen aus zwei Hexadezimalziffern decken somit alle 16×16 möglichen Werte eines Bytes ab.

Bei mehrstelligen Hexadezimalzahlen steigt der Stellenwert nach links jeweils um den Faktor $16_{10} = 10_{16}$.

---

1 Ein GiB (Gibibyte) sind 2^{30} Byte, also rund 1,074 GB (Gigabyte). Im Microsoft-Windows-Explorer werden die Einheiten falsch verwendet und GiB-Zahlenwerte mit der Einheit GB dekoriert. Ebenso verwendet Windows die Einheit MB fälschlicherweise nicht für eine Million Bytes, sondern für 2^{20} Bytes und kB nicht für 1000 Bytes, sondern für 1024 Bytes.

2 Besonders dämlich hat sich hierbei die Firma Microsoft angestellt, die für einen Schadsoftwarescanner in ihrem Produkt „Microsoft Exchange“ das Datum und die Uhrzeit in einem vorzeichenbehafteten 32-Bit-Wert speicherte. Die größte damit darstellbare Zahl ist 2147483647. Beim Jahreswechsel 2021/2022 fielen weltweit Exchange-Server aus, weil 2112312359 damit noch darstellbar ist, 2201010000 aber nicht mehr.

Die Hexadezimalzahl 30C4 hat beispielsweise den dezimalen Wert 12484:

$$30C4_{16} = 3 \cdot 16 \cdot 16 \cdot 16 + 0 \cdot 16 \cdot 16 + 12 \cdot 16 + 4 \cdot 1 = 12484_{10}$$

Python stellt Hexadezimalzahlen als Zeichenkette dar und setzt ihnen den Präfix '0x' voran. Die Umwandlung einer solchen Zeichenkette in einen Zahlenwert geschieht mit den uns schon bekannten Standardfunktionen **eval()** oder **int()** – bei letzterer muss als zweites Argument die Zahlenbasis 16 angegeben werden.

```
>>> hex(12484)
'0x30c4'
>>> eval("0x30c4")
12484
>>> int("30c4", 16)
12484
```

Umgekehrt lassen sich ganzzahlige Werte mit der Funktion **hex()** ins Hexadezimalsystem überführen.

6.2 Zeichenkodierung – von ASCII bis Unicode

In Kapitel 6.1.2 haben wir gesehen, dass sich Buchstaben und Satzzeichen in einem Byte speichern lassen. Seit dem Jahr 1963 ist die Zuordnung der Buchstaben von A bis Z, der Ziffern von 0 bis 9 sowie häufiger Satz-, Sonder- und Steuerzeichen im ASCII festgelegt, dem siebenbittigen *American Standard Code for Information Interchange*.

+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	~	●

Abb. 130: ASCII-Code

Über Jahrzehnte wurde der ASCII mit seinen lediglich 95 verschiedenen druckbaren Zeichen unverändert benutzt, bis die Firma IBM 1981 eine Erweiterung auf 256 Zeichen für ihre ersten Personal Computer zusammenstellen ließ und dazu alle 8 Bits eines Bytes verwendete. Mit dem so hinzugewonnenen Platz in dieser IBM-PC8 getauften Kodierung konnten diese Geräte auch für die internationale Korrespondenz eingesetzt werden.

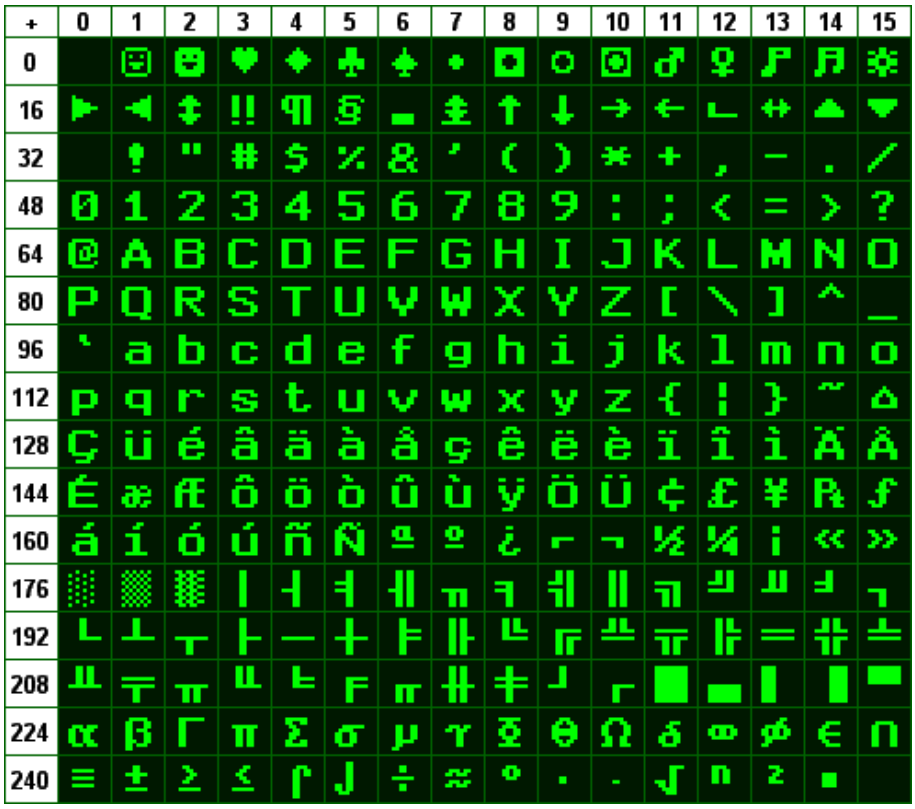


Abb. 131: Die 256 Zeichen im IBM-PC8-Zeichencode

In der Folge entstanden zahlreiche weitere Zeichenkodierungen, die den Zeichenpositionen oberhalb des ASCII immer wieder andere Bedeutungen zuwiesen – so standen sie beispielsweise für griechische oder kyrillische Buchstaben oder für eine größere Zahl von akzentuierten Buchstaben.

Mit diesen ganzen Erweiterungen erwuchs das Problem, dass der in einem Byte gespeicherte Zahlenwert nun nicht mehr eindeutig einem bestimmten Buchstaben zuzuordnen war. Zu jedem von einem zu einem anderen Rechner übertragenen Text musste daher immer angegeben werden, nach welchem System die Zeichen darin kodiert wurden. Häufig erfolgte diese Angabe nicht und Umlaute und Sonderzeichen wurden falsch dargestellt. Der Empfänger musste dann erraten, welche Kodierung vom Absender verwendet wurde.

```
>>> "Schöne Grüße!".encode("Windows-1252").decode("850")
'Sch÷ne Gr³■e!'
>>> "Schöne Grüße!".encode("Windows-1252").decode("855")
'SchШne GrЧ■e!'
>>> "Schöne Grüße!".encode("Windows-1252").decode("869")
'Schψne Grΰ■e!'
>>> "Schöne Grüße!".encode("Windows-1252").decode("maclatin2")
'SchŲne GrŁŖe!'
>>> "Schöne Grüße!".encode("Windows-1252").decode("macturkish")
'Sch^ne Gr¸şe!'
>>> "Schöne Grüße!".encode("Windows-1252").decode("Windows-1251")
'Schцne GrьЯe!'
```

Abb. 132: Windows-Umlaute

Zudem gibt es auf der Welt Zeichensysteme wie die chinesische Schrift, deren tausende Schriftzeichen sich überhaupt nicht in die 128 zusätzlichen Positionen quetschen lassen.

Ein internationales Konsortium stellte darum 1991 eine Zeichenkodierung zusammen, die alle auf der Erde verwendeten Schriftzeichen sammeln und ordnen sollte. Diese universelle Kodierung ist der Unicode[1]. Die acht Bit des ASCII-Codes wurden dazu zunächst auf 16 Bit erweitert, sodass 65536 verschiedene Codes zur Verfügung standen. Später wurden noch 16 weitere dieser 16-Bit-Ebenen hinzugefügt, um auch so exotischen Zeichen wie altägyptischen Hieroglyphen, sumerischer Keilschrift und infantilen Emojis eine Heimat zu geben.

Die Zeichencodes der Positionen 32 bis 126 entsprechen dabei den alten ASCII-Codes, sodass ASCII-Dateien ganz ohne Umwandlung auch gültige Unicode-Dateien im verbreiteten UTF-8-Format sind. Zehntausende weiterer Zeichen lassen sich in Python auf drei verschiedene Arten verwenden.

Zeichen, die sich direkt über die Tastatur eingeben lassen, wie das Eurozeichen €, können unmittelbar als Zeichenkettenkonstante in Anführungszeichen gesetzt werden. Wir können diese Zeichen auch einfach aus anderen Texten oder Webseiten herauskopieren. Alternativ kann Python Unicode-Zeichen auch über ihre Codeposition ansprechen. Das Eurozeichen hat beispielsweise die hexadezimale Unicode-Position 20ac. Schließ-

1 https://de.wikipedia.org/wiki/Unicode

lich hat jedes Unicodezeichen auch einen genormten (englischsprachigen) Namen und kann über diesen angesprochen werden. Das Eurozeichen wird hier als „EURO SIGN“ geführt.

```
>>> print("€ - \u20ac - \N{EURO SIGN}")
€ - € - €
```

Leider unterstützt das Windows-Konsolenfenster „Eingabeaufforderung“ nur einen kleinen Ausschnitt des Unicode-Zeichenumfangs, sodass man hier besser nur Zeichen verwendet, die sich auch auf einer Schreibmaschinentastatur finden lasen.

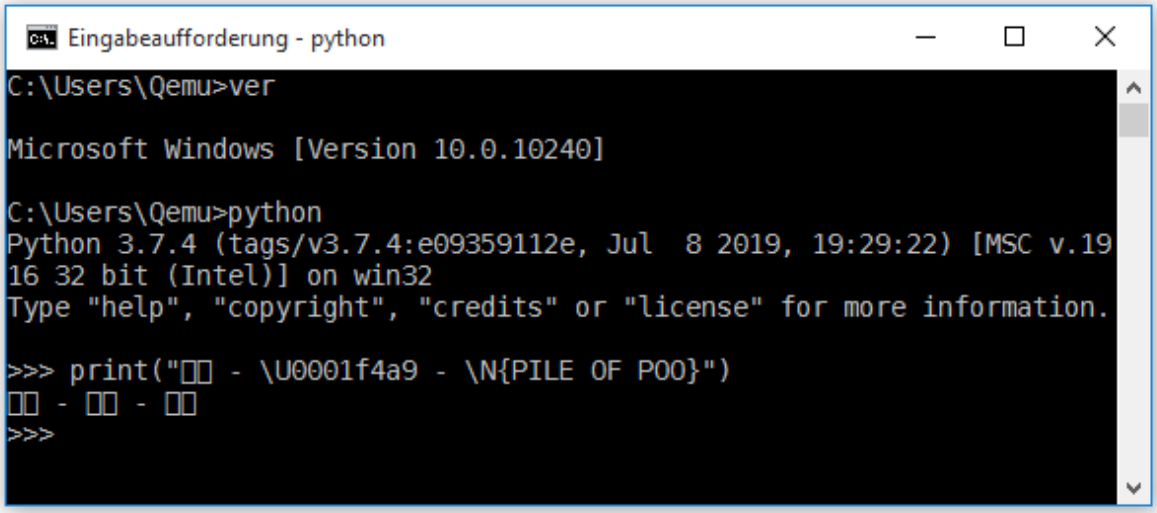


Abb. 133: Windows ist fast schon Unicode-tauglich

Seit der Version 3 von Python ist UTF-8 die standardmäßige Zeichenkodierung für alle Zeichenketten und wir müssten uns eigentlich keine weiteren Gedanken um dieses Thema machen, wenn es nicht noch viele Windowsprogramme gäbe, die nur mit den alten 256-Zeichen-Kodierungen umgehen können. Auch viele Dateiformate zum Datenaustausch verwenden immer noch die alten Windows-Kodierungen und wir müssen darauf gefasst sein, hier nötigenfalls korrigierend eingreifen zu müssen. Wie das in Python umgesetzt werden kann, sehen wir bei der Beschreibung der Methoden **`.encode()`** und **`.decode()`** in Kapitel 5.21.10.

Eine recht unterhaltsame Kurzeinführung in dieses Thema hat Thomas Scholz verfasst[1].

1 Thomas Scholz, Grundlagen der Zeichenkodierung, 2008, http://toscho.de/2008/grundlagen-zeichenkodierung/

7 Anhang

7.1 Häufige Fehlermeldungen

Fehlermeldung	Übersetzung, Bedeutung	Ursache, Behebung
`KeyboardInterrupt`	Programmabbruch über die Tastatur.	Sie haben gerade Strg C gedrückt und damit die Programmausführung abgebrochen.
`NameError: name 'xyz' is not defined`	Namensfehler: Diesen Namen gibt es noch nicht.	Vermutlich haben Sie sich bei einem Variablennamen vertippt. Achten Sie auf Groß- und Kleinschreibung! Möglicherweise haben Sie auch zwei Programmzeilen vertauscht. Sie können eine Variable nicht verwenden, bevor Sie ihr einen Wert zugewiesen haben.
`SyntaxError: invalid syntax`	Sprachlicher Fehler: Das Geschriebene ergibt keinen Sinn.	Entweder steht in der angegebenen Zeile völliger Unfug oder die Zeile darüber ist versehentlich nicht richtig abgeschlossen worden. Zählen Sie dort einmal die öffnenden und schließenden Klammern und die öffnenden und schließenden Anführungszeichen.
`UnicodeEncodeError: … can't encode character …`	Kodierungsfehler: Dieses Unicode-Zeichen gibt es hier nicht.	Eine Zeichenkette sollte in eine Kodierung überführt werden, die nicht alle enthaltenen Zeichen abdeckt. Das geschieht regelmäßig, wenn ein Python-Programm in der MS-DOS-Shell („Eingabeaufforderung“) von Microsoft Windows ausgeführt werden soll. Vermeiden Sie entweder ausgefallene Sonderzeichen oder Microsoft Windows!

Fehler-meldung	Übersetzung, Bedeutung	Ursache, Behebung
`ValueError: math domain error`	Definitionsmengenfehler: Die Eingangsgröße ist für die Funktion ungeeignet.	Die Wurzel aus -1 und der Arkussinus von 12 lassen sich nun mal nicht so einfach ausrechnen. Vermeiden Sie den Fehler durch eine Kontrolle der Eingangswerte oder fangen Sie ihn bei Auftreten ab.
`ZeroDivisionError: division by zero`	Nulldivision: Der Nenner eines Bruchs darf nicht null sein.	Sie haben versucht, durch null zu teilen. Überprüfen Sie die Eingangswerte oder fangen Sie den Fehler über eine try-except-Konstruktion ab.

7.2 Farben und Farbnamen (Auswahl)

Farbe	Rot	Grün	Blau	RGB-Code	Farbname
	240	248	255	**#F0F8FF**	alice blue
	250	235	215	**#FAEBD7**	antique white
	127	255	212	**#7FFFD4**	aquamarine
	240	255	255	**#F0FFFF**	azure
	245	245	220	**#F5F5DC**	beige
	255	228	196	**#FFE4C4**	bisque
	0	0	0	**#000000**	black
	255	235	205	**#FFEBCD**	blanched almond
	0	0	255	**#0000FF**	blue
	138	43	226	**#8A2BE2**	blue violet
	165	42	42	**#A52A2A**	brown
	222	184	135	**#DEB887**	burlywood
	95	158	160	**#5F9EA0**	cadet blue
	127	255	0	**#7FFF00**	chartreuse
	210	105	30	**#D2691E**	chocolate
	255	127	80	**#FF7F50**	coral
	100	149	237	**#6495ED**	cornflower blue
	255	248	220	**#FFF8DC**	cornsilk
	0	255	255	**#00FFFF**	cyan
	0	0	139	**#00008B**	dark blue
	0	139	139	**#008B8B**	dark cyan
	184	134	11	**#B8860B**	dark goldenrod
	0	100	0	**#006400**	dark green
	169	169	169	**#A9A9A9**	dark grey

Farbe	Rot	Grün	Blau	RGB-Code	Farbname
	189	183	107	**#BDB76B**	dark khaki
	139	0	139	**#8B008B**	dark magenta
	85	107	47	**#556B2F**	dark olive green
	255	140	0	**#FF8C00**	dark orange
	153	50	204	**#9932CC**	dark orchid
	139	0	0	**#8B0000**	dark red
	233	150	122	**#E9967A**	dark salmon
	143	188	143	**#8FBC8F**	dark sea green
	72	61	139	**#483D8B**	dark slate blue
	47	79	79	**#2F4F4F**	dark slate gray
	0	206	209	**#00CED1**	dark turquoise
	148	0	211	**#9400D3**	dark violet
	215	7	81	**#D70751**	debianred
	255	20	147	**#FF1493**	deep pink
	0	191	255	**#00BFFF**	deep sky blue
	105	105	105	**#696969**	dim gray
	30	144	255	**#1E90FF**	dodger blue
	178	34	34	**#B22222**	firebrick
	255	250	240	**#FFFAF0**	floral white
	34	139	34	**#228B22**	forest green
	220	220	220	**#DCDCDC**	gainsboro
	248	248	255	**#F8F8FF**	ghost white
	255	215	0	**#FFD700**	gold
	218	165	32	**#DAA520**	goldenrod
	190	190	190	**#BEBEBE**	gray

Farbe	Rot	Grün	Blau	RGB-Code	Farbname
	0	255	0	**#00FF00**	green
	173	255	47	**#ADFF2F**	green yellow
	240	255	240	**#F0FFF0**	honeydew
	255	105	180	**#FF69B4**	hot pink
	205	92	92	**#CD5C5C**	indian red
	255	255	240	**#FFFFF0**	ivory
	240	230	140	**#F0E68C**	khaki
	230	230	250	**#E6E6FA**	lavender
	255	240	245	**#FFF0F5**	lavender blush
	124	252	0	**#7CFC00**	lawn green
	255	250	205	**#FFFACD**	lemon chiffon
	173	216	230	**#ADD8E6**	light blue
	240	128	128	**#F08080**	light coral
	224	255	255	**#E0FFFF**	light cyan
	238	221	130	**#EEDD82**	light goldenrod
	250	250	210	**#FAFAD2**	light goldenrod yellow
	144	238	144	**#90EE90**	light green
	211	211	211	**#D3D3D3**	light grey
	255	182	193	**#FFB6C1**	light pink
	255	160	122	**#FFA07A**	light salmon
	32	178	170	**#20B2AA**	light sea green
	135	206	250	**#87CEFA**	light sky blue
	132	112	255	**#8470FF**	light slate blue
	119	136	153	**#778899**	light slate gray
	176	196	222	**#B0C4DE**	light steel blue

Farbe	Rot	Grün	Blau	RGB-Code	Farbname
	255	255	224	**#FFFFE0**	light yellow
	50	205	50	**#32CD32**	lime green
	250	240	230	**#FAF0E6**	linen
	255	0	255	**#FF00FF**	magenta
	176	48	96	**#B03060**	maroon
	102	205	170	**#66CDAA**	medium aquamarine
	0	0	205	**#0000CD**	medium blue
	186	85	211	**#BA55D3**	medium orchid
	147	112	219	**#9370DB**	medium purple
	60	179	113	**#3CB371**	medium sea green
	123	104	238	**#7B68EE**	medium slate blue
	0	250	154	**#00FA9A**	medium spring green
	72	209	204	**#48D1CC**	medium turquoise
	199	21	133	**#C71585**	medium violet red
	25	25	112	**#191970**	midnight blue
	245	255	250	**#F5FFFA**	mint cream
	255	228	225	**#FFE4E1**	misty rose
	255	228	181	**#FFE4B5**	moccasin
	255	222	173	**#FFDEAD**	navajo white
	0	0	128	**#000080**	navy
	253	245	230	**#FDF5E6**	old lace
	107	142	35	**#6B8E23**	olive drab
	255	165	0	**#FFA500**	orange
	255	69	0	**#FF4500**	orange red
	218	112	214	**#DA70D6**	orchid

Farbe	Rot	Grün	Blau	RGB-Code	Farbname
	238	232	170	**#EEE8AA**	pale goldenrod
	152	251	152	**#98FB98**	pale green
	175	238	238	**#AFEEEE**	pale turquoise
	219	112	147	**#DB7093**	pale violet red
	255	239	213	**#FFEFD5**	papaya whip
	255	218	185	**#FFDAB9**	peach puff
	205	133	63	**#CD853F**	peru
	255	192	203	**#FFC0CB**	pink
	221	160	221	**#DDA0DD**	plum
	176	224	230	**#B0E0E6**	powder blue
	160	32	240	**#A020F0**	purple
	255	0	0	**#FF0000**	red
	188	143	143	**#BC8F8F**	rosy brown
	65	105	225	**#4169E1**	royal blue
	139	69	19	**#8B4513**	saddle brown
	250	128	114	**#FA8072**	salmon
	244	164	96	**#F4A460**	sandy brown
	46	139	87	**#2E8B57**	sea green
	255	245	238	**#FFF5EE**	seashell
	160	82	45	**#A0522D**	sienna
	135	206	235	**#87CEEB**	sky blue
	106	90	205	**#6A5ACD**	slate blue
	112	128	144	**#708090**	slate gray
	255	250	250	**#FFFAFA**	snow
	0	255	127	**#00FF7F**	spring green

Farbe	Rot	Grün	Blau	RGB-Code	Farbname
	70	130	180	**#4682B4**	steel blue
	210	180	140	**#D2B48C**	tan
	216	191	216	**#D8BFD8**	thistle
	255	99	71	**#FF6347**	tomato
	64	224	208	**#40E0D0**	turquoise
	238	130	238	**#EE82EE**	violet
	208	32	144	**#D02090**	violet red
	245	222	179	**#F5DEB3**	wheat
	255	255	255	**#FFFFFF**	white
	245	245	245	**#F5F5F5**	white smoke
	255	255	0	**#FFFF00**	yellow
	154	205	50	**#9ACD32**	yellow green

Bei den Farben mit zusammengesetzten Namen ist auch jeweils die Schreibweise ohne Leerzeichen zulässig, zudem ist die Groß- und Kleinschreibung hier nicht relevant. Anstelle von „**pale violet red**“ können wir also auch „**PaleVioletRed**“ schreiben.

Von vielen Farben existieren zusätzliche Nuancen. So gibt es beispielsweise neben „**PaleVioletRed**“ auch noch „**PaleVioletRed1**“, „**PaleVioletRed2**“, „**PaleVioletRed3**“ und „**PaleVioletRed4**“.

Von der Farbe Grau (deren Name je nach Vorliebe „**gray**“ oder „**grey**“ geschrieben werden darf) gibt es gleich hundertundeins verschiedene Abstufungen von „**grey0**“ bis „**grey100**“. Ich empfehle, anstelle solcher nichtssagender Namen gleich die RGB-Codes zu verwenden.

Ein RGB-Code ist dank Pythons Formatstrings (Kapitel 5.21.8) ziemlich einfach zu erzeugen. Angenommen, die Rot-, Grün- und Blauwerte eines Farbtons befinden sich mit Werten zwischen 0 und 255 in den drei Integervariablen **r**, **g** und **b**. Der RGB-Code dazu lautet **f"#{r:02X}{g:02X}{b:02X}"**.

7.3 Der Windows-Paketmanager WinGet

Seit 2020 wird mit Windows 10 der einfache Paketmanager WinGet ausgeliefert. Die textbasierte Software greift dabei auf zentral verwaltete Kataloge (sogenannte Paketquellen) mit mehreren tausend Anwendungsprogrammen zu, die sich mit einem Befehl installieren und auch wieder entfernen lassen.

Um WinGet zu verwenden, öffnen Sie ein Textterminal („Eingabeaufforderung“) mit Administratorrechten. Dazu drücken Sie die Windowstaste, tippen CMD und klicken dann entweder den gefundenen Eintrag mit der rechten Maustaste an und wählen „Als Administrator ausführen“ oder Sie drücken die Tastenkombination Strg ⇧ ↲.

Wenn Sie im Terminalfenster „winget“ eingeben, erhalten Sie einen kurzen Überblick über die verwendbaren Kommandos (Abb. 134).

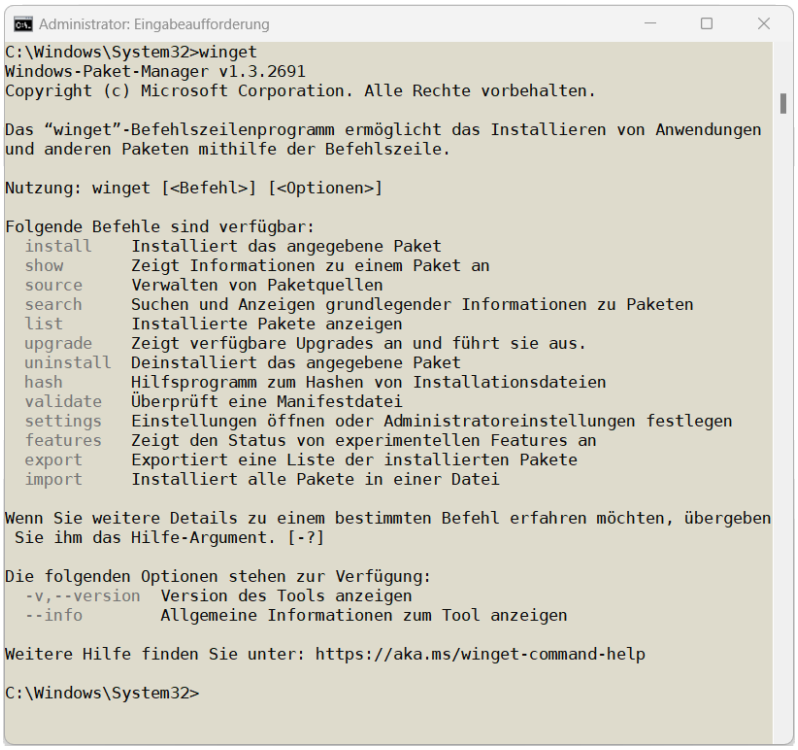


Abb. 134: Hilfstext des Paketmanagers WinGet

Falls Sie Windows 10 einsetzen und anstelle des Hilfstextes nur eine Meldung wie „Befehl oder Dateiname nicht gefunden“ erhalten, sollten Sie sich zuerst darum kümmern, aktuelle Windows-Updates zu installieren.

Wenn Sie den Namen des Programms, das Sie installieren möchten, bereits kennen, installieren Sie es einfach durch Eingabe des folgenden Befehls:

```
winget install Programmname
```

Falls Sie den Namen nur ungefähr wissen, finden Sie durch die Eingabe von

```
winget search Suchbegriff
```

hoffentlich schnell die korrekte Schreibweise heraus.

Außer der WinGet-eigenen Liste werden noch Programme der Paketquelle „msstore“ aufgeführt. Diese enthält zu einem großen Teil kostenpflichtige Programme, Abonnement-Produkte oder eingeschränkte Demoversionen. Mit dem WinGet-Aufruf

```
winget source remove msstore
```

entfernen Sie diese unerwünschten Einträge aus der von WinGet verwalteten Liste.

Wenn Sie anstatt in einer textbasierten Liste lieber in einem grafischen Katalog stöbern, finden Sie auf Webseiten wie winget.run oder winstall.app eine sortierte Aufstellung aller für WinGet verfügbaren Programme mit kurzen Beschreibungen.

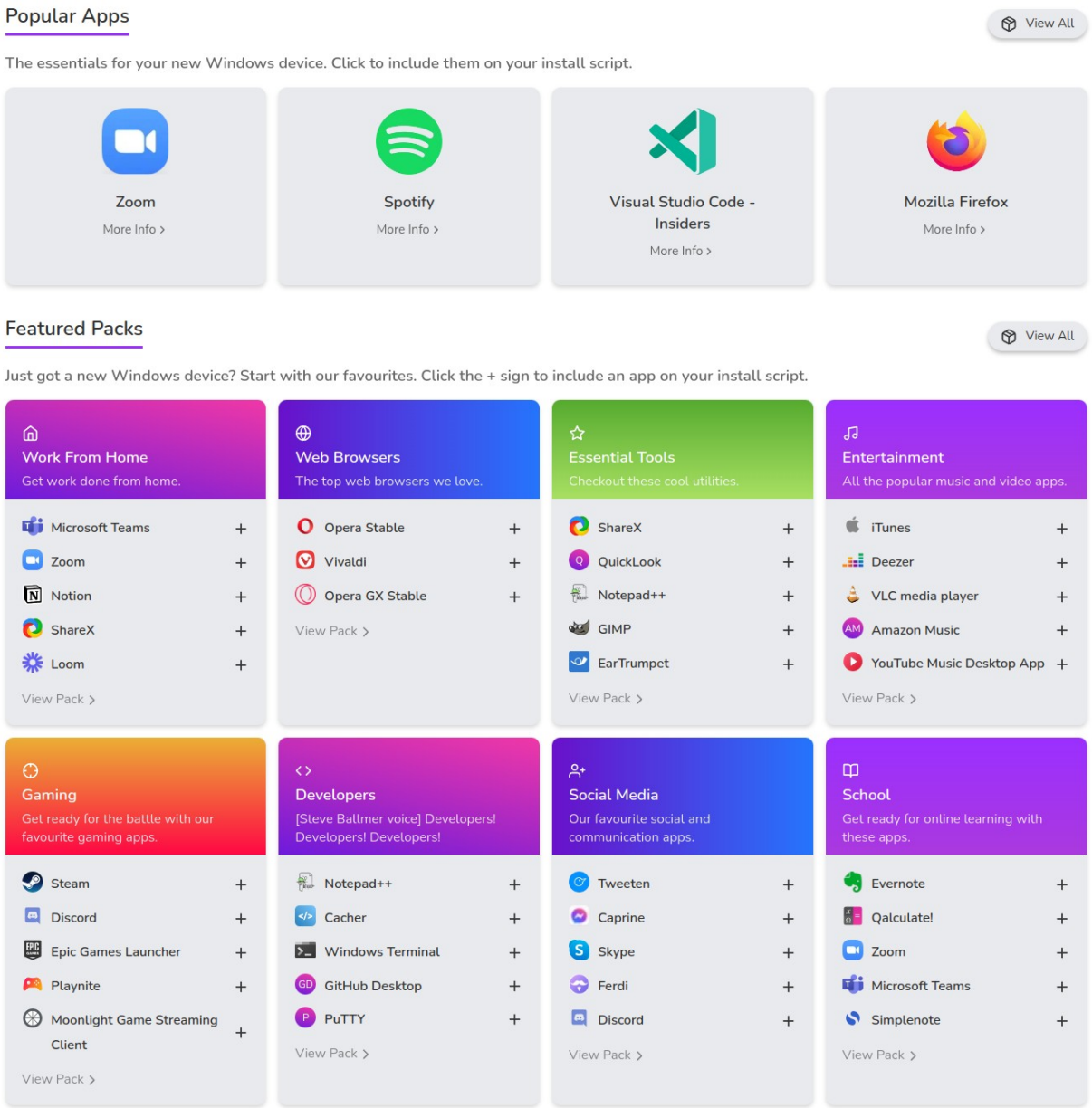


Abb. 135: Katalog der installierbaren Software auf winstall.app

Mit WinGet können Sie Ihre Programmsammlung stets aktuell halten. Der Befehl

```
winget upgrade --all
```

aktualisiert alle bereits auf Ihrem Rechner installierten Anwendungsprogramme, sofern sie WinGet bekannt sind. Die Erweiterung auf

```
winget upgrade --all --include-unknown
```

versucht zusätzlich, Programme zu aktualisieren, deren Versionsnummer aufgrund eines nachlässig konfigurierten Installationsprogramms bisher nicht bekannt ist.

Eine Übersicht über alle installierten Programme erhalten Sie mit

```
winget list
```

und überflüssige sowie unerwünschte Programme werden durch

```
winget uninstall Programmname
```

aus dem System entfernt. Achten Sie aber darauf, dass Sie nicht versehentlich wichtige Windows-Komponenten entfernen. Der mitgelieferte Werbemüll wie beispielsweise das Videoschnittprogramm Clipchamp sowie das alle möglichen persönliche Daten einschließlich sämtlicher ins Startmenü eingetippter Suchbegriffe an Microsoft übertragende Cortana sind vermutlich durchaus geeignete Kandidaten. Beim Deinstallieren von Programmen, deren Name Leerzeichen enthält, muss dieser in Anführungszeichen gesetzt werden (Abb. 136).

```
C:\Users\Qemu>winget uninstall "xbox tcui"
Gefunden Xbox TCUI [Microsoft.Xbox.TCUI_8wekyb3d8bbwe]
Paket-Deinstallation wird gestartet...
  ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒  100%
Erfolgreich deinstalliert

C:\Users\Qemu>winget uninstall "xbox console companion"
Gefunden Xbox Console Companion [Microsoft.XboxApp_8wekyb3d8bbwe]
Paket-Deinstallation wird gestartet...
  ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒  100%
Erfolgreich deinstalliert

C:\Users\Qemu>winget uninstall "xbox game bar"
Gefunden Xbox Game Bar [Microsoft.XboxGamingOverlay_8wekyb3d8bbwe]
Paket-Deinstallation wird gestartet...
  ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒  100%
Erfolgreich deinstalliert

C:\Users\Qemu>winget uninstall "xbox game bar plugin"
Gefunden Xbox Game Bar Plugin [Microsoft.XboxGameOverlay_8wekyb3d8bbwe]
Paket-Deinstallation wird gestartet...
  ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒  100%
Erfolgreich deinstalliert

C:\Users\Qemu>winget uninstall "xbox identity provider"
Gefunden Xbox Identity Provider [Microsoft.XboxIdentityProvider_8wekyb3d8bbwe]
Paket-Deinstallation wird gestartet...
  ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒  100%
Erfolgreich deinstalliert

C:\Users\Qemu>winget uninstall "xbox game speech window"
Gefunden Xbox Game Speech Window [Microsoft.XboxSpeechToTextOverlay_8wekyb3d8bbwe]
Paket-Deinstallation wird gestartet...
  ▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒  100%
Erfolgreich deinstalliert
```

Abb. 136: Deinstallation von Programm durch WinGet

Weitere Hilfe zu Winget erhalten Sie auf Microsofts Hilfeseiten[1]. Die dort an einigen Stellen verwendete seltsame Bezeichnung „Moniker“ bedeutet übrigens soviel wie „Spitzname“ oder „Ersatzname“.

1 https://aka.ms/winget-command-help

7.4 Abbildungsverzeichnis

Abb. 1: The Difference (Randall Munroe).....13
Abb. 2: Schreib-/Leseköpfe einer Festplatte im Größenvergleich.....18
Abb. 3: Tastenbezeichnungen unter Linux und Windows.....19
Abb. 4: WinCompose rüstet auch eine Unicode-Eingabe nach.....23
Abb. 5: Teil des Verzeichnisbaums unter Microsoft Windows.....27
Abb. 6: Ordneroptionen in Windows XP (2001) und Windows 11 (2022). 29
Abb. 7: Dateinamenerweiterungen bei macOS heißen Suffixe.....30
Abb. 8: Bibliotheken als „Dieser PC“ im Windows-10-Explorer.....31
Abb. 9: Eigene Dateien unter Windows 10.....33
Abb. 10: Anlegen eines ZIP-Archives im Windows-Explorer.....34
Abb. 11: ZIP-Datei mit Windows-Umlauten unter Linux.....35
Abb. 12: Icons für „Ausschneiden“, „Kopieren“ und „Einfügen“.....37
Abb. 13: Bildschirmkopiemenü in Windows (oben) und Linux (unten).....40
Abb. 14: Die Zeichentabelle von Windows 10.....43
Abb. 15: Gedit unter Ubuntu Linux.....45
Abb. 16: Zwei vermeidenswerte Programme.....46
Abb. 17: Formatvorlagen in LibreOffice.....47
Abb. 18: Zeichenformatierung in LibreOffice.....49
Abb. 19: Navigation durch die Dokumentenstruktur im Browser.....51
Abb. 20: Tabellenkalkulation 1979 (Bild: Wikipedia).....53
Abb. 21: Formel einer Tabellenkalkulation.....54
Abb. 22: Funktion mit zwei Parametern.....56
Abb. 23: Bereichschreibweise.....57
Abb. 24: Fallunterscheidung.....58
Abb. 25: Tabelle mit Fallunterscheidungen.....59
Abb. 26: Drei verschachtelte Fallunterscheidungen.....60
Abb. 27: VERWEIS.....61
Abb. 28: SVERWEIS.....62
Abb. 29: Zielwertsuche.....63
Abb. 30: Solver.....64
Abb. 31: Matrixformeln.....65
Abb. 32: Gelöstes Gleichungssystem.....66
Abb. 33: x-y-Diagramm.....67
Abb. 34: CSV-Import-Dialog in LibreOffice Calc.....69
Abb. 35: Meme „Incel vs. Excel“.....70
Abb. 36: Nerdwitz. Foto: Markus Tacker, Lizenz: CC BY-ND 2.0.....73
Abb. 37: HTML-Struktur.....74
Abb. 38: Ein Browser stellt HTML-Seiten dar.....75
Abb. 39: Bestandteile eines HTML-Elements.....76

Abb. 40: Ein einfaches Flussdiagramm....80
Abb. 41: Struktogramm: Sequenz von Arbeitsschritten....82
Abb. 42: Struktogramm: Fallunterscheidung....82
Abb. 43: Struktogramm: Mehrfachauswahl....83
Abb. 44: Struktogramm: Schleife....84
Abb. 45: Struktogramm: Nichtabweisende Schleife....84
Abb. 46: Struktogramm: Endlosschleife....85
Abb. 47: Struktogramm: Endlosschleife mit Aussprung....85
Abb. 48: Struktogrammbeispiel „Zahlenraten“....86
Abb. 49: Der Struktogramm-Editor „Structorizer“....87
Abb. 50: Guido van Rossum 2006 (dsearls, CC-BY-SA 2.0)....89
Abb. 51: Wählen Sie „Customize installation“....91
Abb. 52: Setzen Sie ruhig alle Häkchen....92
Abb. 53: Installation für alle Benutzerinnen und Benutzer....92
Abb. 54: Das Startmenü von Windows 11....93
Abb. 55: Paketverwaltung Synaptic in Ubuntu Linux....95
Abb. 56: Paketinstallation mit PIP unter Windows 10....96
Abb. 57: Paketinstallation ohne Administratorrechte....97
Abb. 58: Die Python-Shell der IDLE unter Windows....98
Abb. 59: Die IDLE-Shell unter Linux....98
Abb. 60: Die IDLE-Shell als Taschenrechner....99
Abb. 61: Python-Fehlermeldungen....101
Abb. 62: Variablenmodell „beschriftete Kästchen“....104
Abb. 63: Funktion mit Eingangswerten und Rückgabewert....111
Abb. 64: Funktion mit Wirkung....111
Abb. 65: Funktionen mit und ohne Wirkung oder Rückgabewert....112
Abb. 66: Modulliste nach Drücken der Tabulatortaste....115
Abb. 67: Fallunterscheidung im Struktogramm....137
Abb. 68: if … elif … else im Struktogramm....139
Abb. 69: Bedingte Schleife im Struktogramm....142
Abb. 70: Nicht abweisende Schleife im Struktogramm....144
Abb. 71: Else-Zweig einer For-Schleife....152
Abb. 72: Verschachtelte Schleifen im Struktogramm....153
Abb. 73: Merkhilfe für Sequenzabschnitte....161
Abb. 74: www.pythontutor.com....164
Abb. 75: deepcopy....165
Abb. 76: Vorbild für ein Objekt: Ein Einfeldträger....185
Abb. 77: Das Arithmetikmodul „labermath“....195
Abb. 78: Anzeige der Modulverzeichnisse unter Windows XP....197
Abb. 79: ASCII-Zeichen....220
Abb. 80: Schreibrechte unter Windows....233

Abb. 81: Eines der einfachsten Matplotlib-Diagramme.....236
Abb. 82: Verbessertes Matplotlib-Diagramm.....239
Abb. 83: Plot mit Markern.....243
Abb. 84: Scatterplot mit Flächen- und Farblisten.....244
Abb. 85: Textausrichtung mit Matplotlib.....245
Abb. 86: Flächenfüllung mit plt.fill().....246
Abb. 87: Unbeeinflusste Anzeigereihenfolge.....247
Abb. 88: Einfluss von zorder.....248
Abb. 89: x-y-z-Oberfläche mit Terrain-Farbgebung.....249
Abb. 90: GUI-Programm aus dem ersten Semester 2014/2015.....252
Abb. 91: Das Tk-Hauptfenster.....253
Abb. 92: Tk-Fenster mit festgelegter Größe und Überschrift.....254
Abb. 93: Hauptfenster und Unterfenster.....255
Abb. 94: Fensterdekorationen.....256
Abb. 95: Die leere Leinwand.....257
Abb. 96: Das tk-Koordinatensystem.....258
Abb. 97: Koordinatentransformation.....260
Abb. 98: Linienzug mit Breite und Farbe.....262
Abb. 99: Pfeilspitzen am Anfang und/oder am Ende von Linien.....263
Abb. 100: Gestrichelte Linien.....263
Abb. 101: Linienzug und Spline.....264
Abb. 102: Dreieck als geschlossenes Polygon.....265
Abb. 103: Rechteck und Ellipse.....266
Abb. 104: Die Ankerpunkte eines Canvas-Textes.....268
Abb. 105: Wo ist die Maus?.....270
Abb. 106: Glade.....276
Abb. 107: Pack.....278
Abb. 108: Grid.....279
Abb. 109: Button.....282
Abb. 110: Die Ankerpunkte eines Label-Textes.....283
Abb. 111: Text- und Image-Label.....283
Abb. 112: Entry.....284
Abb. 113: Scale.....286
Abb. 114: Horizontales Scale-Widget.....288
Abb. 115: Frame.....289
Abb. 116: Anordnung der LabelFrame-Beschriftung.....290
Abb. 117: LabelFrame.....291
Abb. 118: PanedWindow.....294
Abb. 119: Checkbuttons.....296
Abb. 120: Radiobuttons.....299
Abb. 121: ttk-Menubutton.....301

Abb. 122: Webserver unter Windows 10.....304
Abb. 123: Wahrheitstabelle and.....309
Abb. 124: Wahrheitstabelle or.....309
Abb. 125: Wahrheitstabelle ^.....310
Abb. 126: Venn-Diagramm mit zwei Aussagen A und B.....312
Abb. 127: Venn-Diagramme und logische Aussagen.....313
Abb. 128: QR-Code.....315
Abb. 129: ASCII-Zeichen als Bits und Bytes.....316
Abb. 130: ASCII-Code.....319
Abb. 131: Die 256 Zeichen im IBM-PC8-Zeichencode.....320
Abb. 132: Windows-Umlaute.....321
Abb. 133: Windows ist fast schon Unicode-tauglich.....322
Abb. 134: Hilfstext des Paketmanagers WinGet.....332
Abb. 135: Katalog der installierbaren Software auf winstall.app.....334
Abb. 136: Deinstallation von Programm durch WinGet.....336

7.5 Links und Literaturhinweise

Wenn im Text Bezug auf andere Werke genommen wird, finden Sie die entsprechenden Angaben direkt im Text oder als Fußnote. Anstelle einer wissenschaftlichen Literaturliste möchte ich Ihnen in diesem Kapitel daher lieber ein paar Tipps zum Weiterlesen zur Verfügung stellen:

- „Python – Der Grundkurs“ ist ein Buch von Michael Kofler, das viele Übungen und Codebeispiele enthält und besonders von Studierenden empfohlen wird.

  https://kofler.info/buecher/python/
  2. Auflage 2021, ISBN 978-3-8362-8513-1

- Die Universität Waterloo in Ontario, Kanada, bietet einen hervorragend gemachten Onlinekurs in Deutscher Sprache an, der in Zusammenarbeit mit dem Bundeswettbewerb Informatik entstand. Besonders gelungen ist die Einbindung von interaktiven Elementen, mit denen sich Codebeispiele direkt auf der Webseite ausprobieren und überprüfen lassen. Lehrende können sich dort als „Guru“ eintragen und von ihren Schülern oder Studierenden bei Problemen angeschrieben werden. Mein Guru-Name dort ist übrigens „MV“.

  https://cscircles.cemc.uwaterloo.ca/using-website-de/

- Wer im Englischunterricht nicht nur Kreide holen war, kann die wichtigsten Python-Konzepte in einem Online-Tutorial mit 92 kurzen Lektionen in neun Modulen kennenlernen und vertiefen. Nach Abschluss jedes Moduls kann man seinen Lernstand in einem kleinen Quiz überprüfen und zum Schluss winkt ein Teilnahmezertifikat.

  https://www.sololearn.com/learn/courses/python-introduction

- „Die Website „Pythonbuch“ von Marco Schmid und Beni Keller richtet sich an Schüler der Oberstufe und eignet sich hervorragend für alle, die ohne lange Umschweife schnell ans Programmieren kommen wollen. Mir gefällt besonders, dass sie die wichtigsten Elemente eines Programms zuerst behandeln: Die Quelltextkommentare.

https://pythonbuch.com

- „Das Python-3.3-Tutorial“ in der deutschen Übersetzung ist eine Fundgrube für Informatikfans, die sich etwas intensiver mit dem Stoff befassen wollen, als es der Rahmen dieses Skriptes erlaubt. Auch zum Nacharbeiten und Vertiefen der Vorlesungen ist der Bereich ab Kapitel 3 ein geeignetes Hilfsmittel.

  https://readthedocs.org/projects/py-tutorial-de/

- Bernd Klein hat nicht nur das Buch „Einführung in Python 3 – In einer Woche programmieren lernen“ geschrieben, sondern ist auch Verfasser eines Online-Kurses.

  http://www.python-kurs.eu/python3_kurs.php

- Das erste Semester ist viel zu kurz, um alles über Python zu lernen, was man in Wissenschaft und Ingenieurwesen gebrauchen kann. Gert-Ludwig Ingold hat mit dem Online-Vorlesungsskript „Python für Naturwissenschaftler“ eine Übersicht der fortgeschrittenen Aspekte von Python geschaffen.

  https://gertingold.github.io/pythonnawi/index.html

- Das weite Feld der Grafikprogrammierung mit *tkinter* wurde vom 2017 verstorbenen John W. Shipman vom *New Mexico Tech Computer Center* sehr ausführlich dokumentiert. Die archivierte Website ist in englischer Sprache und kann als gut lesbar gesetztes PDF heruntergeladen werden.

  https://anzeljg.github.io/rin2/book2/2405/docs/tkinter

- Individuelle Hilfe zu allen möglichen Programmierfragen bieten die englischsprachigen Fragen-und-Antworten-Seiten der Community auf *Stackoverflow*. Hier gehört es zum guten Ton, Fragen gleich mit einem Stück Programmtext zu beantworten.

  http://stackoverflow.com/search?q=python3

- Dass ein großes Sprachmodell beim Lernen einer neuen Sprache auch dann helfen kann, wenn es sich um eine Programmiersprache handelt, zeigt die Firma OpenAI mit ihrem vortrainierten Textgenerator ChatGPT. Wer in der Lage ist, Algorithmen klar zu formulieren und Fragen zielgerichtet und problembezogen zu stellen, erhält von dem Sprachmodell teilweise beeindruckende Antworten und mitunter sogar direkt lauffähige Programme.

  https://chat.openai.com/

- Zu guter Letzt sei auf die offizielle Dokumentation des Python-Projektes hingewiesen. Hier sind auch die als „PEP 8“ bekannt gewordenen Gestaltungsvorschläge festgehalten, die dafür sorgen, dass unsere Programme nicht nur vom Python-Interpreter, sondern auch von Menschen gut gelesen werden können.

  https://docs.python.org/3/tutorial/index.html

  https://www.python.org/dev/peps/pep-0008.

7.6 Lizenz

Der Inhalt dieses Werkes ist urheberrechtlich geschützt und steht unter einer Creative-Commons-Lizenz. Das heißt, dass ich zu Recht ziemlich sauer werden darf, wenn ich Inhalte aus diesem Buch irgendwo wiederfinde, wo sie als eigenes Werk der Kopistin oder des Kopisten ausgegeben werden.

Sie dürfen den Text und die Grafiken für Ihre eigenen Werke verwenden, auch verändern und weitergeben, solange Sie sich an die Creative-Commons-Lizenzbedingungen halten. Die beiden wesentlichen Punkte dieser Bedingungen lauten: Ihr eigenes Werk muss auch wieder unter einer Creative-Commons-Lizenz stehen und Sie müssen stets den Urheber angeben.

Eine kommerzielle Nutzung dieses Textes ist untersagt. Sie dürfen ihn also auch in veränderter Form nicht verkaufen oder auf gewerblich betriebene Plattformen wie Docplayer, Scribd oder Yumpu hochladen.

Weitere Informationen dazu finden Sie auf der Webseite

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.de.

Alle Bildinhalte in diesem Lehrbuch, die keine eigenen Werke des Autors sind, stehen selbst ebenfalls unter einer Creative-Commons-Lizenz (die jeweilige Lizenz ist am Bild angegeben) oder sind gemeinfrei (public domain).

Das Python-Logo ist ein eingetragenes Warenzeichen der Python Software Foundation.

Das Titelfoto zeigt die Pythonbrücke (Pythonbrug) in Amsterdam. Es wurde am 10. Mai 2008 von Alain Rouiller aufgenommen. Er gab ihm den Titel „Java Eiland 51“. Original: http://klixan.de/?dce

7.7 Download und Feedback

Dieses Vorlesungsskript wird bei Bedarf aktualisiert.

Die jeweils aktuellste Version stelle ich als PDF-Datei über den Link

https://martinvogel.de/python

zum Herunterladen bereit.

Ihr Pythonbuch-Exemplar wurde am 23. November 2023 veröffentlicht.

Über Anregungen und Kommentare freue ich mich immer sehr. Sie sind herzlich eingeladen, dazu den Kommentarbereich in meinem Blog zu verwenden:

https://martinvogel.de/blog/index.php?/archives/120-Kommentare-zum-Python-3-Buch.html

Wer nicht öffentlich schreiben möchte, darf gerne eine Mail schicken:

martin.vogel@hs-bochum.de