



Ralph Steyer

Programmierung in Python

Ein kompakter Einstieg für die Praxis

2. Auflage



CODE
INSIDE



Springer Vieweg

Programmierung in Python

Ralph Steyer

Programmierung in Python

Ein kompakter Einstieg für die Praxis

2. Auflage



Springer Vieweg

Ralph Steyer
Bodenheim, Deutschland

ISBN 978-3-658-44285-9 ISBN 978-3-658-44286-6 (eBook)
<https://doi.org/10.1007/978-3-658-44286-6>

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <https://portal.dnb.de> abrufbar.

© Der/die Herausgeber bzw. der/die Autor(en), exklusiv lizenziert an Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2018, 2024

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Planung/Lektorat: David Imgrund
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Wenn Sie dieses Produkt entsorgen, geben Sie das Papier bitte zum Recycling.

Vorwort

„And now for something completely different.“ Was ist der Sinn des Lebens? Wie geht es Brian? Was machen Ritter mit den Kokosnüssen? Ist die Schwerkraft wirklich so wunderbar? Was macht man mit diesem lebensgefährlichen Witz? Und was haben diese Fragen mit dem Buch zu tun? Fragen Sie für die Antworten einfach einmal den Erfinder von Python – Guido van Rossum. Denn der war bei der Entwicklung der Sprache ein großer Fan der englischen Komikertruppe Monty Python und deren Show Monty Python's Flying Circus. Und nun fragen Sie sich immer noch, woher der Name „Python“ stammt? Ich überlasse die Antwort Ihrer Phantasie oder intensiven Recherchen im Internet. Oder Ihrer Aufmerksamkeit beim Lesen dieses Buchs, denn ich nehme mir die Freiheit und lasse immer wieder Zitate von Monty Python einfließen – denn auch ich bin Fan der Truppe.

Die Programmiersprache Python hat sich in den vergangenen Jahren zu einem Schwergewicht in der Programmierszene entwickelt. Offensichtlich trifft das Konzept von Python den Nerv der Zeit. Oder genauer gesagt: Das Konzept stellt Ansätze, Lösungen und Vorgehensweisen für Probleme bereit, die andere Sprachen so nicht bieten und viele Leute interessant finden. Insbesondere auch die KI (künstliche Intelligenz) oder engl. AI („artificial intelligence“) entwickelt sich seit einiger Zeit als wesentliches Einsatzgebiet für Python.

Python gilt aktuell als eine der beliebtesten Einsteigersprachen überhaupt, denn Python wurde mit dem Ziel größter Einfachheit und Übersichtlichkeit entworfen. Zentrales Ziel bei der Entwicklung der Sprache war die Förderung eines gut lesbaren, knappen Programmierstils. In vielen Ländern hat Python an Universitäten bei Anfängerkursen in informatikbezogenen Studiengängen Java abgelöst, das über viele Jahre die Szene beherrschte und im professionellen Umfeld immer noch das Maß aller Dinge darstellt. Wobei man erwähnen muss, dass viele neue Python-Programmierer dann später zusätzlich Java oder eine andere OO-Sprache wie C# lernen oder auch darauf umsteigen. Aber auch für diesen Weg legt Python mit dem Zwang zu einem strukturierten, klaren Programmierstil eine hervorragende Grundlage – wenn man dessen Freiheiten nicht missbraucht.

Umgekehrt lässt sich Python sehr schnell erfassen, wenn man bereits Erfahrung mit anderen, weitgehend beliebigen Programmiersprachen hat. Denn Python unterstützt sowohl die objektorientierte, die aspektorientierte, die strukturierte als auch die funktionale Programmierung. Das bedeutet, Python zwingt den Programmierer nicht zu einem einzigen Programmierstil, sondern erlaubt, das für die jeweilige Aufgabe am besten geeignete Paradigma zu wählen. Und damit können ebenso Erfahrungen aus anderen Programmierkonzepten mehr oder weniger direkt weitergenutzt werden. Dieser universell mögliche Zugang ist neben der Einfachheit vermutlich eines der Erfolgsrezepte von Python.

Nun noch kurz zu meiner Person. Ich habe vor vielen Jahren in Frankfurt/Main an der Goethe-Universität Mathematik studiert (Diplom) und danach anfangs einen recht typischen Werdegang für Mathematiker genommen: Ich bin bei einer großen Versicherung gelandet – aber schon da mit IT-Schwerpunkt. Zuerst habe ich einige Jahre als Programmierer mit Turbo Pascal und später mit C und C++ gearbeitet. Nach vier Jahren habe ich in die fachliche Konzeption für eine Großrechnerdatenbank unter MVS gewechselt. Die Erfahrung war für meinen Schritt in die Selbstständigkeit sehr motivationsfördernd, denn mir wurde klar, dass ich das nicht auf Dauer machen wollte. Seit 1996 verdiene ich daher meinen Lebensunterhalt als Freelancer, wobei ich fliegend zwischen der Arbeit als Fachautor, Fachjournalist, EDV-Dozent, Consultant und Programmierer wechsele. Daneben referiere ich gelegentlich auf Web-Kongressen, unterrichte an verschiedenen Akademien und Fachhochschulen oder nehme Videotrainings auf. Das macht aus meiner Sicht einen guten Mix aus, bewahrt vor beruflicher Langeweile und hält mich in der Praxis wie auch am Puls der Entwicklung. Insbesondere habe ich das Vergnügen wie auch die Last, mich permanent über neue Entwicklungen auf dem Laufenden zu halten, denn die Halbwertszeit von Computerwissen ist ziemlich kurz. Dementsprechend ist mein Job zwar anstrengend, aber vor allem immer wieder spannend. Wenn Sie weitere und detaillierte Informationen zu meiner Person erhalten wollen, finden Sie mich natürlich im Internet. Ich bin in diversen sozialen Netzwerken und natürlich auch mit einer eigenen Webseite im weltweiten Web unterwegs. Etwa hier:

<http://www.rjs.de>

<http://blog.rjs.de>

Doch lassen Sie uns also jetzt mit Python beginnen.

Herbst und Winter
2023 und 2024

Ralph Steyer

Inhaltsverzeichnis

1 Einleitung und Grundlagen – Bevor es richtig losgeht	1
1.1 Was behandeln wir in dem einleitenden Kapitel?	1
1.2 Das Ziel des Buchs.	1
1.3 Was sollten Sie bereits können?	2
1.4 Was ist Python?	2
1.4.1 Das Ziel von Python.	3
1.4.2 Was umfasst Python?	3
1.4.3 Die verschiedenen Python-Paradigma	4
1.5 Was benötigen Sie zum Arbeiten mit dem Buch?	4
1.5.1 Hardware und Betriebssystem	4
1.5.2 Die Python-Version	5
1.5.3 Python laden und installieren	5
2 Erste Beispiele – Der Sprung ins kalte Wasser	21
2.1 Was behandeln wir in diesem Kapitel?	21
2.2 Der Interaktivmodus – die Kommandozeile von Python	21
2.2.1 Das Prompt	23
2.2.2 Der Hilfemodus in der Kommandozeile	26
2.3 Anweisungen in (echten) Quelltext auslagern	29
2.4 Von IDLE & Co bis zu Python in der Cloud	31
2.4.1 Weitere IDEs und Editoren für Python	32
2.4.2 Cloud und RIA	33
3 Built-in Functions – Modularisierung durch Unterprogramme	41
3.1 Was behandeln wir in diesem Kapitel?	41
3.2 Was sind Funktionen im Allgemeinen?	41
3.3 Built-in-Funktionen	42
3.3.1 Hilfe zu Built-in Functions im Hilfemodus	42
3.3.2 Hilfe zu Built-in Functions im Editormodus	43

3.3.3	Die print()-Funktion	44
3.3.4	Die input()-Funktion	50
3.3.5	Eine kurze Übersicht aller Built-in Functions	52
4	Grundlegende Begriffe – Kommentare, SheBang und Strukturanalysen	55
4.1	Was behandeln wir in diesem Kapitel?	55
4.2	Token und Parser	55
4.2.1	Zerlegen von Quelltext	56
4.3	Kommentare	58
4.3.1	Kommentare in Python	58
4.4	SheBang und eine Python-Datei direkt ausführen	60
4.4.1	SheBang als besonderer Kommentar	60
5	Anweisungen – Dem Computer Befehle geben	61
5.1	Was behandeln wir in diesem Kapitel?	61
5.2	Was sind Anweisungen?	61
5.2.1	Eine Frage der Reihenfolge	61
5.3	Anweisungsarten	62
5.3.1	Blockanweisung	62
5.3.2	Kontrollflussanweisungen	63
5.3.3	Deklarationsanweisung	64
5.3.4	Ausdrucksanweisung	64
5.3.5	Die leere Anweisung <i>pass</i>	65
6	Datentypen, Variablen und Literale – Die Art der Information	67
6.1	Was behandeln wir in diesem Kapitel?	67
6.2	Variablen	67
6.2.1	Variablen deklarieren	67
6.2.2	Variablen im Quellcode verwenden	69
6.3	Die Datentypen in Python	69
6.3.1	Lose Typisierung und Typumwandlung in Python	69
6.3.2	Die Python-Datentypen	71
6.3.3	Zahlen – <i>int</i> , <i>float</i> und <i>complex</i>	73
6.3.4	Zeichenketten -(<i>str</i>) und Zeichenliterale	78
6.4	Den Datentyp bestimmen und umwandeln	79
6.4.1	Den Datentyp mit <i>type()</i> dynamisch bestimmen	79
6.4.2	Implizite und explizite Typumwandlung	80
7	Ausdrücke, Operatoren und Operanden – Die Verarbeitung von Daten	83
7.1	Was behandeln wir in diesem Kapitel?	83
7.2	Ausdrücke	83
7.3	Operationen mit Operatoren und Operanden	84
7.3.1	Arithmetische Operatoren	84
7.3.2	Der String-Verkettungsoperator	87

7.3.3	Zuweisungsoperatoren	88
7.3.4	Boolesche Operatoren (Vergleichsoperatoren)	89
7.3.5	Logische Operatoren	90
7.3.6	Die Membership-Operatoren	92
7.3.7	Identitätsoperatoren	92
7.3.8	Bitweise Operatoren.	93
7.4	Python-Sonderfälle bei der Auswertung	98
7.4.1	Verkettete Auswertung	98
7.4.2	Der Walross-Operator	99
7.5	Operatorvorrang und Ausdrucksbewertung	101
7.5.1	Die Priorität der Python-Operatoren	101
7.5.2	Bewertung von Ausdrücken	101
8	Kontrollstrukturen – Die Steuerung des Programmflusses	103
8.1	Was behandeln wir in diesem Kapitel?	103
8.2	Was sind Kontrollstrukturen?	103
8.3	Die Kontrollstrukturen in Python.	104
8.3.1	Entscheidungsanweisungen	104
8.3.2	Iterationsanweisungen	113
8.3.3	Sprunganweisungen	116
9	Funktionen in Python – Modularisierung mit „Unterprogrammen“	119
9.1	Was behandeln wir in diesem Kapitel?	119
9.2	In Python eigene Funktionen deklarieren – Das Schlüsselwort def	119
9.2.1	Übergabewerte	120
9.2.2	Rückgabewerte.	121
9.3	Funktionen aufrufen.	121
9.3.1	Stehen in Python global deklarierte Variablen in der Funktion zur Verfügung?	123
9.4	Rekursion	124
9.5	Innere Funktionen – Closures	128
9.6	Lambda-Ausdrücke und anonyme Funktionen	129
9.6.1	Lambda-Funktionen verwenden	129
9.7	Besondere Situationen bei Funktionen in Python	130
9.7.1	Lokale Variablen in Funktionen	130
9.7.2	Die Anzahl der Parameter passt nicht	131
9.7.3	Unerreichbarer Code	134
10	Sequentielle Datenstrukturen – Mehrere Informationen gemeinsam verwalten	137
10.1	Was behandeln wir in diesem Kapitel?	137
10.2	Was sind sequentielle Datenstrukturen?	137

10.2.1	Zeichenketten als sequenzielle Ansammlung von Zeichenliteralen	138
10.2.2	Arrays	138
10.3	Tupel	139
10.3.1	Verschachtelte Tupel	139
10.3.2	Tupel und der Membership-Operator	140
10.3.3	Einzelne Einträge in Tupel ansprechen	143
10.3.4	Die Anzahl der Elemente in einem Tupel bestimmen	146
10.4	Dynamische Listen	146
10.4.1	Warum Listen und Tupel?	147
10.5	Methoden für Listen	148
10.5.1	Verschiedene Listenmethoden in einem Beispiel	149
10.5.2	Einen Stack erzeugen	150
10.5.3	Eine Queue mit einer Liste erzeugen	151
10.6	Dictionaries	152
10.6.1	Spezielle Methoden für Dictionaries	153
10.6.2	Ein Beispiel zum allgemeinen Umgang mit Dictionaries	154
10.6.3	Ein Dictionary aktualisieren oder erweitern	155
10.6.4	Iteration über ein Dictionary	156
10.7	Mengen	157
10.7.1	Vereinfachte Notation	157
10.7.2	Operationen auf set-Objekten	158
10.8	Operatoren bei sequenziellen Datentypen	160
10.8.1	Der Plusoperator	160
10.8.2	Multiplikationen mit sequenziellen Datentypen	160
10.8.3	Inhalt überprüfen	161
10.9	Über sequenzielle Strukturen iterieren	163
10.10	Pattern-Matching	164
10.11	Comprehensions	169
10.12	Anwendung des Walrus Operator	171
11	Objektorientierte Programmierung in Python – Klassen, Objekte, Eigenschaften und Methoden	173
11.1	Was behandeln wir in diesem Kapitel?	173
11.2	Hintergründe der OOP	173
11.2.1	Ziele der OOP – Wiederverwendbarkeit und bessere Softwarequalität	174
11.2.2	Kernkonzepte der Objektorientierung	175
11.3	Klassen	177
11.3.1	Klassen als Baupläne, Konstruktoren und Destruktoren	177
11.3.2	Der konkrete Klassenaufbau in Python	177
11.3.3	Die konkrete Instanziierung	178

11.4	Details zu Objekten	182
11.4.1	OO-Philosophie als Abstraktion	183
11.4.2	Instanzelemente versus Klassenelemente	183
11.4.3	Der Aufbau von Objekten in Python	184
11.4.4	Zugriff auf Objektbestandteile	184
11.4.5	Von Grund auf objektorientiert	189
11.5	Klassenmethoden und statische Methoden	190
11.5.1	Klassenmethoden	190
11.5.2	Statische Methoden	191
11.6	Eine Frage der Sichtbarkeit	193
11.6.1	Ein Beispiel für den Zugriff auf ein öffentliches Element	193
11.6.2	Ein Beispiel für den versuchten Zugriff auf ein privates Element von außen	194
11.6.3	Getter und Setter	195
11.7	Ein Objekt löschen	197
11.7.1	Ein Beispiel für das Redefinieren des Destruktors	198
11.8	Ein paar besondere OO-Techniken	198
11.8.1	Eine to-string-Funktionalität bereitstellen – <code>__str__</code>	198
11.8.2	Objekte dynamisch erweitern, das Dictionary <code>__dict__</code> und Slots	199
11.8.3	Dynamische Erzeugung von Klassen, Metaklassen und die Klasse <code>type</code>	201
11.8.4	Shallow Copy: flaches und tiefes Kopieren	202
11.8.5	Reference Counting	204
11.9	Vererbung	205
11.9.1	Grundlagentheorie zur Vererbung	205
11.9.2	Umsetzung von Vererbung in Python	206
11.9.3	Mehrfachvererbung in Python	207
11.9.4	Polymorphie über Überschreiben und Überladen	210
11.10	Was ist mit Schnittstellen und abstrakten Klassen in Python?	212
11.10.1	Abstrakte Superklassen	212
11.10.2	Was ist im Allgemeinen eine Schnittstelle?	213
11.11	Module und Pakete	213
11.11.1	Die <code>import</code> -Anweisung	214
11.11.2	Importieren mit <code>from</code>	215
11.11.3	Pakete	215
11.11.4	Das Python-API	217
11.11.5	Fremde Pakete nutzen	218
12	Exception-Handling – Ausnahmsweise	219
12.1	Was behandeln wir in diesem Kapitel?	219
12.2	Was sind Ausnahmen?	220

12.3	Warum ein Ausnahmekonzept?	220
12.4	Konkrete Ausnahmebehandlung in Python	221
12.4.1	Ein erstes Beispiel mit einfacher Ausnahmebehandlung	221
12.4.2	Mehrere Ausnahmeblöcke	223
12.4.3	Die finally-Anweisung	223
12.4.4	Praktische Beispiele	224
12.5	Standard Exceptions	226
12.5.1	Die Reihenfolge bei mehreren Ausnahmetypen	227
12.6	Der else-Block	228
12.7	Ausnahmeobjekte auswerten	229
12.8	Werfen von Ausnahmen mit raise	230
12.9	Eigene Ausnahmeklassen definieren	232
12.10	Die assert-Anweisung	232
13	String-Verarbeitung in Python – Programmierte Textverarbeitung	233
13.1	Was behandeln wir in diesem Kapitel?	233
13.2	Typische String-Verarbeitungstechniken	234
13.3	Das konkrete Vorgehen in Python	234
13.3.1	String-Konstanten und die Format Specification Mini-Language	234
13.3.2	String-Funktionen	235
13.3.3	String-Methoden	235
13.4	Umgang mit regulären Ausdrücken	236
13.4.1	Was sind allgemein reguläre Ausdrücke?	237
13.4.2	Wo setzt man reguläre Ausdrücke ein?	237
13.4.3	Details zu Pattern	238
13.4.4	Optionen für die Häufigkeit	241
13.4.5	Die Umsetzung von regulären Ausdrücken in Python – das Modul re	241
13.4.6	Die Match-Objekte	244
13.4.7	Ein paar Beispiele mit regulären Ausdrücken	246
14	Datei-, Datenträger- und Datenbankzugriffe – Dauerhafte Daten	249
14.1	Was behandeln wir in diesem Kapitel?	249
14.2	Datenströme für die Ein- und Ausgabe	249
14.2.1	Das Öffnen und Schließen einer Datei	250
14.2.2	Schreiben in eine Datei	251
14.2.3	Auslesen aus einer Datei	252
14.2.4	Die with-Anweisung nutzen	253
14.2.5	Lese- und Schreibvorgänge absichern	253
14.3	Allgemeine Datei- und Verzeichnisoperationen	256

14.4	Objekte serialisieren und deserialisieren	258
14.4.1	Mit dump() den Objektzustand persistent machen	259
14.4.2	Mit load() den Objektzustand reproduzieren	259
14.5	Datenbankzugriffe	260
14.5.1	Was ist SQLite?	260
14.5.2	Zugriff auf SQLite in Python	261
14.5.3	Ein konkretes Datenbankbeispiel	262
15	Umgang mit Datum und Zeit – Terminsachen	265
15.1	Was behandeln wir in diesem Kapitel?	265
15.2	Allgemeines zum Umgang mit Datum und Zeit	265
15.3	Die Python-Module	266
15.4	Typische Beispiele für Operationen mit Zeit und Datum	266
15.4.1	Das aktuelle Systemdatum des Computers auslesen	266
15.4.2	Ein beliebiges Datumsobjekt erstellen	267
16	Grafische Oberflächen (GUI) mit Python – tkinter & Co als GUI-Framework	271
16.1	Was behandeln wir in diesem Kapitel?	271
16.2	Hintergrundinformationen zu modernen grafischen Oberflächen	271
16.3	Konkrete GUI-Konzepte in Python und das Modul tkinter	272
16.3.1	Ein Fenster vom Typ TK als Basis jeder GUI-Applikation	272
16.3.2	Der übliche OO-Ansatz	272
16.3.3	Die Layout-Manager	273
16.3.4	Wichtige GUI-Elemente	280
16.4	Die Ereignisbehandlung	281
16.4.1	Die konkrete Ereignisbehandlung in Python	281
16.4.2	Lambda-Ausdrücke verwenden	283
16.5	Eine grafische Datenbankapplikation	284
16.6	PyQt als Alternative	288
	Stichwortverzeichnis	291



Einleitung und Grundlagen – Bevor es richtig losgeht

1

1.1 Was behandeln wir in dem einleitenden Kapitel?

Bevor es mit Python richtig losgeht, sollen in diesem einleitenden Kapitel einige Dinge geklärt werden, die Ihnen die folgende Arbeit mit diesem Buch und der Programmiersprache im Allgemeinen erleichtern werden. Insbesondere sorgen wir an der Stelle dafür, dass Ihnen Python überhaupt zur Verfügung steht.

1.2 Das Ziel des Buchs

Dieses Buch ist zum Lernen von Python gedacht, entweder in Form des Selbststudiums oder als Begleitmaterial in entsprechenden Kursen. Zuerst einmal lernen Sie die elementaren Grundlagen, um überhaupt Programme mit Python erstellen wie auch pflegen zu können. Danach werden erweiterte Techniken wie umfassendere Anwendung der Syntax, objektorientierte Programmierung mit Python, komplexere Anwendungen mit sequenziellen Datenstrukturen, Umgang mit Modulen, Ausnahmebehandlung, Dateizugriffe, Datenbankzugriffe und die Erstellung grafischer Oberflächen behandelt. Dabei wird über sämtliche Themen hinweg Wert auf die grundsätzliche Anwendung der verschiedenen Techniken und einfache Beispiele gelegt und nicht auf Vollständigkeit aller möglichen Anweisungen, Befehle oder Parameter. Ebenso wird immer wieder darauf hingewiesen werden, dass man gewisse Freiheiten, die Ihnen Python bietet, nicht unbedingt ausnutzen sollte.

- Wir erstellen im Laufe des Buchs immer wieder praktische Beispiele. Die Quellcodes des Buchs finden Sie nach Kapiteln und darin erstellten Projekten sortiert auf den Webseiten des Verlags bzw. in einem Repository auf GitHub. Die Namen der jeweilig aktuellen Dateien beziehungsweise Projekte werden als Hinweise oder direkt im Text vor den jeweiligen Beispielen angegeben und bei Bedarf wiederholt. Ich empfehle allerdings, dass Sie die Beispiele unbedingt allesamt von Hand selbst erstellen. Das ist für das Verständnis und das Lernen eindeutig besser als ein reines Kopieren oder nur Anschauen.

1.3 Was sollten Sie bereits können?

Der Kurs ist als Einsteigerkurs konzipiert, in dem die Sprache Python von Grund auf erarbeitet wird. Dabei wird der Tatsache Rechnung getragen, dass Python mittlerweile oft als erste Programmiersprache überhaupt gelernt wird. Um nicht die einfachsten Programmiergrundlagen zu ausführlich erläutern zu müssen, wird aber zumindest ein Verständnis der Idee von Programmierung vorausgesetzt, obwohl Sie noch nicht zwingend programmieren müssen. Ebenso sollten Sie mit einem Texteditor umgehen können. Kenntnisse der wichtigsten Befehle Ihres Betriebssystems (Windows, Linux oder macOS) zum Umgang mit Dateien und Verzeichnissen in der Konsole sind hilfreich. Umsteiger aus anderen Sprachen sind aber auch explizit als Zielgruppe des Buchs einkalkuliert, und entsprechende Hinweise auf andere Programmiersprachen werden immer wieder Bezüge herstellen – insbesondere zu Sprachen der Webprogrammierung (PHP, JavaScript, ...), Java und .NET-Sprachen.

1.4 Was ist Python?

Python ist eine universelle, höhere Programmiersprache, die üblicherweise interpretiert wird. Es gibt also in der Laufzeitumgebung von Python einen Interpreter samt notwendiger weiterer Ressourcen.

Hintergrundinformation

Nun ist der Begriff des Interpreters gefallen, und die Konzepte und Fachausdrücke zur Übersetzung von Quellcode sollen nicht einfach vorausgesetzt werden. Wenn Sie Python-Programme erstellen, schreiben Sie den sogenannten **Quellcode** oder **Quelltext**, der der englischen Sprache angelehnt ist. Aber zur Ausführung muss dieser übersetzt werden, und zwar in ein Format, das ein Computer versteht, und das ist **Maschinencode**. Dazu gibt es verschiedene Möglichkeiten.

- Einmal gibt es den **Interpreter**. Dieser sorgt für eine Übersetzung in Maschinencode während der Programmausführung. Interpreter übersetzen den Quellcode eines Programms zeilenweise.
- Die Alternative ist der **Compiler**, der den kompletten Quelltext vor der Laufzeit übersetzt.

- In den vergangenen Jahren hat sich auch eine **zweistufige Übersetzung** etabliert – die Übersetzung mit Zwischencode. Das wird in Java oder dem .NET-Framework gemacht. Denn ein Nachteil bei kompilierten Programmen ist, dass diese Programme maschinenabhängig sind und somit nicht auf jeder Computerplattform, zum Beispiel Linux und Windows, ausgeführt werden können. In Java oder bei der .NET-Plattform wird Quellcode nicht zu einem ausführbaren Programm, sondern in einen Zwischencode kompiliert (1. Schritt). Dieser Code ist für alle Plattformen gleich und kann mithilfe des entsprechenden plattformspezifischen Interpreters (2. Schritt) auf der jeweiligen Plattform ausgeführt werden.

1.4.1 Das Ziel von Python

Python ist den meisten gängigen Programmiersprachen verwandt,¹ wurde aber mit dem Ziel größter Einfachheit und Übersichtlichkeit entworfen. Zentrales Ziel bei der Entwicklung der Sprache ist die Förderung eines gut lesbaren, knappen Programmierstils. So wird beispielsweise der Code nicht durch geschweifte Klammern (wie in fast allen C-basierten Sprachen), sondern durch zwingende Einrückungen strukturiert.² Zudem ist die gesamte Syntax reduziert und auf Übersichtlichkeit optimiert.

Wegen dieser klaren und überschaubaren Syntax gilt Python als einfach zu erlernen, zumal die Sprache mit relativ wenig Schlüsselwörtern auskommt. Es wird immer wieder zu hören sein, dass sich Python-basierte Skripte deutlich knapper formulieren lassen als in anderen Sprachen.

1.4.2 Was umfasst Python?

Es gibt einmal die Sprache Python, die aus den üblichen Schlüsselworten, Operatoren, eingebauten Funktionalitäten etc. sowie einer eigenständigen Syntax besteht. Python besitzt zudem eine umfangreiche Standardbibliothek (API – Application Programming Interface, wörtlich „Anwendungsprogrammierschnittstelle“), und zahlreiche Pakete im Python Package Index, bei deren Entwicklung ebenfalls großer Wert auf Überschaubarkeit, aber auch eine leichte Erweiterbarkeit gelegt wurde.

Python-Programme lassen sich deshalb auch in anderen Sprachen als Module einbetten. Umgekehrt lassen sich mit Python Module³ und Plug-ins⁴ für andere Programme

¹ Die meisten aktuellen Programmiersprachen gehen von der Syntax her auf C zurück.

² Was allerdings viele erfahrene Programmierer mit C-Background eher verstört, denn die saubere Notation von geschweiften Klammern ist sehr übersichtlich.

³ Zusammenfassungen von Quellcodestrukturen. Modul steht allgemein für einen Teil eines größeren Systems.

⁴ Plug-ins sind allgemeine kleine Programme oder Programm-Pakete, mit denen sich Software nach den eigenen Bedürfnissen anpassen und erweitern lässt. Sie integrieren sich dazu in das System, das sie erweitern oder anpassen sollen.

schreiben, die die entsprechende Unterstützung bieten. Dies ist zum Beispiel bei Blender, Cinema 4D, GIMP, Maya, OpenOffice beziehungsweise LibreOffice, PyMOL, SPSS, QGIS oder KiCad der Fall.

- ▶ Ganz wichtig ist, dass Python über Pakete (packages) organisiert wird und dazu ein integriertes Paketverwaltungssystem mitbringt. Genaugenommen ist das **pip** genannte Tool (früher pyinstall) das de facto und empfohlene Paketverwaltungsprogramm für Python-Pakete aus dem Python Package Index (PyPI), aus dem diverse Erweiterungen von Python bezogen werden können. PyPI ist der zentrale Paketpool und umfasst mittlerweile weit über 100.000 Pakete. Entwickler können nach einer Registrierung dort Module hochladen und so anderen Benutzern zur Verfügung stellen.

1.4.3 Die verschiedenen Python-Paradigma

Unter der Oberfläche ist Python streng objektorientiert. Aber die Art, wie man mit Python umgeht bzw. programmiert, ist flexibel. Python unterstützt sowohl die objektorientierte, die aspektorientierte, die strukturierte als auch die funktionale Programmierung. Das bedeutet, Python zwingt den Programmierer nicht zu einem einzigen Programmierstil, sondern erlaubt, das für die jeweilige Aufgabe am besten geeignete Paradigma zu wählen. Objektorientierte und strukturierte Programmierung werden vollständig, funktionale und aspektorientierte Programmierung werden zumindest durch einzelne Elemente der Sprache unterstützt. Ein zentrales Feature ist in Python die dynamische Typisierung samt dynamischer Speicherbereinigung. Damit kann man Python auch als reine Skriptsprache nutzen.

1.5 Was benötigen Sie zum Arbeiten mit dem Buch?

Python steht auf verschiedenen Plattformen und in verschiedenen Versionen zur Verfügung. Wir schauen uns erst einmal an, mit welchen Voraussetzungen Sie an das Buch gehen können.

1.5.1 Hardware und Betriebssystem

Python gibt es für Windows und macOS sowie Linux. Damit sind die drei Betriebssysteme genannt, mit denen wohl die meisten Leser arbeiten werden. Aber es gibt Python auch für viele weitere, teils ziemlich alte beziehungsweise exotische Betriebssysteme (unter anderem AS/400 [OS/400], BeOS, MorphOS, MS-DOS, OS/2, RISC OS, Series 60, Solaris oder HP-UX), wobei Sie beachten sollten, dass für ältere

Betriebssysteme die neuesten Python-Versionen oft nicht mehr zur Verfügung stehen. Ab der Version Python 3.5 wird etwa Windows XP nicht mehr unterstützt. Aber auch viele vorinstallierte Versionen von Python – wenn es denn solche gibt – sind nicht auf dem aktuellen Stand.

Letztendlich sind aber die Voraussetzungen für die Arbeit mit Python ziemlich niedrig. Sie benötigen für die Arbeit mit dem Buch und Python im Grunde nur einen Computer mit einem halbwegs modernen Betriebssystem. Als Basis für die Unterlagen wird explizit ein PC vorausgesetzt.

1.5.2 Die Python-Version

Die **Referenzbetriebssysteme** in diesem Buch werden Windows 10 und 11 sowie eine aktuelle Linux-Distribution sein.⁵ Aber auch macOS wird beachtet. Überwiegend wird in den Unterlagen mit Windows 10/11 gearbeitet, aber die Ausführungen und Beispiele sind auf andere Systeme übertragbar oder nicht von der Plattform abhängig. Sollten Spezifika interessant sein, wird das hervorgehoben.

1.5.2.1 Die Evolution von Python

- Python wurde Anfang der 1990er-Jahre von Guido van Rossum am Centrum Wiskunde & Informatica in Amsterdam als Nachfolger für die Sprache ABC entwickelt.
- Ursprüngliche Zielplattform war das verteilte Betriebssystem Amoeba.
- Die erste Vollversion erschien im Januar 1994.
- Python 2.0 erschien Oktober 2000.
- Python 3.0 (auch Python 3000) erschien im Dezember 2008, und die Serie 3 ist 2024 immer noch aktuell.

1.5.3 Python laden und installieren

Zum Erstellen von Python-Programmen braucht man im Grunde nur einen Klartexteditor. Aber um Python-Programme oder -Skripte auszuführen, braucht man mehr – eine Umgebung für Python. Deshalb muss für die weiteren Schritte mit dem Buch Python auf Ihrem Rechner installiert sein, oder Sie sollten das zu Beginn machen. Dabei müssen Sie beachten, dass es aktuell zwei „Schienen“ bei den Versionen von Python gibt. Zwar ist Python 3.0 wie erwähnt bereits im Dezember 2008 erschienen. Da Python 3.0 jedoch teilweise inkompatibel zu früheren Versionen ist, wurde Python 2.7 lange Zeit parallel zu Python 3.x weiter durch Updates unterstützt, wobei die Weiterentwicklung der 2.x-Schiene

⁵ Konkret wird an einigen Stellen Mint Linux 21 verwendet.

mittlerweile ausgelaufen ist. Dennoch ist diese Version nicht unwichtig, denn Python in der Version 2.x wird immer noch als Standard bei verschiedenen Betriebssystemen bereitgestellt.

- ▶ Die Besonderheiten der 2er-Versionen von Python werden im Buch nicht mehr berücksichtigt.

Die **Referenzversion** von Python für dieses Buch ist die Version 3.12.x, wobei unsere Ausführungen im Grunde für alle Versionen 3.x gelten. Die Unterschiede sind für dieses Buch in der Regel nicht wirklich relevant, außer für einige neuere Features in Python, die jeweils mit spezifischen Versionen von 3.x erst verfügbar sind. Bei den betroffenen Features wird im Buch explizit angegeben, ab welcher Version diese genutzt werden können. Sie sollten auch beachten, dass es für unsere Referenzbetriebssysteme kleinere Abweichungen in der verfügbaren Version von Python geben kann. Oder genauer: Es kann sich zeitlich etwas unterscheiden, wann etwa eine Version für Windows, Linux oder macOS veröffentlicht wird.

1.5.3.1 Das zentrale Webportal und die Python Software Foundation (PSF)

Wenn Sie erstmals Kontakt zu Python aufnehmen wollen, gehen Sie am besten zuerst auf die Webseite <https://www.python.org/> (Abb. 1.1). Das ist das zentrale Webportal von Python. Hier finden Sie die wichtigsten Informationen zu Python und auch alle notwendigen Ressourcen.

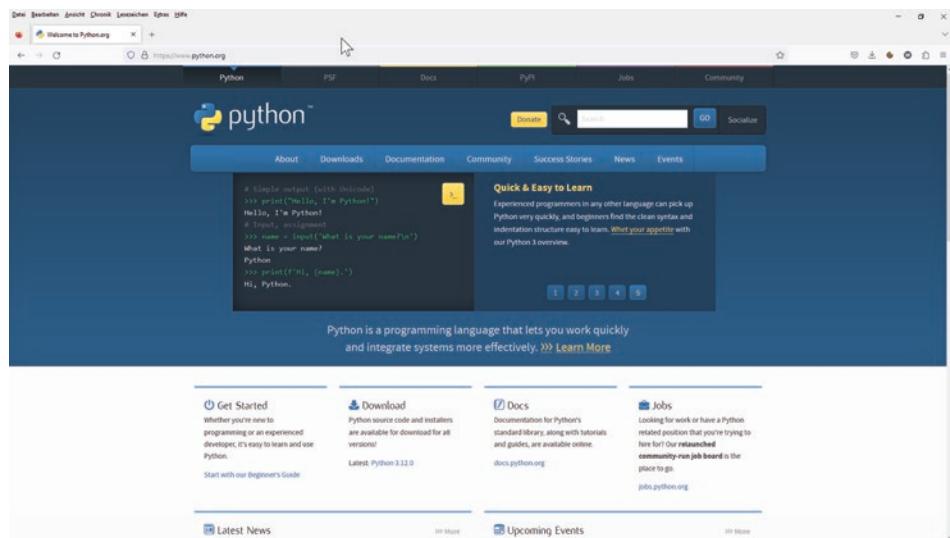


Abb. 1.1 Das „offizielle“ Webportal zu Python



Abb. 1.2 Die Webseite der PSF

Von da gelangen Sie über einen Link oben in der Navigation oder direkt über den Link <https://www.python.org/psf/> zur Python Software Foundation (PSF, Abb. 1.2).

PSF ist eine Non-Profit-Organisation, die hinter dem Open-Source-Projekt der Programmiersprache steht. Mitglieder der PSF sind sowohl relevante Einzelpersonen aus dem Python-Umfeld als auch Firmen wie Google, Microsoft, Redhat und Canonical, die als Sponsoren agieren. Die PSF ist Herausgeber und Rechteinhaber der Python-Software-Foundation-Lizenz (<https://docs.python.org/3/license.html> Abb. 1.3) und besitzt die Markenrechte an Python. Die Python-Software-Foundation-Lizenz ist ähnlich der BSD-Lizenz und kompatibel mit der GNU General Public License.

1.5.3.2 Die Dokumentation

Im Python-Webportal finden Sie unter dem Menüpunkt DOCUMENTATION und dort Docs zahlreiche Informationen inklusive einer kompletten Dokumentation, FAQs und verschiedenen Tutorials (Abb. 1.4).

Von da aus kann man diese Materialien herunterladen, aber das ist im Grunde gar nicht notwendig. Zum einen ist man ja sowieso meist online, und zudem ist die Dokumentation bereits im Installationspaket von Python enthalten beziehungsweise wird mit installiert, wenn Sie das bei der Installation auswählen.

1.5.3.3 Der Download von Python

Bereits im Dokumentationsbereich steht Ihnen Python zum Download bereit. Aber es gibt auch auf der Einstiegsseite des Webportals einen eigenen Menüpunkt Downloads. Darüber erhalten Sie die aktuellen, aber auch bei Bedarf frühere Versionen für verschiedene Betriebssysteme (Abb. 1.5).

Table of Contents

- History and License
- History of the software
- Agreement for access to or distribution of Python
- PEP 107: AGREEMENT FOR ACCESS TO OR DISTRIBUTION OF PYTHON
- PEP 108: AGREEMENT FOR PYTHON 3.12.0
- BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.8
- CNRI LICENSE AGREEMENT FOR PYTHON 1.4.3
- CNRI LICENSE AGREEMENT FOR PYTHON 3.0 & THEREAFTER
- ZEDO CLAUSE
- BSD LICENSE FOR DOCUMENTATION THE PYTHON 3.12.0 DOCUMENTATION
- License and Acknowledgments for Incorporated Software
- Major Thanks
- Sponsors
- Asyncio and related protocols
- Coroutines
- Implementation
- Execution tracing
- Unicode and Internationalization functions
- IBM, Numeric, Threadsafe Cells
- test_gptk
- Select Issues

History and License

History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <https://www.cwi.nl/>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <https://www.cnri.reston.va.us/>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation, see <https://www.zope.org/>). In 2001, the Python Software Foundation (PSF, see <https://www.python.org/psf/>) was founded, a non-profit organization dedicated to open Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see <https://opensource.org/> for the Open Source Definition). Historically most, but not all, Python releases have also been GPL-compatible, the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 and above	2.1.1	2001-now	PSF	yes

Abb. 1.3 Details zur Python-Lizenz

Browse the docs online or download a copy of your own.

Python's documentation, tutorials, and guides are constantly evolving. Get started here, or scroll down for documentation broken out by type and subject.

Python Docs

See also Documentation Releases by Version

Beginner

- Beginner's Guide
- Python FAQs

Moderate

- Python Periodicals
- Python Books

Advanced

- Python Packaging User Guide
- In-development Docs
- Gentry's Essays

General

- PEP Index
- Python Videos
- Developer's Guide

Python 3.x Resources

- Python 3.12.0 Documentation – (Module index)
- What's new in Python 3.12
- Tutorial
- Library Reference
- Language Reference
- Extending and Embedding
- Reference Site

Porting from Python 2 to Python 3

- FAQ: Porting Python 2
- Final Python 2.7 Release Schedule
- Python 3 Statement
- Porting Python 2 Code to Python 3
- Determining what projects are blocking you from porting to Python 3
- Python 3 Support and Migration

Abb. 1.4 Informationen, Quellen und Tutorials

- Laden Sie sich am besten immer die neueste Version von Python für Ihr Betriebssystem herunter.

1.5.3.4 Die konkrete Installation

Wir gehen nun die Installation für Windows, Linux und macOS einzeln durch, und Sie werden sehen, dass diese in der Regel geradezu trivial ist, wenn es einen Installer gibt. Unter Linux und teils auch macOS kann es aber sein, dass Sie Python in einem Terminal installieren müssen bzw. können, wenn Sie die aktuelle Version haben wollen.

1.5.3.4.1 Die Installation unter Windows

Unser wichtigstes Referenzsystem im Buch ist Windows, und damit wollen wir beginnen. Unter Windows starten Sie einfach den Installer, den Sie auf Ihren PC geladen haben. Dieser startet wiederum mit einem Auswahldialog mit allen wichtigen Informationen und Vorgabeeinstellungen (Abb. 1.6).

Wenn Sie in dem Dialog bereits den Installationsbefehl geben (INSTALL Now), läuft die Installation ohne weitere Interaktion mit Ihnen durch – abgesehen von eventuellen Rückfragen von Windows, zum Beispiel, ob Sie dem Installationsprogramm wirklich erlauben wollen, Dinge auf dem PC zu verändern. Auf dem Rechner hat der Python-Installer danach alle Dateien im voreingestellten Verzeichnis installiert, die man für die Ausführung von Python-Programmen benötigt.

Nun gibt es in dem Dialog u. a. drei optionale Einstellungsmöglichkeiten, die Sie auswählen sollten.

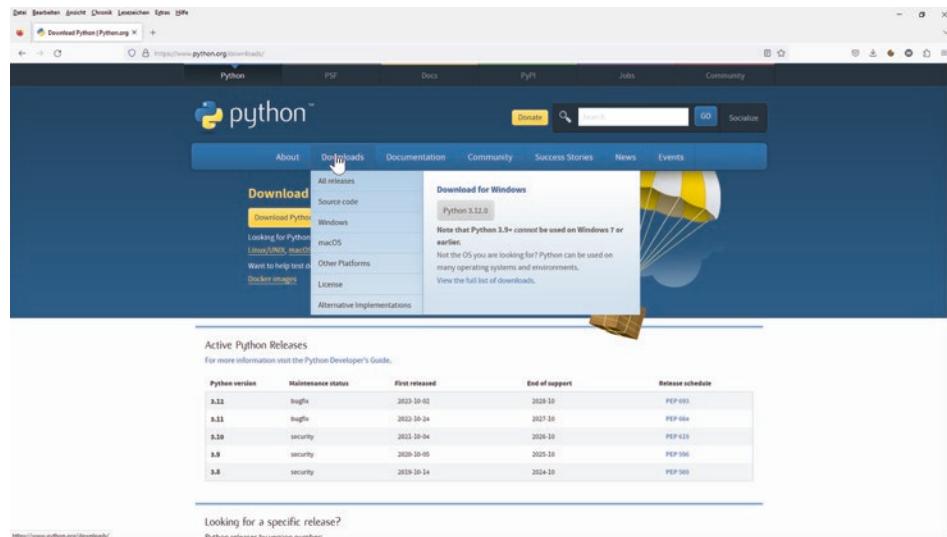


Abb. 1.5 Python laden



Abb. 1.6 Start des Setupassistenten unter Windows 11

1. Die erste Option stellt allen Benutzern auf dem Rechner den sogenannten Launcher für Python zur Verfügung. Das bedeutet, dass diese Anwender dann Python-Dateien direkt ausführen können. Oder anders ausgedrückt: Die Python-Dateierweiterung wird mit dem Python-Interpreter verknüpft (Abb. 1.7).
 2. Besonders zu empfehlen ist das Hinzufügen von Python zur *Path*-Angabe – dem Suchpfad von Windows (Abb. 1.6). Dazu klicken Sie in dem Dialog die entsprechende Option an – bei der im Buch verwendeten Version von Python nennt die sich ADD PYTHON.EXE TO PATH. Wenn Sie die entsprechende Option ausgewählt haben, wird das Installationsverzeichnis in den Suchpfad von Windows aufgenommen, und Sie können Python in der Konsole von jedem Verzeichnis aus aufrufen. Das ist zwar nicht zwingend notwendig, aber sehr bequem.
 3. Achten Sie darauf, dass Sie pip installieren (Abb. 1.7), damit Sie Pakete bequem installieren und verwalten können. Das Paketverwaltungssystem wird mittlerweile auch über reines Python hinaus verwendet.
- Sie können die Installationsverzeichnisse unter Windows selbstverständlich nachträglich dem Suchpfad hinzufügen. Das erfolgt in den Einstellungen zum System unter ERWEITERTE UMGEBUNGS-VARIABLEN. Aber wenn das vom Setup-Assistent schon geleistet wird, ist das natürlich angenehm. Wenn Sie nicht mehr wissen, wo der Installationsordner von Python zu finden ist, können Sie den Befehl *where* verwenden. Geben Sie einfach in der Eingabeaufforderung *where python* an.

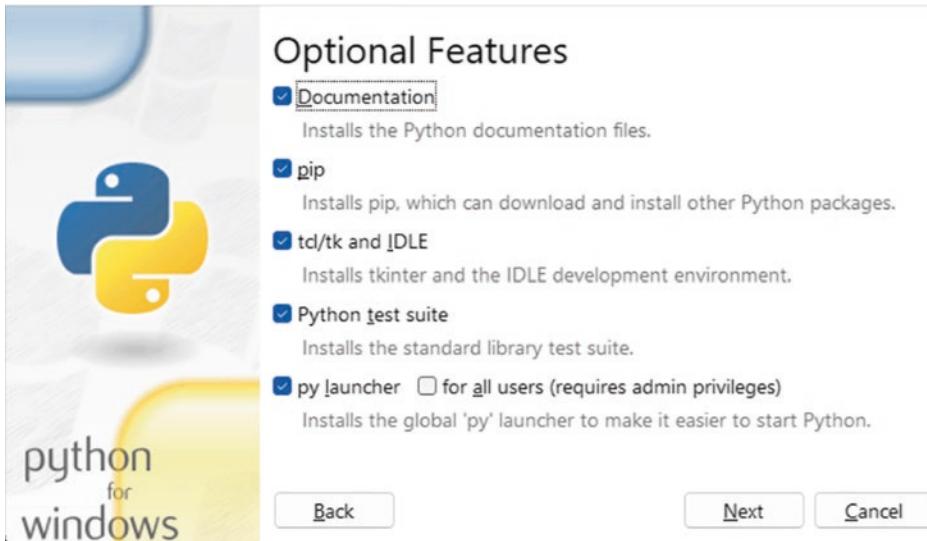


Abb. 1.7 Optionale Features

Nach der Installation gibt es in Windows insbesondere zwei Programme, die von zentraler Bedeutung sind für Python:

- Die **Python-Kommandozeile** – auch Python-Interpreter⁶ oder Python-Shell genannt,
- die Python-GUI, die man mit **IDLE** (Python's Integrated Development Environment) abkürzt.

Auf die Details dazu gehen wir später ein.

Wenn Sie wollen, können Sie vom Einstiegsdialog aus die Arbeit des Assistenten auch individuell anpassen.

Zum einen können Sie optionale Features festlegen (Abb. 1.7). Diese Einstellungen können aber meist in der Voreinstellung bleiben.

Aber sehr interessant sind die erweiterten Optionen im darauffolgenden Anpassungsdialog (Abb. 1.8). Denn insbesondere die Änderung des Installationsorts von Python kann wichtig sein. Ich verwende etwa bei meinem Windows-System aktuell eine Workstation mit einer recht kleinen SSD, aber einer sehr großen HDD. Die Daten und nicht wirklich performancehungrige Programme lagere ich deshalb grundsätzlich auf die größere HDD aus. Aber standardmäßig werden Sie Python meist auf Laufwerk C: installieren.

⁶Was eigentlich etwas ungenau ist, denn der Interpreter an sich ist keine Konsole in dem Sinn. Aber man benutzt den Begriff dennoch im Python-Umfeld.

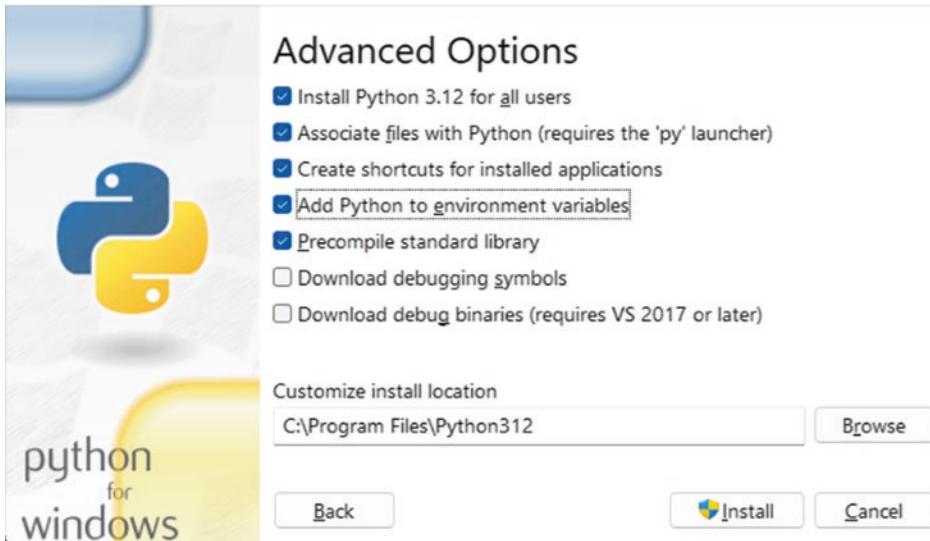


Abb. 1.8 Erweiterte Optionen

Wenn die Installation ohne Probleme beendet ist, erhalten Sie eine Erfolgsmeldung (Abb. 1.9).

Wenn ein Fehler passiert ist (was selbstverständlich weder zu hoffen noch zu erwarten ist), werden Sie natürlich auch über die Art der Probleme informiert.

1.5.3.4.2 Installieren auf einem Linux-System

Python gibt es wie gesagt auch für Linux. Dabei muss man einmal einschränken, dass es verschiedene Formate für die unterschiedlichen Linux-Distributionen gibt. Das macht die Sache etwas unübersichtlich. Aber dafür haben Sie auf der anderen Seite die Flexibilität und können Python sogar individuell angepasst aus Quellcodes für Ihr System erstellen.

Im Prinzip ist das meist gar nicht notwendig. Denn viele Linux-Distributionen installieren Python bereits automatisch mit. Oder sie stellen Python über die distributions-eigene Anwendungsverwaltung zur Verfügung, womit die Installation genauso trivial wie unter Windows abläuft (Abb. 1.10).

Allerdings muss man beachten, dass in der Regel nicht die aktuelle Version von Python in den Installationspaketen beziehungsweise der standardmäßig installierten Version bereitsteht. Sogar halbwegs aktuelle Linux-Versionen (Abb. 1.11) oder auch macOS (Abb. 1.12) stellen noch Python in der Version 2.7.x bereit, und die Besonderheiten der 2er-Versionen werden wir wie gesagt explizit nicht mehr beachten. Wie erwähnt, liegt Python zum Zeitpunkt der Bucherstellung in der Version 3.12.x vor, und diese sollten Sie auch unter Linux zur Verfügung haben.

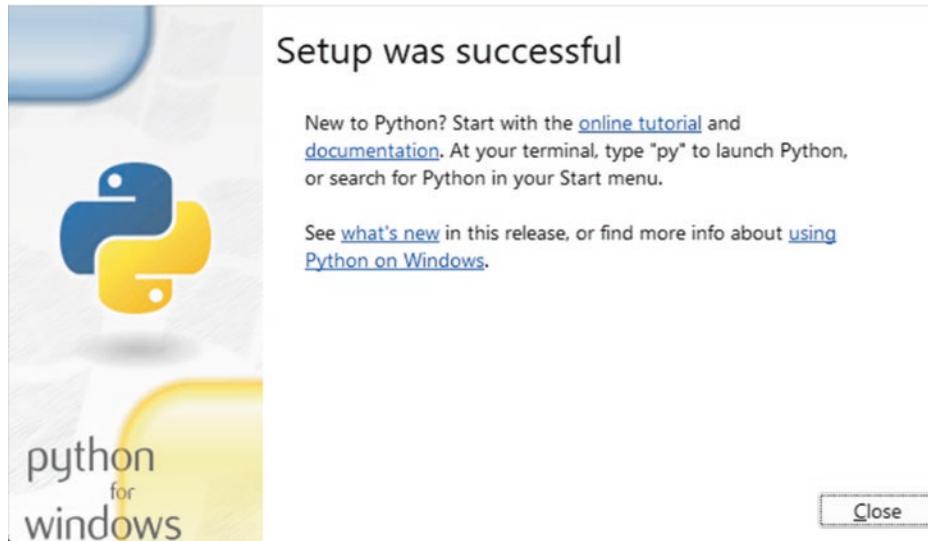


Abb. 1.9 Python wurde installiert

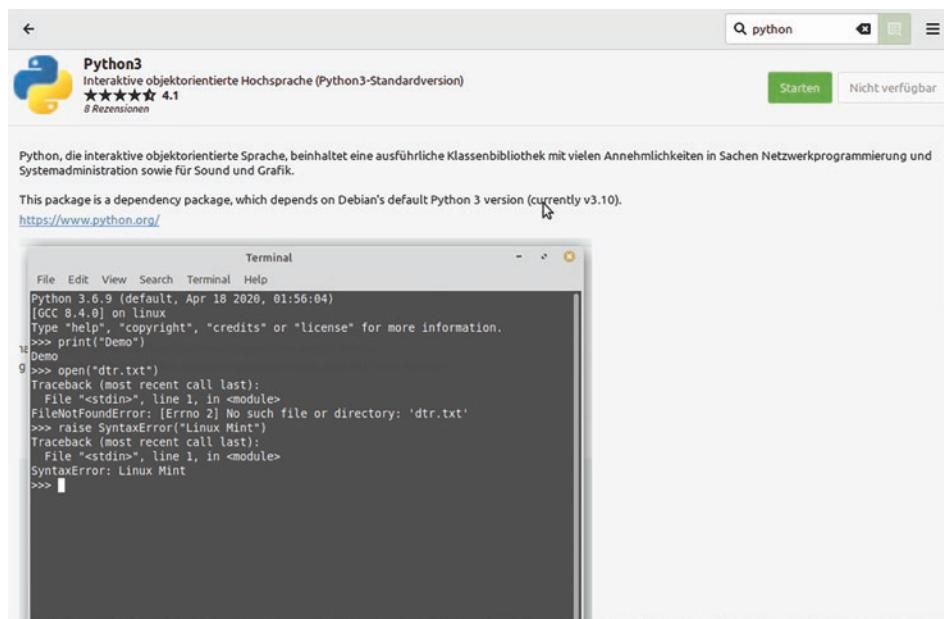


Abb. 1.10 Linux-Distributionen bringen Python eigentlich immer mit

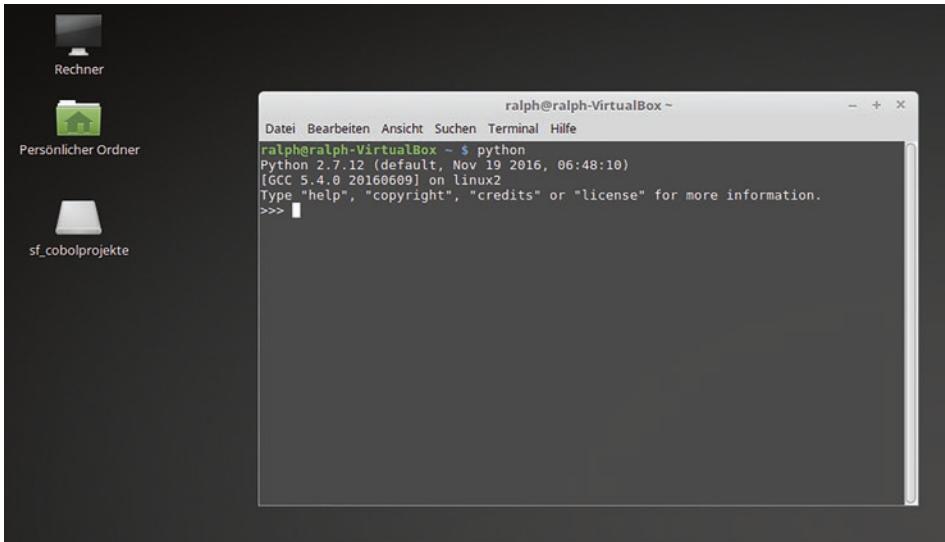


Abb. 1.11 Bei einigen halbwegs aktuellen Linux-Versionen ist standardmäßig noch die Version 2.x vorinstalliert

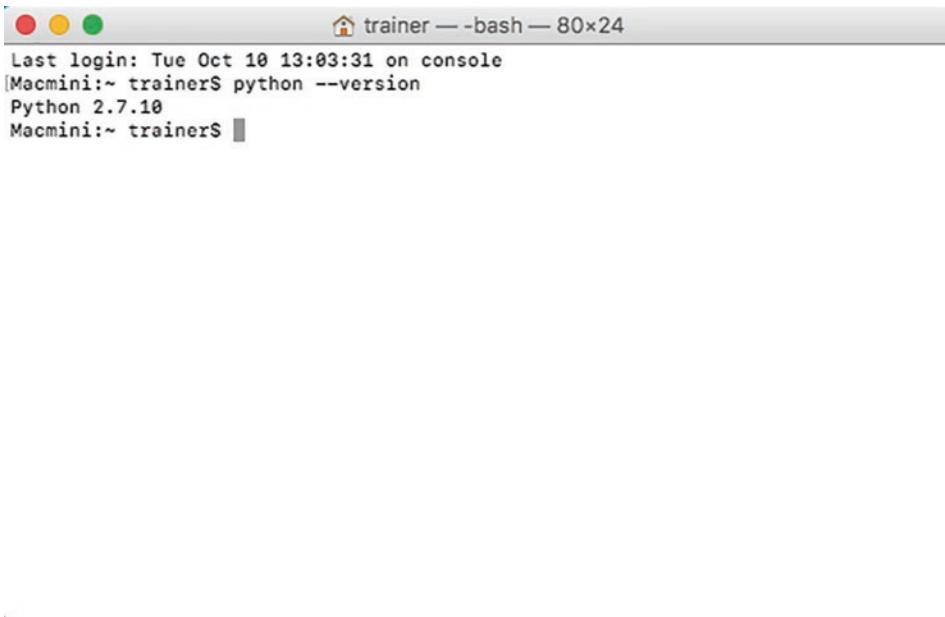


Abb. 1.12 Auch bei etwas älteren, aber immer noch verwendeten macOS findet man manchmal noch die Version 2.7

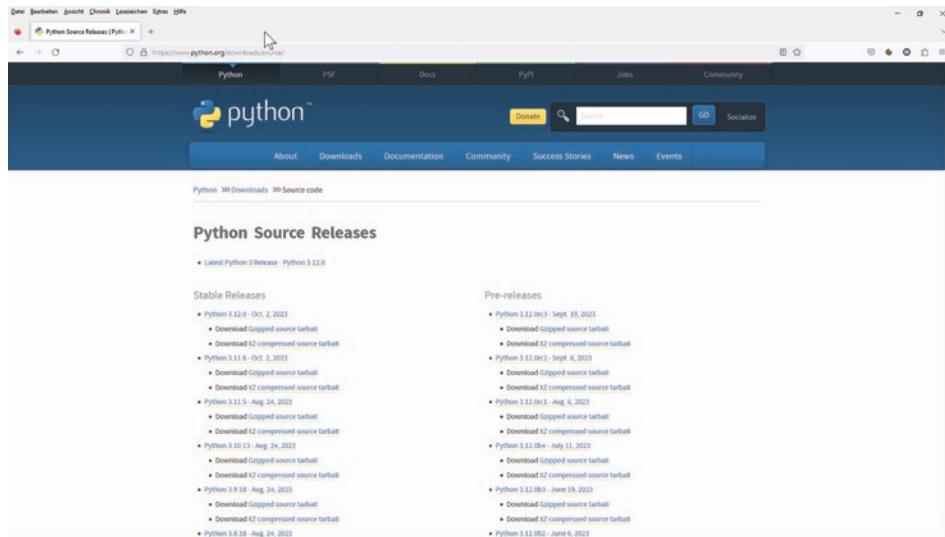


Abb. 1.13 Python für Linux laden

Falls Sie also die neueste Version oder zumindest eine Version der 3er-Schiene von Python nicht über die Anwendungsverwaltung Ihrer Linux-Distribution erhalten, können Sie eine individuelle Installationsdatei (wenn verfügbar) oder auch die Quellcodes laden und dann auf Ihrem Rechner konfigurieren, übersetzen und installieren (Abb. 1.13). Die letztgenannte Option ist meist besonders vielversprechend, um die aktuellste Version von Python zu erhalten.

Wie die dann aber konkret installiert wird, kann sich unterscheiden, und es muss für Details zu einer bestimmten Distribution auf die Dokumentation verwiesen werden – gerade im Fall von Problemen. Aber normalerweise erhalten Sie auf jeden Fall die Quelltextdateien von Python, die Sie auf die übliche Art unter Linux konfigurieren und installieren können. Die Ressourcen liegen dabei als Archiv vor, das Sie erst einmal extrahieren müssen (Abb. 1.14).

Danach öffnen Sie ein Terminal und gehen im allgemeinen Fall wie folgt vor (Abb. 1.15):

1. Wechseln Sie in das Verzeichnis, das bei der Extraktion der Python-Ressourcen angelegt wurde, etwa so:

Beispiel

```
cd Python-3.12.0 ◀
```

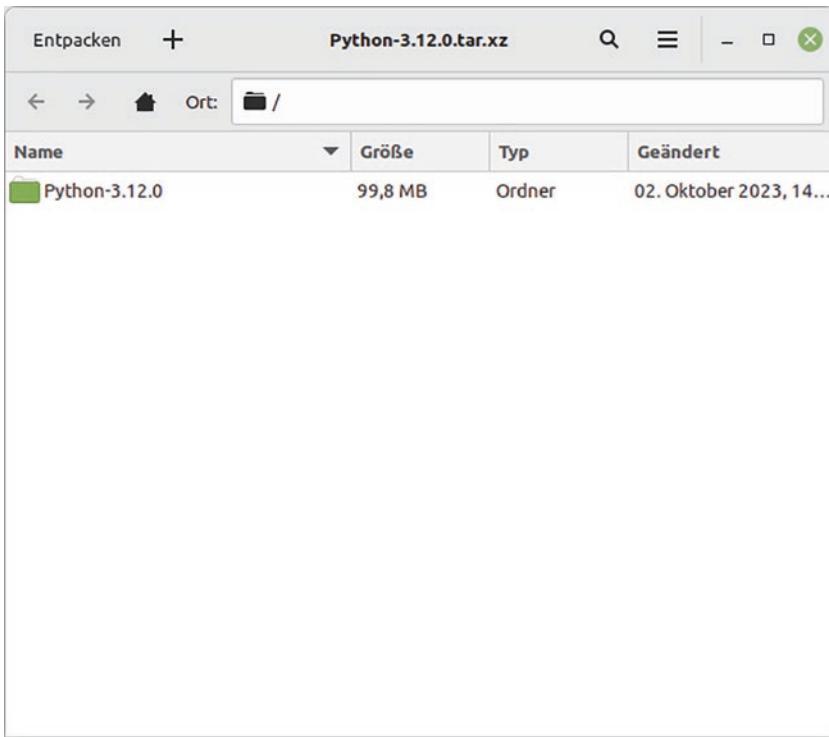


Abb. 1.14 Das Archiv muss extrahiert werden

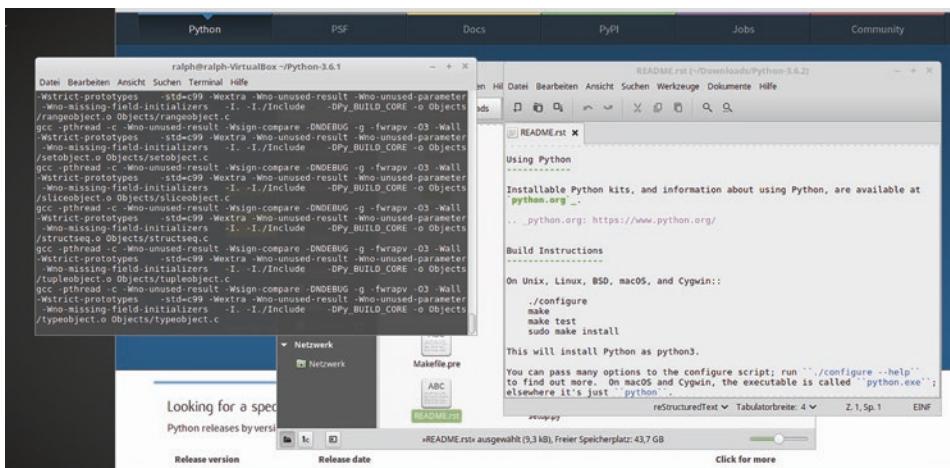


Abb. 1.15 Die Installation läuft

2. Konfigurieren Sie die Ressourcen für Ihren Rechner:

Beispiel

./configure ◀

3. Übersetzen Sie den Quellcode und erstellen Sie eine Make-Datei:

Beispiel

make ◀

4. Testen Sie das Make-File

Beispiel

make test ◀

5. Wenn der Test keine Fehler ergab, erfolgt die konkrete Installation. Das muss man in der Regel allerdings als root erledigen, also so, sofern Sie nicht schon in einem root-Terminal arbeiten oder sich vorher zum root gemacht hatten (dann kann *sudo* entfallen):

Beispiel

sudo make install ◀

Nach der erfolgreichen Installation sollte die neue Python-Version verfügbar sein (Abb. 1.16). Das können Sie etwa testen, indem Sie im Installationsverzeichnis von Python Folgendes eingeben:

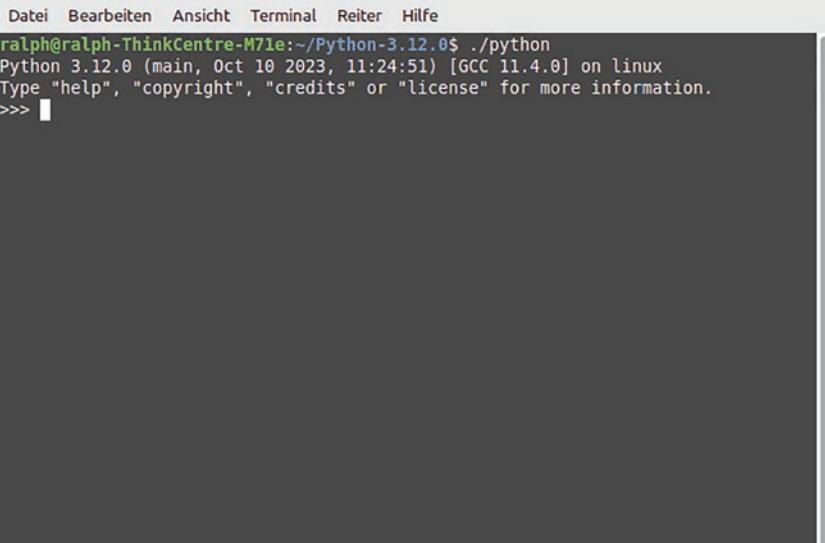
Beispiel

./python ◀

Zu Beginn der nun aktivierten Python-Kommandozeile erkennen Sie die Python-Version (Abb. 1.16).

- ▶ Die Python-Kommandozeile schließen mit der Eingabe *exit()*. Beachten Sie die Klammern.

Auch unter Linux gilt, dass Python für eine bequemere Anwendung im Suchpfad eingetragen sein sollte. Wenn Python bei der Distribution schon installiert wurde oder Sie die Anwendungsverwaltung der Distribution ver-



The screenshot shows a terminal window with a dark background and light-colored text. At the top, there is a menu bar with German labels: Datei, Bearbeiten, Ansicht, Terminal, Reiter, Hilfe. Below the menu, the terminal prompt is shown: ralph@ralph-ThinkCentre-M71e:~/Python-3.12.0\$./python. The output of the command follows, showing the Python version information: Python 3.12.0 (main, Oct 10 2023, 11:24:51) [GCC 11.4.0] on linux. It also includes a note: Type "help", "copyright", "credits" or "license" for more information. A cursor is visible at the end of the line.

Abb. 1.16 Die neue Version wurde installiert

wenden, sollte das automatisch der Fall sein. Installieren Sie Python allerdings direkt oder zusätzlich zu einer vorhandenen Version, müssen Sie sich gegebenenfalls selbst darum kümmern und eventuell auch den Verweis auf die ältere Version aus dem Suchpfad löschen. Wenn Sie nicht (mehr) wissen, wo Python installiert ist, können Sie den Befehl *which* verwenden, um den Installationsordner von Python zu finden. Das geht auch unter macOS. Beachten Sie, dass Python unter Linux manchmal mit *python3* statt *python* benannt wird. Dementsprechend kann es auch sein, dass der Start von Python von manchen Stellen im Verzeichnissystem mit *python3* erfolgen muss.

1.5.3.5 Installieren auf einem Mac

Für die Installation auf einem Mac steht ebenfalls ein Installer zur Verfügung, der genauso einfach wie der Installer unter Windows arbeitet (Abb. 1.17).

Allerdings müssen Sie bei der Installation auf einem Mac eine ganze Reihe an Lizenzbedingungen explizit bestätigen und – zumindest „offiziell“ – vor der Installation diverse Hinweise lesen (Abb. 1.18).

Anschließend läuft aber auch diese Installation normalerweise in einem Rutsch durch (Abb. 1.19).

- ▶ Sie sollten auch auf einem Mac darauf achten, dass Python im Suchpfad zur Verfügung steht. Das sollte aber automatisch der Fall sein.

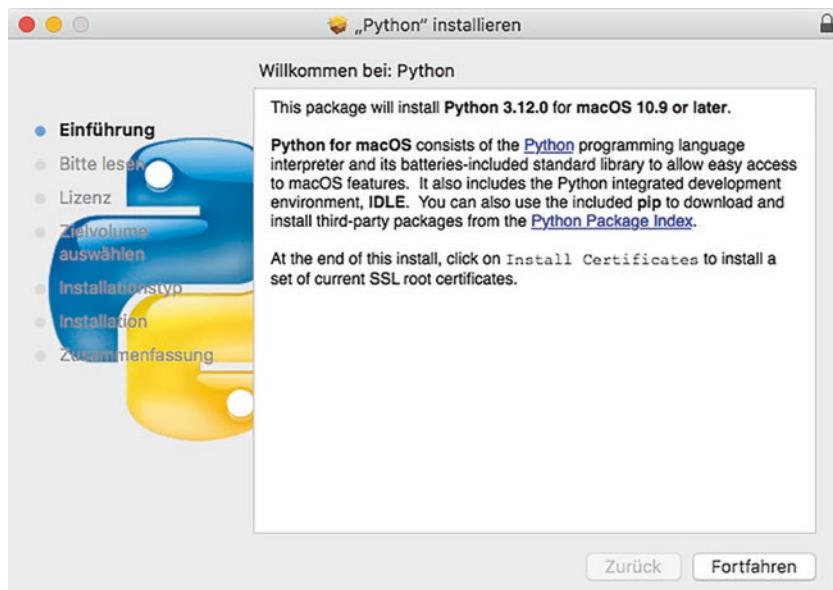


Abb. 1.17 Beginn der Installation auf einem Mac



Abb. 1.18 Im Apple-Umfeld findet man viele Lizenzen, die zu akzeptieren sind

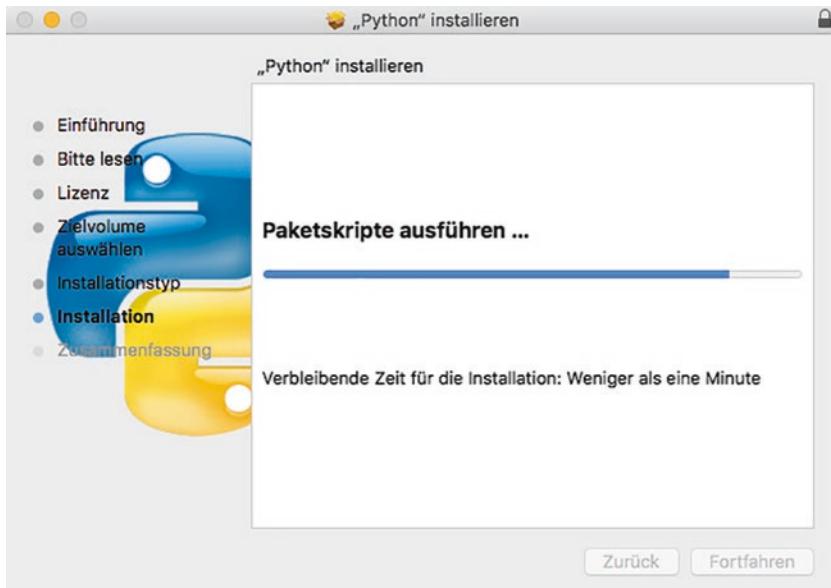


Abb. 1.19 Die Installation auf dem Mac läuft

Auch wenn es vielen Mac-Anwendern nicht bewusst ist, macOS ist UNIX-basiert und damit sehr eng mit Linux verwandt. Das bedeutet, dass es auch für macOS möglich ist, die Python-Ressourcen individuell angepasst zu installieren, indem sie, wie bei Linux beschrieben, in einem Terminal konfiguriert, übersetzt und installiert werden. Die Schrittfolge und die Befehle sind identisch.



Erste Beispiele – Der Sprung ins kalte Wasser

2

2.1 Was behandeln wir in diesem Kapitel?

In diesem Kapitel kommen wir schon zum Erstellen der ersten Python-Programme beziehungsweise -Skripte, und Sie lernen, wie Sie diese übersetzen beziehungsweise ausführen können. Dabei werden wir den Python-Interpreter, die Python-Kommandozeile und die IDLE im ersten Einsatz sehen. Das sind die Tools, die Sie im Umgang mit Python immer wieder benötigen.

2.2 Der Interaktivmodus – die Kommandozeile von Python

Nachfolgend erstellen wir zuerst einmal ein paar ganz einfache Python-Programme¹ mit minimaler Syntax, die wir in der sogenannten **Kommandozeile** von Python (auch **Python Shell** genannt) ausführen wollen. Damit werden die eingegebenen Python-Anweisungen von einem Python-Interpreter unmittelbar ausgeführt. Das nennt man den **Interaktivmodus**.

- ▶ Beachten Sie, dass in Python die **Spalten** beziehungsweise **Einrückungen** bei Quellcodes im Editor eine Bedeutung haben. Das ist bei vielen anderen Sprachen nicht der Fall. Durch Einrückungen werden in Python Strukturen zusammengefasst. Inkorrekte Einrückungen führen zu Fehlern!

¹ „Programme“ ist eigentlich auch schon zu hoch gegriffen. Aber es ist wirklich Python, was Sie hier sehen werden.



Abb. 2.1 Die Einträge für Python im Startmenü von Windows

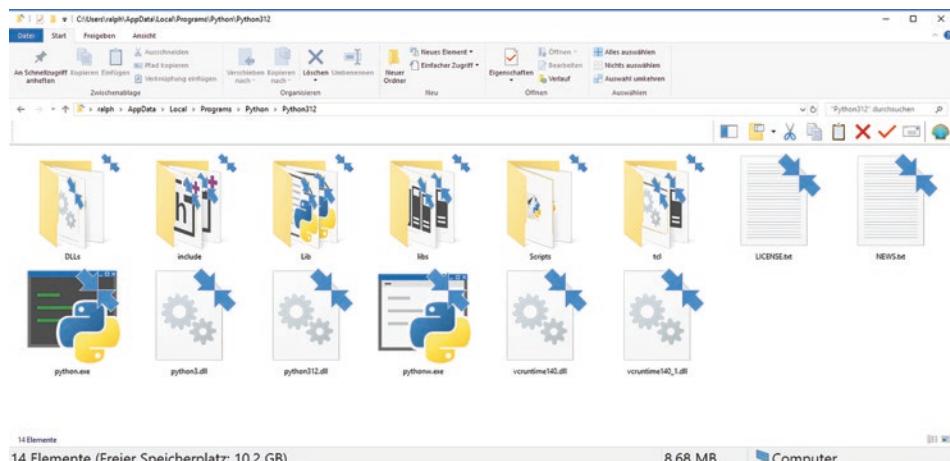


Abb. 2.2 Im Explorer können Sie einen Doppelklick auf python.exe ausführen

Für den Interaktivmodus brauchen wir zuerst die Kommandozeile von Python. An diese gelangen wir auf verschiedene Weisen, die vom Betriebssystem abhängen. Deshalb werden wir an der Stelle noch einmal zwischen Windows, Linux und macOS differenzieren beziehungsweise auf Details eingehen müssen. In der Folge ist das aber kaum noch notwendig.

Die Python-Kommandozeile wurde bei der Installation von Python unter Windows in der Regel im Startmenü Ihres Betriebssystems eingetragen (Abb. 2.1) oder steht über ein Icon auf dem Desktop zur Verfügung.

Wenn das nicht der Fall ist, kann die Python-Kommandozeile in der Konsole² mit dem Befehl `python`, `./python` oder manchmal auch `python3` aus dem Installationsverzeichnis³ oder der grafischen Windows-Oberfläche mit einem Doppelklick auf die Programmdatei aufgerufen werden (Abb. 2.2).

² Angabe von `cmd` im Ausführen-Dialog von Windows.

³ Oder einem beliebigen Verzeichnis, wenn Python dem Suchpfad des Betriebssystems hinzugefügt wurde.

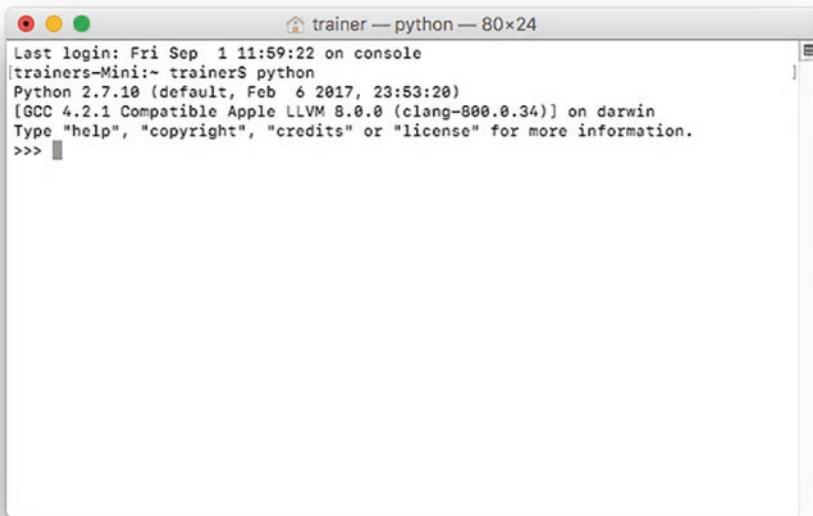
```
ralph@ralph-ThinkCentre-M71e:~$ python3
Python 3.12.0 (main, Oct 10 2023, 11:24:51) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Abb. 2.3 Start der Python-Konsole in Linux

- **Achtung** Beachten Sie, dass es in Abhängigkeit vom Betriebssystem spezifische Dateierweiterungen und auch sehr unterschiedliche Verhaltensweisen für den Start der Python-Kommandozeile gibt.
- Unter Windows werden Sie etwa .exe als Dateierweiterung für das Programm der Python-Kommandozeile finden (also *python.exe*).
 - Unter Linux haben Sie zwar keine Dateierweiterung für die Programmdatei, müssen beim Aufruf in einem Terminal aber in der Regel ./ voranstellen (also *./python* – oder *./python3*, Abb. 2.3), um explizit das aktuelle Verzeichnis anzugeben. Ebenso werden Sie unter Linux nach einer Installation über die im letzten Kapitel beschriebene Vorgehensweise in der Regel keinen Eintrag im Startmenü oder auf dem Desktop vorfinden. Hier zeigt sich, dass in Linux traditionell viel über die Befehlszeile gearbeitet wird.
 - Unter macOS können Sie zwar in einem normalen Terminal *python* eingeben, aber dann gelangen Sie in der Regel zur (veralteten) Standardinstallation von Python (Abb. 2.4). Im Installationsverzeichnis von Python, das Sie etwa über den Finder auswählen können, werden Sie allerdings sehr wahrscheinlich gar keine direkte Aufrufmöglichkeit der Python-Kommandozeile finden. Da zeigt sich aus meiner Sicht, dass man in der Welt von Apple – im Gegensatz zu Linux – nicht wirklich die Befehlszeile mag. Wenn Sie allerdings IDLE (Abschn. 2.4, Abb. 2.5) starten, wird Ihnen die aktuelle Python-Shell auch unter macOS angezeigt (Abb. 2.5).

2.2.1 Das Prompt

Was Sie nach dem Start der Kommandozeile in jedem Fall sehen, ist eine Konsole beziehungsweise ein Terminal oder eine Shell mit drei Größer-Zeichen unterhalb einiger Hinweise. Das ist die Python-Eingabeaufforderung (**Prompt**). Hinter dem Prompt kann man alles schreiben, was Python auswerten kann. Mit der EINGABE-Taste wird die Anweisung dann direkt ausgeführt, denn wir sind hier ja wie gesagt im interaktiven Modus. Dabei kommt dann nach der Bestätigung der Anweisung der Python-Interpreter zum Einsatz.



```
Last login: Fri Sep 1 11:59:22 on console
[trainers-Mini:~ traine$ python
Python 2.7.10 (default, Feb  6 2017, 23:53:20)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Abb. 2.4 Der direkte Aufruf führt in der Regel unter macOS zu einer veralteten Version von Python

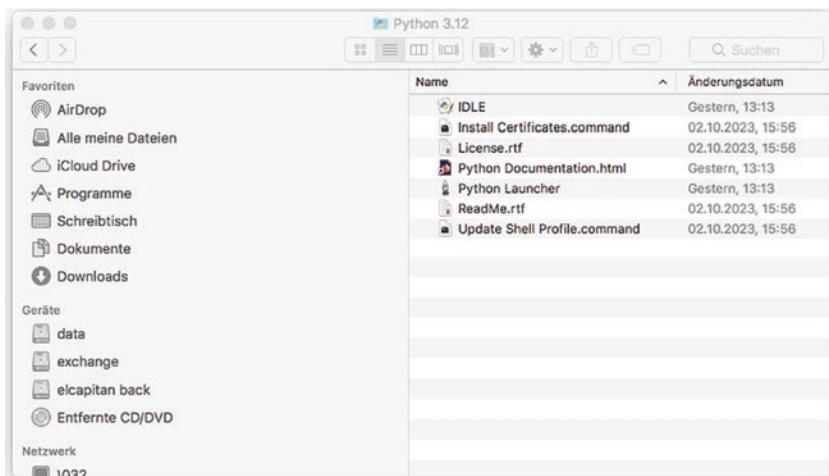


Abb. 2.5 Zur aktuellen Shell gelangen Sie auf einem Mac am einfachsten über IDLE

```
ralph@ralph-ThinkCentre-M71e:~$ python3
Python 3.12.0 (main, Oct 10 2023, 11:24:51) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> "Die erste Python-Anweisung"
'Die erste Python-Anweisung'
>>> []
```

Abb. 2.6 Die Anweisung wurde ausgeführt – hier unter Linux

2.2.1.1 Die erste Anweisung

Im einfachsten Fall besteht eine gültige Python-Anweisung einfach aus einem Text, den man in Anführungsstriche setzen muss – ein String beziehungsweise Stringliteral.

► Eine Zahl oder eine andere Form von Wert (meist ein Text) nennt man in einer Programmiersprache ein **Literal**.

- Geben Sie das nachfolgende Listing direkt hinter dem Prompt ein:

```
"Die erste Python-Anweisung"
```

- Bestätigen Sie die Eingabe mit der EINGABETASTE.

Die Kommandozeile zeigt genau diesen Text auch wieder an – nur ohne die vorangestellten drei Größer-Zeichen, die ja eine explizite Eingabe anfordern (Abb. 2.6).

Das mag trivial erscheinen, ist es aber nicht wirklich, denn es wird tatsächlich Python ausgeführt. Python versucht, die Eingabe in der Kommandozeile in der Tat auszuwerten. Nur gibt es nichts zu berechnen oder sonst weiter auszuwerten. Es wird nur ein Literal vorgefunden. Deshalb wird einfach der String wieder ausgegeben. Dennoch – das war bereits ein klassischer Turn-around nach dem EVA-Prinzip (Eingabe-Verarbeitung-Ausgabe). Und dass etwas „passiert“ ist, können Sie auch daran erkennen, dass der Text bei der Ausgabe nicht mehr in doppelten, sondern nur einfachen Hochkommata eingeschlossen wird.

► Die Kommandozeile von Python verfügt über einen Tastaturspeicher. Damit kann man mit der Pfeiltaste vorher eingegebene Befehle wiederherstellen und neu ausführen. Vorher können sie bei Bedarf modifiziert werden.

Diese Anweisung lässt sich sowohl unter Linux (Abb. 2.6) als auch macOS (Abb. 2.8) vollkommen analog wie unter Windows (Abb. 2.7) ausführen.⁴

⁴Was ja fast für alle Python-Anweisungen gelten wird.

```
C:\Windows\System32>python
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct  2 2023, 13:03:39) [MSC v.1935 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> "Die erste Python-Anweisung"
'Die erste Python-Anweisung'
>>>
```

Abb. 2.7 Die Anweisung unter Windows

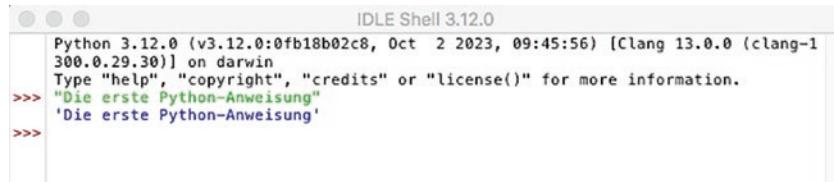


Abb. 2.8 So sieht das in der Python-Shell unter macOS aus

- ▶ Die Kommandozeile können Sie durch die Eingabe von `quit()` oder `exit()` schließen. Die Anweisung muss klein geschrieben werden.
- ▶ Python beachtet grundsätzlich Groß- und Kleinschreibung! Das gilt auch für die Befehle zum Aktivieren der Hilfe oder dem Schließen der Kommandozeile.

2.2.2 Der Hilfemodus in der Kommandozeile

Oft benötigen Sie bei der Eingabe von Python-Anweisungen Informationen oder Hilfe. Wenn Sie `help()` in der Kommandozeile eingeben, gelangen Sie in den Hilfemodus der Kommandozeile.

- Geben Sie in der Kommandozeile hinter dem Prompt die folgende Anweisung ein:

Beispiel

`help() ◀`

- Bestätigen Sie die Eingabe.

Sie haben nun den Hilfemodus aktiviert. Sie erkennen den Hilfemodus auch daran, dass nun keine drei Größer-Zeichen, sondern nur noch ein Größer-Zeichen für das Prompt steht (Abb. 2.9).

- ▶ Den Hilfemodus können Sie durch die Eingabe von *quit* oder kurz *q* wieder verlassen. Danach sind Sie wieder im „normalen“ Interaktivmodus.

2.2.2.1 Hilfe zu Schlüsselwörtern

Im Hilfemodus kann man sich etwa alle Schlüsselwörter von Python über die Eingabe *keywords* anzeigen lassen.

- Geben Sie im Hilfemodus in der Kommandozeile hinter dem Prompt die folgende Anweisung ein:

Beispiel

keywords ◀

- Bestätigen Sie die Eingabe.

```
C:\Windows\System32>python
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct  2 2023, 13:03:39) [MSC v.1935 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> help()

Welcome to Python 3.12's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the internet at https://docs.python.org/3.12/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help>
```

Abb. 2.9 Der Hilfemodus

```
If this is your first time using Python, you should definitely check out  
the tutorial on the internet at https://docs.python.org/3.12/tutorial/.
```

```
Enter the name of any module, keyword, or topic to get help on writing  
Python programs and using Python modules. To quit this help utility and  
return to the interpreter, just type "quit".
```

```
To get a list of available modules, keywords, symbols, or topics, type  
"modules", "keywords", "symbols", or "topics". Each module also comes  
with a one-line summary of what it does; to list the modules whose name  
or summary contain a given string such as "spam", type "modules spam".
```

```
help> keywords
```

```
Here is a list of the Python keywords. Enter any keyword to get more help.
```

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

```
help> ■
```

Abb. 2.10 Auflistung der Schlüsselwörter

Sie haben eine Auflistung aller Schlüsselworte von Python erhalten (Abb. 2.10). Viele sind das im Vergleich zu anderen Sprachen nicht, wie schon erwähnt wurde.

Hier sehen Sie beispielsweise, dass Python ein Schlüsselwort *if* kennt. Dazu kann man sich als Vertreter die genaueren Informationen einmal anzeigen lassen.

- Geben Sie in der Kommandozeile hinter dem Prompt des Hilfemodus die folgende Anweisung ein:

Beispiel

```
if ◀
```

Sie erhalten detaillierte Informationen aus der Dokumentation, wie man *if* verwenden kann (Abb. 2.11). Unter Umständen müssen Sie den Hilfetext scrollen, wenn er nicht vollständig angezeigt werden kann.

```
True          def          if           raise
and          del          import       return
as           elif         in          try
assert       else         is           while
async        except       lambda      with
await        finally      nonlocal   yield
break        for          not

help> if
The "if" statement
*****
The "if" statement is used for conditional execution:

if_stmt ::= "if" assignment_expression ":" suite
          ("elif" assignment_expression ":" suite)*
          ["else" ":" suite]

It selects exactly one of the suites by evaluating the expressions one
by one until one is found to be true (see section Boolean operations
for the definition of true and false); then that suite is executed
(and no other part of the "if" statement is executed or evaluated).
If all expressions are false, the suite of the "else" clause, if
present, is executed.

Related help topics: TRUTHVALUE

help> -
```

Abb. 2.11 Hilfe zu einem konkreten Schlüsselwort

2.3 Anweisungen in (echten) Quelltext auslagern

Wenn Sie in der Kommandozeile direkt Python-Anweisungen notieren, werden diese nach Eingabe der ENTER-Taste unmittelbar interpretiert, also einzeln und nacheinander im Rahmen der speziellen Python-Umgebung (Interaktivmodus). In der Regel wird man aber Python-Anweisungen in eine extra Textdatei auslagern – den Quelltext (auch Sourcecode oder Quellcode genannt), der dann ohne Bestätigen der einzelnen Anweisungen abgearbeitet wird. Das hat auch den Vorteil, dass die Anweisungen wiederverwendet werden können. Diesen Klartext kann man sogar mit einem beliebigen Editor erstellen.

- Die übliche Dateierweiterung für Python-Quellcode ist `.py`.

Das Python-Programm kann dann in einer normalen Konsole/Terminal über den Python-Interpreter ausgeführt werden. Dieser steht ja überall zur Verfügung, wenn man das Installationsverzeichnis von Python dem Suchpfad des Betriebssystems hinzugefügt hat. Andernfalls funktionieren natürlich auch explizite Pfadangaben.

- ▶ Erzeugen Sie am besten einen Ordner für Ihre Python-Quelltexte. Es ist zudem sinnvoll, dass man darin dann weitere Unterordner für einzelne Projekte erstellt.

- Erstellen Sie in einem Editor wie Notepad++ folgende Syntax:

```
"Das ist ein Python-Quelltext"
```

Achten Sie darauf, dass die Anweisung direkt in der ersten Spalte beginnt! Es darf keine Einrückung vorhanden sein – sonst kommt es zu einem Fehler. Speichern Sie die Datei in Ihrem Python-Arbeitsverzeichnis unter dem Namen *Stringliteral.py*.

- Öffnen Sie eine Konsole (etwa mit dem Befehl *cmd* unter Windows oder das Terminal-Icon, das Sie in allen Referenzbetriebssystemen vorfinden sollten).
- Wechseln Sie mit dem *cd*-Befehl in das Verzeichnis, in dem Sie die Python-Datei gespeichert haben, etwa so:

Beispiel

```
cd PythonQuellcodes  
cd kap 2 ◀
```

- ▶ Sollten Sie mehrere *cd*-Befehle in einem Schritt eingeben (also gleich mehrere Verzeichnisse), achten Sie darauf, dass man unter Windows einen Backslash zum Trennen der Verzeichnisse verwendet, während unter macOS und Linux der Slash notwendig ist.
- Geben Sie in der Kommandozeile hinter dem Prompt die folgende Anweisung ein:

Beispiel

```
python Stringliteral.py ◀
```

- Bestätigen Sie die Eingabe mit ENTER.

Das war es, wobei – wie schon erwähnt – unter Umständen auch *python3* statt *python*5 Dazu brauchen wir eine geeignete Funktion, die wir im nächsten Kapitel behandeln werden. Wir kommen auf diese Schritte also zurück.

⁵ Das funktioniert so nur im Interaktivmodus.

```
F:\PythonQuellcodes\kap2>python Stringliteral.py
  File "Stringliteral.py", line 1
    ^Das ist ein Python-Quelltext"
IndentationError: unexpected indent
F:\PythonQuellcodes\kap2>_
```

Abb. 2.12 Bereits ein falsches Leerzeichen führt zu einem Fehler

Aber ich will Ihnen dennoch beweisen, dass der Interpreter den Quelltext interpretiert und ausgeführt hat. Und der Beweis läuft darüber, dass ich einfach einen Fehler mache.

- Notieren Sie einmal vor dem Stringliteral ein Leerzeichen.
- Speichern Sie die Datei.
- Rufen Sie erneut den Python-Interpreter mit dem Dateinamen als Parameter auf.

Bereits dieses eine Leerzeichen in der ersten Spalte des Quelltextes genügt, dass das „Programm“ nicht mehr ausgeführt wird (Abb. 2.12).

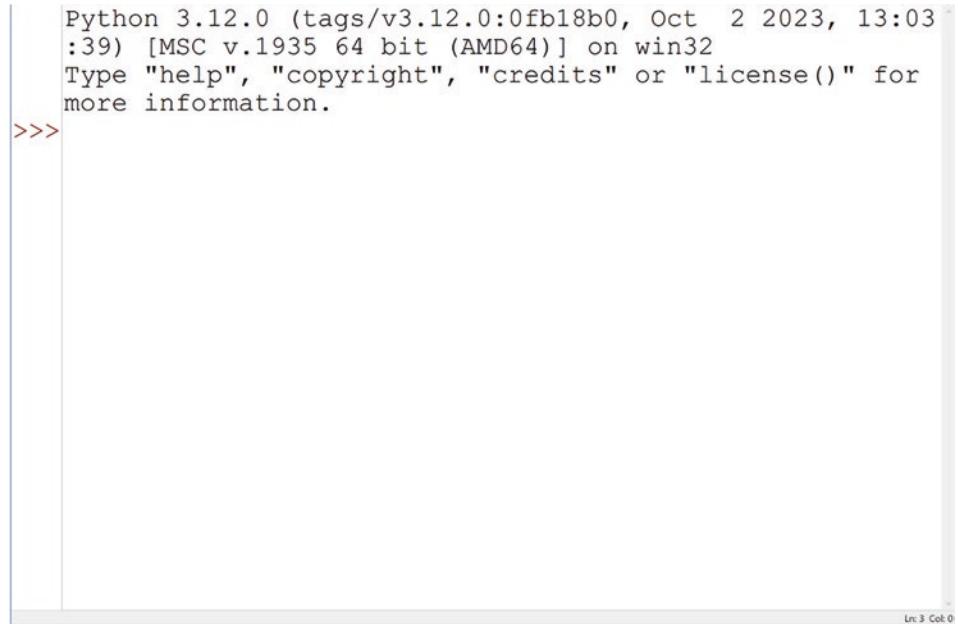
Aber für uns ist dies der Beweis, dass der Python-Interpreter den Quellcode genommen und verarbeitet hat. Und Ihnen sollte ein für alle Mal klar sein, dass Leerzeichen bei der Einrückung eine Bedeutung in Python haben.

2.4 Von IDLE & Co bis zu Python in der Cloud

Die integrierte Entwicklungsumgebung IDLE, die zum Python-System gehört und meist mit Python automatisch installiert wird, besteht im Wesentlichen aus einem Multi-Fenster-Texteditor mit Dateizugriff, Syntaxhervorhebung, Auto vervollständigung, intelligentem Einzug und ein paar anderen Features sowie einer Shell beziehungsweise Kommandozeile mit Syntax-Hervorhebung. Dazu gibt es einen integrierten Debugger und die Möglichkeit zur Ausführung des Programms direkt aus der IDE. Sie ist für eine IDE ziemlich einfach gehalten, wenn man Eclipse, XCode, NetBeans oder Visual Studio im Vergleich betrachtet. Das macht den Umgang mit IDLE aber auch ziemlich leicht.

Beachten Sie, dass sich IDLE auf verschiedenen Plattformen unterscheiden kann. Wir betrachten als Referenz die Version für Windows. Aber der Kern sollte bei allen Versionen von IDLE identisch sein.

Wenn IDLE gestartet, wird Ihnen in der Regel die bereits bekannte Kommandozeile mit dem Interaktivmodus angezeigt (Abb. 2.13). Aber darüber hinaus gibt es einige weitere Möglichkeiten, die über verschiedene Menübefehle angeboten werden, von dem Anpassen von IDLE selbst über den Aufruf der Dokumentation bis zum Laden von Quelltext und dem Ausführen eines Python-Programms aus der IDE heraus. Wer mit einem



The screenshot shows the Python 3.12.0 IDLE shell window. It displays the following text:

```
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct 2 2023, 13:03  
:39) [MSC v.1935 64 bit (AMD64)] on win32  
Type "help", "copyright", "credits" or "license()" for  
more information.  
>>>
```

In the bottom right corner of the window, there is a small status bar with the text "Ln: 3 Col: 0".

Abb. 2.13 Das IDLE-Fenster nach dem Start

modernen Programm wie einer Textverarbeitung oder gar einer der viel mächtigeren oben genannten IDEs vertraut ist, wird sich wahrscheinlich sofort zurechtfinden.

Vor allen Dingen ist das Menü zum Umgang mit Dateien (also Quellcode) wichtig.

- Öffnen Sie über das Menü FILE und dann OPEN die Datei *Stringliteral.py*, die wir vorhin erstellt haben. Sie gelangen zu dem integrierten Python-Editor von IDLE (Abb. 2.14). Dieser stellt die oben beschriebenen Features zur bequemen Arbeit mit Python bereit.
- Korrigieren Sie wieder den Fehler mit dem Leerzeichen.
- Speichern Sie wieder die Datei über das Menü FILE (Abb. 2.15).

2.4.1 Weitere IDEs und Editoren für Python

Neben IDLE wurden in den vergangenen Jahren einige vollwertige Entwicklungs-umgebungen (IDEs) für Python entwickelt – beispielsweise Eric Python IDE (<http://eric-ide.python-project.org/index.html>) oder PyCharm (<https://www.jetbrains.com/pycharm/>).

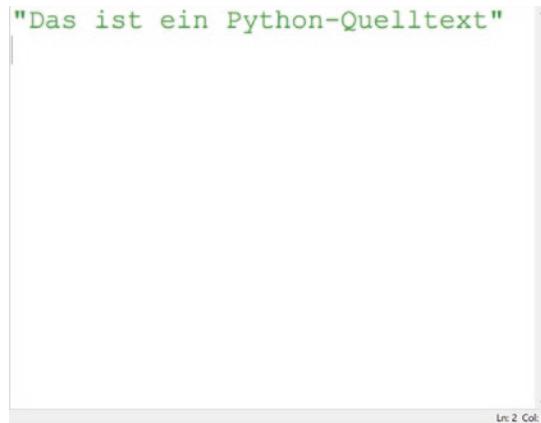


Abb. 2.14 Der integrierte Python-Editor von IDLE

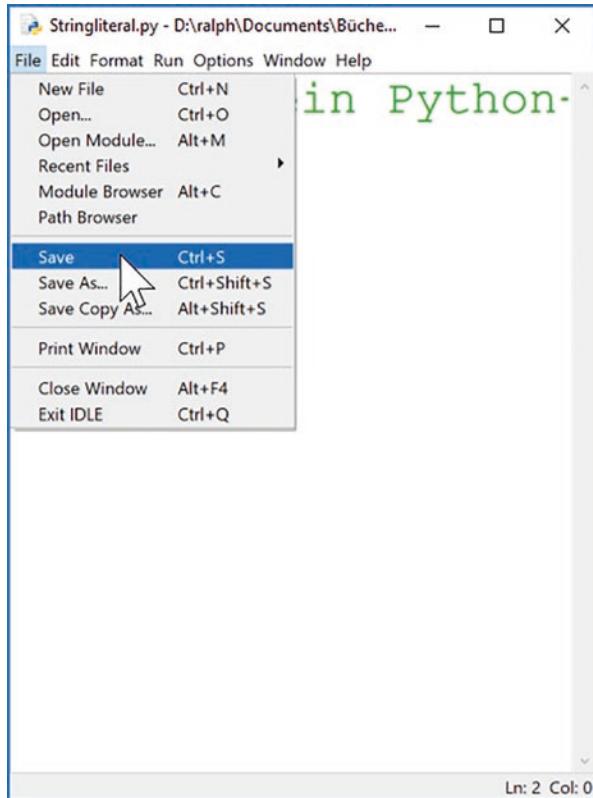
Des Weiteren existieren Plug-ins/Erweiterungen für größere IDEs wie Eclipse, Visual Studio, IntelliJ (<https://www.jetbrains.com>) und NetBeans oder aufbauende Derivate wie PyDev (<http://www.pydev.org/>). Verschiedene Texteditoren für Programmierer lassen sich gegebenenfalls auch für Python anpassen und bieten dann Syntaxunterstützung etc. oder haben bereits von Anfang an so eine Unterstützung implementiert (etwa Notepad++ – <https://notepad-plus-plus.org/>).

Besonders interessant ist Visual Studio Code (<https://code.visualstudio.com/>), das sich über diverse Extensions erweitern lässt und so auch für Python eine hervorragende Unterstützung bietet, die über die einfachen Möglichkeiten von Editoren oder auch IDLE hinausgehen, aber auf der anderen Seite nicht so schwergewichtig wie etwa Eclipse oder Visual Studio ist (Abb. 2.16).

2.4.2 Cloud und RIA

In der letzten Zeit arbeitet man bei der Programmierung immer häufiger online bzw. in der Cloud. Sollten Sie beispielsweise Github verwenden, gibt es dort nicht nur die Möglichkeit, die Ressourcen in der Cloud zu versionieren und bereitzustellen, sondern ebenfalls eine Möglichkeit, eine Art virtuelles Visual Studio Code (bei Bedarf auch einen anderen browserbasierten Editor) mit Namen **Codespaces** im Rahmen eines Browsers als RIA (Rich Internet Application) auszuführen und sämtliche Quellcodes unmittelbar in die Cloud auszulagern als auch dort zu bearbeiten, bis hin zur Ausführung, solange das in einem Browser möglich ist (Abb. 2.17).

Abb. 2.15 Mit IDLE Quellcode speichern



GitHub Codespaces bietet cloudbasierte Entwicklungsumgebungen für viele Aktivitäten rund um die IT – egal, ob es sich um ein langfristiges Projekt oder eine kurzfristige Aufgabe handelt. Dazu werden sogenannte Umgebungen bereitgestellt (eben auch für Python), was die „Backend“-Hälfte von GitHub Codespaces bezeichnet (Abb. 2.18).

Hier findet die gesamte mit der Softwareentwicklung verbundene Rechenleistung statt: Kompilieren, Debuggen, Wiederherstellen usw. Wenn Sie an einem neuen Projekt arbeiten, eine neue Aufgabe übernehmen oder eine PR überprüfen müssen, können Sie einfach eine in der Cloud gehostete Software einrichten. Umgebung und GitHub Codespaces kümmert sich um die korrekte Konfiguration. Es konfiguriert automatisch alles, was Sie für die Arbeit an Ihrem Projekt benötigen: Quellcode, Laufzeit, Compiler, Debugger, Editor, benutzerdefinierte Konfigurationen, relevante Editorerweiterungen und mehr. Allerdings müssen Sie bei Interesse über ein entsprechendes Konto verfügen, das es jedoch für verschiedene Gruppen auch kostenlos gibt. Details müssten Sie auf der entsprechenden Webseite eruieren.

Auch sonst gibt es im Umfeld von Python einige Möglichkeiten, um über einen Browser zu arbeiten. Anaconda (<https://anaconda.org>) und das Jupyter Notebook, die oft

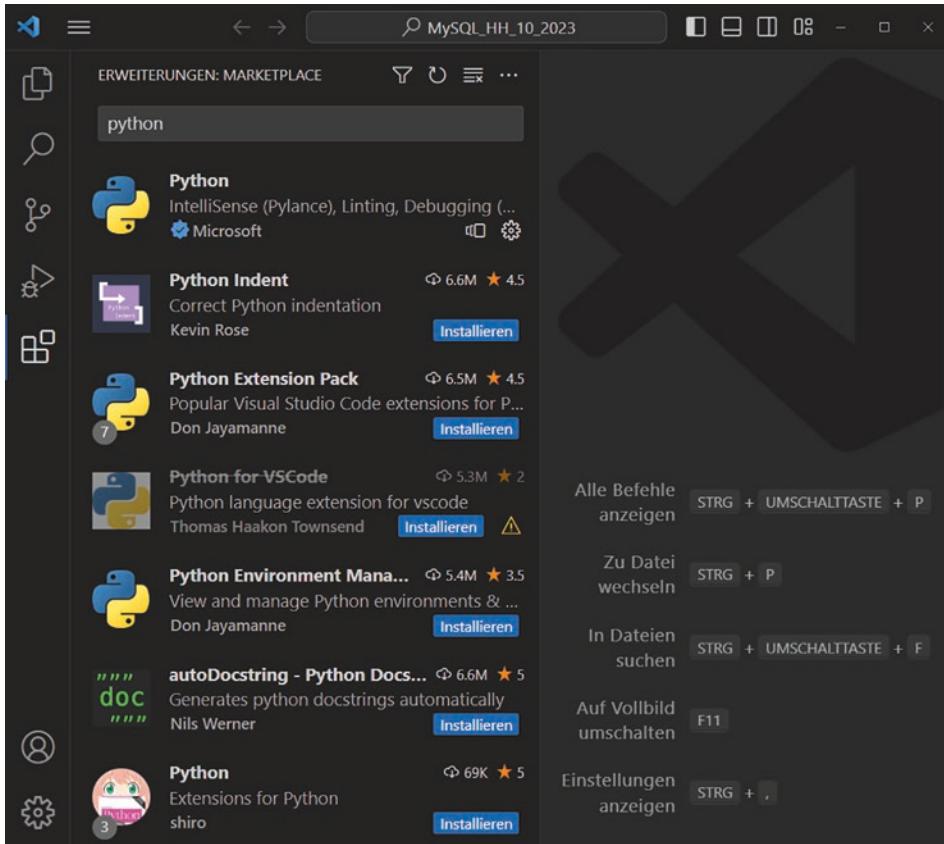


Abb. 2.16 Visual Studio Code bietet hervorragende Unterstützung für Python und kann auch bei Bedarf erweitert werden

in Kombination zum Einsatz kommen, sollen hier exemplarisch für populäre Varianten genannt werden.

Anaconda (Abb. 2.19) ist eine quelloffene Distribution für die Programmiersprachen Python und R, die unter anderem die Entwicklungsumgebung Spyder, den Kommandozeileninterpreter IPython und die besagte Webanwendung Jupyter Notebook (Abb. 2.20) enthält. Fokus liegt vor allem auf der Verarbeitung von großen Datenmengen, Vorphersageanalyse und wissenschaftlichem Rechnen. Mit der Paketverwaltung **conda** gibt es dort ebenfalls eine Alternative zu pip, das in Python quasi den Standard bei der Paketverwaltung darstellt.

Das vollständige Jupyter-Projekt stellt mehrere Softwareprodukte (nicht nur das Notebook) für interaktive wissenschaftliche Datenauswertung und wissenschaftliche Berechnungen bereit. Der Name Jupyter bezieht sich auf die drei wesentlichen Programmiersprachen Julia, Python und R.

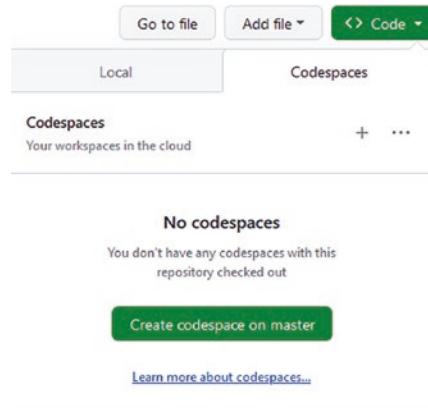


Abb. 2.17 Ein entsprechendes Konto vorausgesetzt, können Sie in Github über das Code-Register Codespaces nutzen

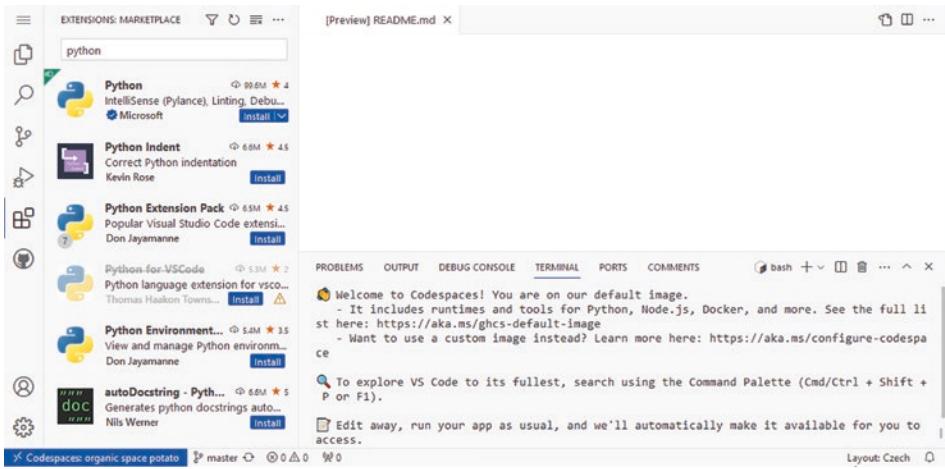


Abb. 2.18 Codespaces kann als eine Art virtuelles Visual Studio Code genutzt werden

Beim Jupyter Notebook wird als webbasierte Anwendung wie bei Codespaces die Oberfläche im Browser bereitgestellt (Abb. 2.21), welche dabei auf weit verbreiteten Open-Source-Programmbibliotheken aufbaut:

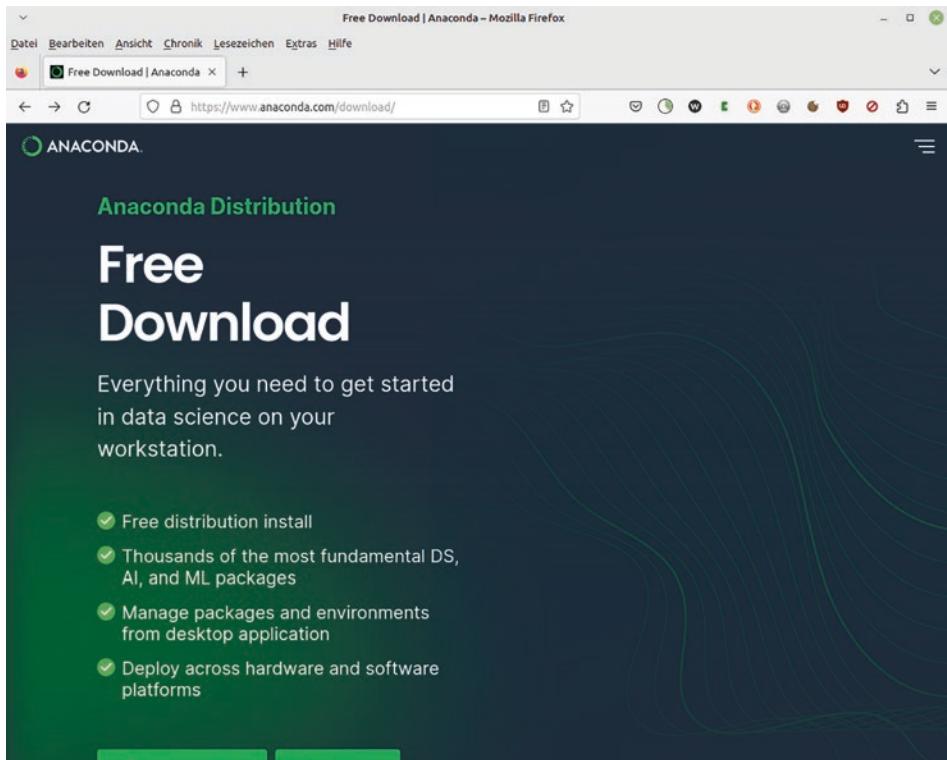


Abb. 2.19 Die Webseite von Anaconda



Abb. 2.20 Gerade Linux-Distributionen stellen das Jupyter Notebook oft schon bereit

- IPython – Kommandozeileninterpreter zum interaktiven Arbeiten mit der Programmiersprache Python.
- ZeroMQ – asynchrone Nachrichtenaustauschbibliothek.
- Tornado – nicht blockierender Webserver sowie ein einfaches Mikro-Webframework in Python.
- jQuery – JavaScript-Bibliothek, die Funktionen zur DOM-Navigation und -Manipulation zur Verfügung stellt.

- Bootstrap – Frontend-CSS-Framework mit Gestaltungsvorlagen für Navigations- und andere Oberflächengestaltungselemente.
- MathJax – JavaScript-Bibliothek, die mathematische Formeln und Gleichungen, die in LaTeX und MathML Markup geschrieben wurden, in Webbrowsersn grafisch darstellt.

Im Hintergrund der RIA (u. U. in der Cloud, aber ebenso lokal oder im eigenen Netzwerk möglich – Abb. 2.21) läuft ein Server, der die Webseiten ausliefert und den Python-Code ausführt.

Abgelegt werden die Daten in einem offenen Format, das auf JSON basiert. Ein solches Jupyter-Notebook-Dokument im JSON-Format besteht aus einer Liste von Eingabe- und Ausgabezellen, die auch unabhängig ausgeführt werden können. Es kann jeweils Code, Text und Graphen oder Diagramme (Plots) enthalten und ist versioniert. Seine Dateiendung ist *.ipynb*. Ein Jupyter-Notebook-Dokument kann durch die JSON-Basis leicht in verschiedene Formate konvertiert und ausgegeben werden, z. B. HTML, PDF, LaTeX und Folien für Präsentationen.

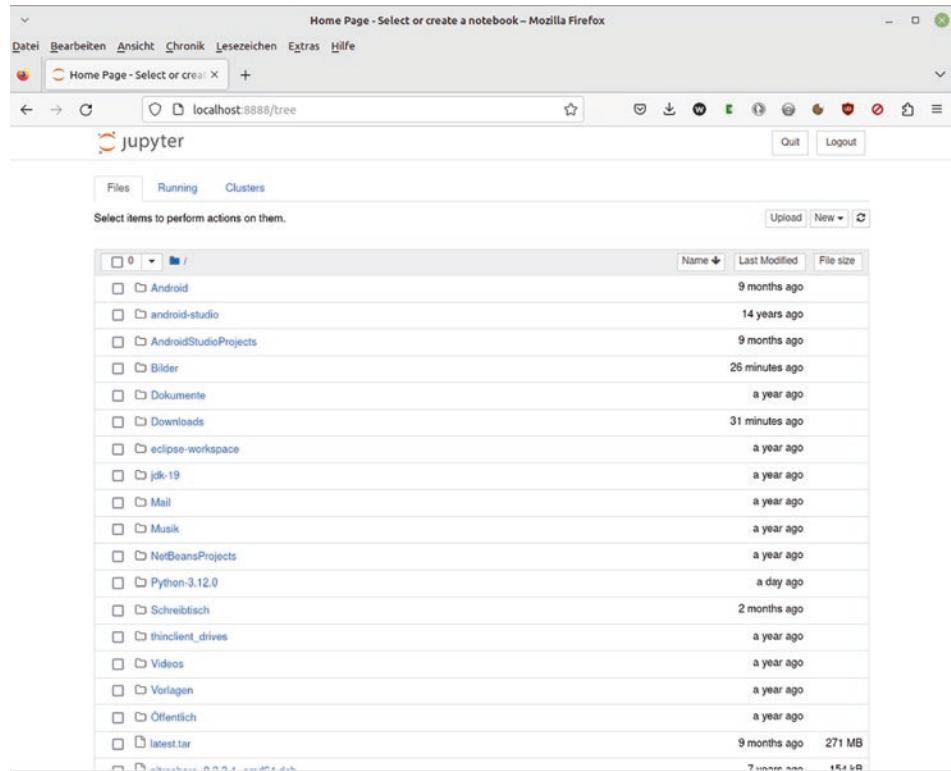


Abb. 2.21 Das Notebook mit Zugriff auf den lokalen Server unter Linux

Die Verwendung vom Jupyter Notebook erfolgt recht einfach mit Anaconda, da das Tool dort bereits integriert ist und sich über den Navigator starten lässt. Sollten Sie Anaconda nicht nutzen, kann man das reine Notebook z. B. mit pip installieren (*pip install jupyter*). Verschiedene Betriebssysteme bringen das Jupyter Notebook auch bereits in ihrer Paketverwaltung mit (Abb. 2.20).

Nach einer Installation muss der Server noch gestartet werden (*jupyter notebook* oder direkt mit einem Button), und dann ist das Notebook mit dem Browser lokal unter dem Defaultport 8888 zu erreichen (Abb. 2.21).

Etwa so:

```
http://localhost:8888
```

Will man den Jupyter-Server von einem anderen Rechner aus nutzen, ist das recht gut über einen SSH-Tunnel möglich. Das folgende Kommando öffnet beispielsweise einen Tunnel samt Port-Binding von einem lokalen Rechner zum Rechner mit der IP-Adresse 192.168.188.70, auf dem der Jupyter-Server und natürlich ein SSH-Server laufen.

```
ssh -L 8000:localhost:8888 ralph@192.168.188.70
```

Nach dem Login per SSH startet man den Jupyter-Server wie oben geschildert auf dem entfernten Rechner (sofern dort noch nicht sowieso gestartet) und kann dann auf dem lokalen Rechner unter der URL localhost:8000 auf das entfernte Notebook zugreifen (achten Sie die hier verwendete Port-Bindung zwischen dem Port 8000 auf dem lokalen und Port 8888 auf dem entfernten Rechner).

Beim ersten Zugriff auf das entfernte Notebook werden aus Sicherheitsgründen in der Regel ein Passwort oder ein spezifischer Token benötigt, das man in der Serverausgabe findet (Abb. 2.22).

Nach Eingabe der Verifizierung greift man mit dem Browser auf den entfernten Server und das dort gestartete Notebook analog zu wie im lokalen Fall oder auch bei der Verwendung einer Cloud (Abb. 2.23).

Das Jupyter-Notebook ist äußerst interessant und bietet weitreichende Möglichkeiten (etwa das bereits angesprochene Aufteilen von Code in Blöcke). Aber weitere Details sollen an dieser Stelle nicht ausgeführt werden, denn das würde zu weit führen und ist für die folgenden Schritte mit Python nicht notwendig.

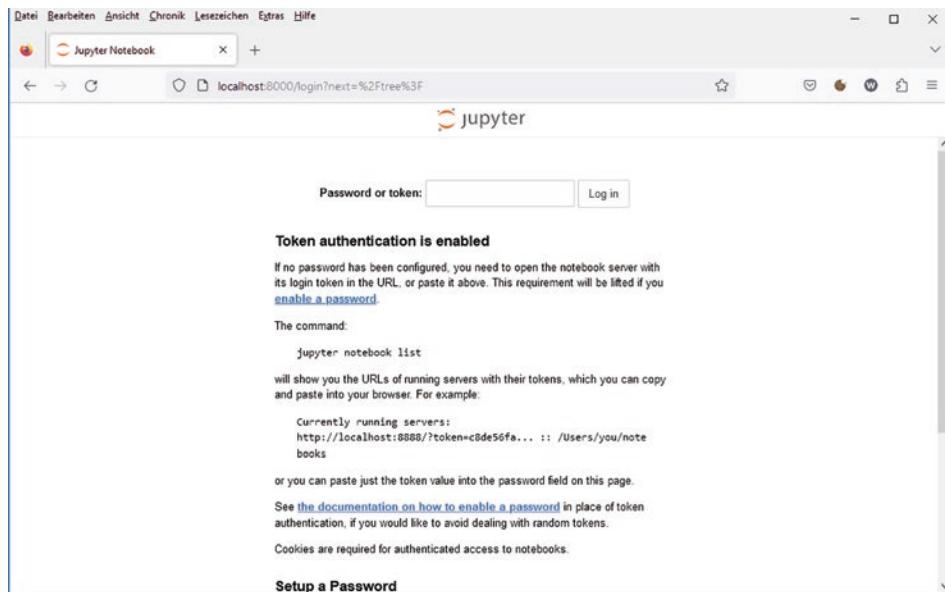


Abb. 2.22 Der Zugriff auf das entfernte Notebook benötigt einen Token oder ein Passwort

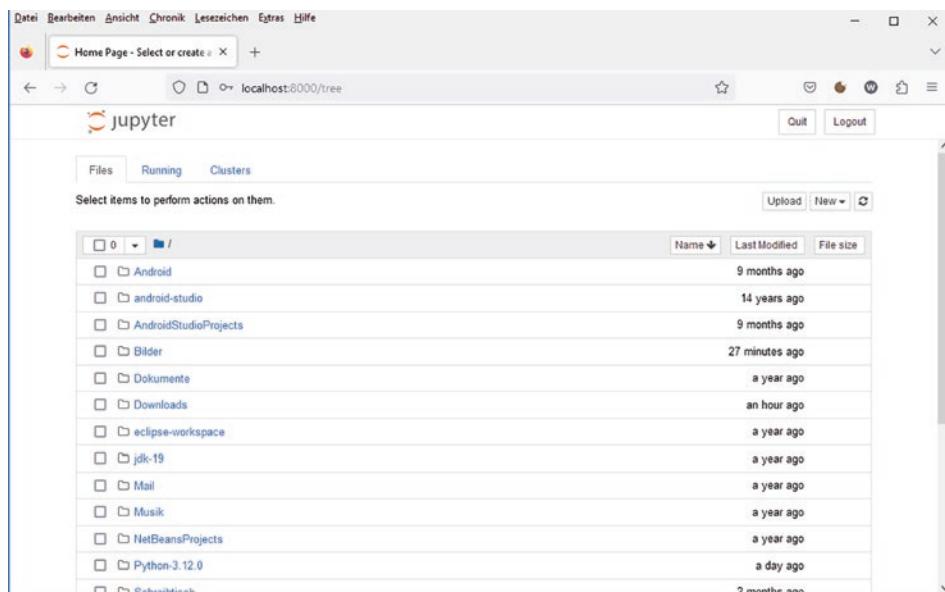


Abb. 2.23 Zugriff auf den Server unter Linux aus Windows via SSH und Port-Binding



Built-in Functions – Modularisierung durch Unterprogramme

3

3.1 Was behandeln wir in diesem Kapitel?

Für nahezu alle sinnvollen Schritte in Python benötigen Sie zumindest ein paar Funktionen. Denn alleine um das EVA-Prinzip (Eingabe-Verarbeitung-Ausgabe) auf einfachste Art umsetzen zu können, benötigen Sie in der Regel für zwei der Schritte (Eingabe und Ausgabe) solche vorgegebenen Funktionen. In diesem Kapitel schauen wir uns allgemein die sogenannten Built-in-Funktionen in Python und speziell die Funktionen zur Ein- und Ausgabe etwas genauer an.

3.2 Was sind Funktionen im Allgemeinen?

Sie werden in Python über kurz oder lang eigene Funktionen schreiben. Aber das wollen wir an der Stelle noch gar nicht angehen. Wir wollen aber auf einem ganz einfachen Niveau klären, was unter dem Begriff „Funktion“ überhaupt zu verstehen ist.

Sie können in Python (oder einer anderen Programmiersprache) im Grunde ein komplettes Programm oder Skript so erstellen, dass Sie im Quellcode nacheinander jede auszuführende Anweisung notieren. Aber das hat eine Reihe von Nachteilen.

- Es fehlt eine Wiederverwendbarkeit von schon vorhandenen Anweisungen.
- Der Quellcode wird unnötig groß (durch Redundanzen von allen möglichen Dingen).
- Man kann kaum wartbar und nachvollziehbar von einer sequentiellen Abarbeitung abweichen.
- Die Performance wird schlecht.
- Der Speicherbedarf ist unnötig hoch.

Und das ist nur ein Teil der Nachteile. Bei umfangreicheren Programmen muss und wird man in der Regel also eine **Modularisierung** vornehmen. Das erfolgt in einer Art **Unterprogramm**, das immer wieder gemeinsam auszuführende Anweisungen zusammenfasst.

Die Einführung eines solchen Unterprogramms im Quellcode nennt man **Deklaration**. Dabei werden die Anweisungen des Unterprogramms bei der Abarbeitung dieser Stelle im Quellcode durch den Interpreter **nicht** ausgeführt. Aber der Interpreter „merkt“ sich den Code oder zumindest, wo er ihn finden kann. Man kann jedoch nicht nur Funktionen deklarieren, sondern auch andere „Dinge“, etwa Variablen oder Klassen. Der Fachbegriff „Deklaration“ ist also allgemein zu sehen und steht für eine erste Einführung.

Ein Unterprogramm kann man im Quellcode dann über einen Namen aufrufen. Eine Funktion ist so ein Unterprogramm.

- ▶ Beachten Sie schon jetzt, dass in Python beim Aufruf einer Funktion immer ein Paar runder Klammern dem Funktionsnamen nachgestellt wird. Zwar gibt es einige Sondersituationen, in denen das unterbleiben kann, aber diese auszunutzen sollten Sie unbedingt vermeiden.

3.3 Built-in-Funktionen

Python besitzt eine Reihe an sogenannten vorinstallierten Funktionen (Built-in Functions), die automatisch in Python bereitstehen und überall im Quellcode verwendet werden können. Mit anderen Worten: Sie brauchen diese vor einer Verwendung nicht selbst zu deklarieren und auch sonst keine besonderen Vorbereitungen vor der Verwendung zu treffen.

3.3.1 Hilfe zu Built-in Functions im Hilfemodus

Sie erhalten im Hilfesystem von Python eine Hilfestellung zu diesen vorinstallierten Funktionen, indem Sie einfach deren Namen eingeben. Schauen wir uns die Vorgehensweise mit einem Beispiel an.

- Gehen Sie in den Hilfemodus von Python. Dazu geben Sie wieder *help()* in der Kommandozeile ein.
- Geben Sie in der Kommandozeile hinter dem Prompt des Hilfemodus die folgende Anweisung ein:

Beispiel

print ◀

The screenshot shows the Python help() output for the built-in function print(). It includes introductory text about Python modules, keywords, and topics, followed by detailed documentation for print(). The documentation lists parameters: *args, sep=' ', end='\n', file=None, flush=False, with descriptions for each. The help prompt 'help>' is visible at the bottom.

```
If this is your first time using Python, you should definitely check out
the tutorial on the internet at https://docs.python.org/3.12/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules. To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics". Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> print
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
        string inserted between values, default a space.
    end
        string appended after the last value, default a newline.
    file
        a file-like object (stream); defaults to the current sys.stdout.
    flush
        whether to forcibly flush the stream.

help>
```

Abb. 3.1 Hilfe zur print()-Funktion

- Bestätigen Sie die Eingabe.

Sie erhalten danach Informationen aus der Dokumentation zur Syntax von *print()* (Abb. 3.1). Das ist eine Ausgabefunktion, die wir gleich genauer betrachten werden.

Genau genommen liefert *print()* einen sogenannten **Ausgabestrom**.

3.3.2 Hilfe zu Built-in Functions im Editormodus

Sie brauchen nicht unbedingt explizit in den Hilfemodus zu wechseln, um in der Kommandozeile Hilfe zu erhalten. Sie können ebenso bei der Anweisung *help()* in den Klammern einen Suchbegriff als Parameter eingeben, zu dem Ihnen dann direkt Informationen im „normalen“ Editier- beziehungsweise Befehlsmodus angezeigt werden.

- Geben Sie in der Python-Kommandozeile hinter dem Prompt die folgende Anweisung ein:

Beispiel

help(input) ◀

- Bestätigen Sie die Eingabe.

```
>>>help(input)
Help on built-in function input in module builtins:

input(prompt='', /)
    Read a string from standard input. The trailing newline is stripped.

    The prompt string, if given, is printed to standard output without a
    trailing newline before reading input.

    If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return), raise EOFError.
    On *nix systems, readline is used if available.

>>>
```

Abb. 3.2 Hilfe aus dem normalen Eingabemodus

Abb. 3.3 Hilfe zu
Schlüsselworten erhalten Sie
nur im expliziten Hilfemodus

```
>>> help(if)
SyntaxError: invalid syntax
>>> |
```

Sie haben nun die analogen Informationen erhalten, wie Sie diese eben zur *print()*-Funktion im expliziten Hilfemodus erhalten haben (Abb. 3.2), nur dieses Mal zur Funktion *input()* – einer Eingabefunktion, die wir auch gleich genauer betrachten.

- ▶ Etwas „unschön“ in der Python-Kommandozeile ist, dass diese Hilfe aus dem normalen Editormodus bei Python-Schlüsselworten **nicht** funktioniert. Wenn Sie etwa zum Schlüsselwort *if* so Hilfe erhalten wollten, bekommen Sie einen Fehler (Abb. 3.3). Um dazu Hilfe zu erhalten, müssen Sie erst in den expliziten Hilfemodus wechseln.

3.3.3 Die `print()`-Funktion

Eine der wichtigsten Python-Funktionen beziehungsweise -Anweisungen ist sicher *print()*. Damit generiert man eine Ausgabe in der Konsole und kann den letzten Teil des EVA-Prinzips umsetzen. Dieser Funktion widmen wir uns jetzt als erster Python-Built-in Function.

Da es sich um eine Funktion handelt, sind bei der Notation des Aufrufs immer eine öffnende und eine schließende runde Klammer notwendig. Innerhalb der Klammern schreibt man dann die Werte hinein, die man ausgeben möchte. Dabei kann man kommassepariert auch mehrere Werte notieren, die man zusammen ausgegeben haben möchte.

- ▶ Die in die Klammern notierten Werte sind die sogenannten **Parameter**, **Argumente** oder **Übergabewerte** der Funktion. Beachten Sie, dass diese Begriffe streng genommen nicht ganz identisch sind. Aber Sie werden „umgangssprachlich“ im IT-Umfeld synonym eingesetzt.

Abb. 3.4 Anlegen einer neuen Quelltextdatei

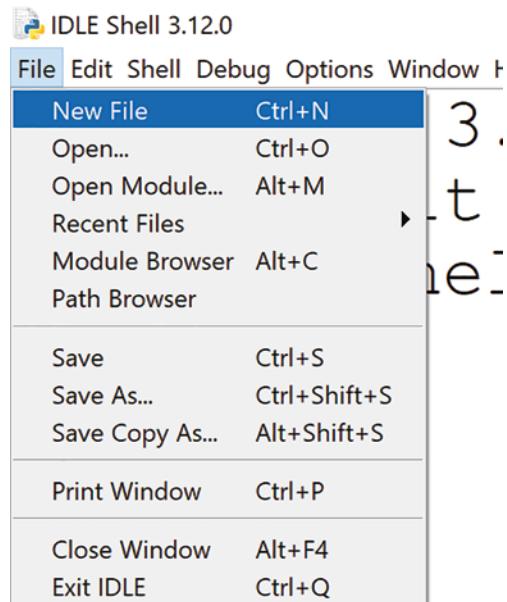


Abb. 3.5 Der Quellcode im integrierten Editor von IDLE

```
Untitled
File Edit Format Run Options Window Help
print("Das ist ein Python-Quelltext");
```

Nutzen wir die `print()`-Funktion nun in einem Python-Programm. Wir wollen zuerst dafür sorgen, dass das Beispiel aus dem letzten Kapitel mit der reinen Notation eines Stringliterals (ein Wert in Form eines Textes) den dort deklarierten Text auch ausgibt.

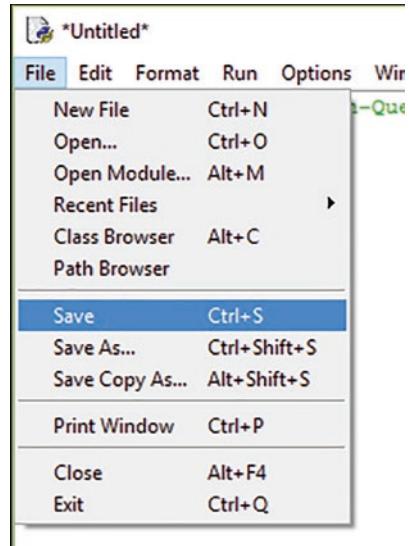
- Starten Sie IDLE oder eine andere IDE bzw. einen geeigneten Editor (die folgenden Ausführungen werden mit IDLE fortgesetzt).
- Wählen Sie das Menü FILE und dann NEW FILE (Abb. 3.4).
- Sie gelangen zum integrierten Python-Editor von IDLE. Geben Sie dort die folgende Anweisung ein (Abb. 3.5):

```
print("Das ist ein Python-Quelltext")
```

Beachten Sie im Editor die farbliche Hervorhebung der verschiedenen Python-Strukturen (Abb. 3.5). Das ist die schon erwähnte Syntaxhervorhebung (**syntax highlighting**).

Achten Sie auch wieder darauf, dass kein Leerzeichen vor der `print()`-Anweisung steht.

Abb. 3.6 Verschiedene Varianten zum Speichern



Nun sollte Ihnen auffallen, dass in der Titelzeile des Editors noch kein konkreter Dateiname steht. Das kann auch nicht sein, denn wir haben bis dato noch keinen Dateinamen vergeben. Der Stern hinter „Untitled“ zeigt zudem an, dass der aktuelle Stand des Quelltextes noch nicht gespeichert wurde (Abb. 3.5).

Um nun den Dateinamen festzulegen und den Quellcode zu speichern, können wir auf verschiedene Weise vorgehen.

1. Speichern der Datei über das FILE-Menü und dann einen der Befehle SAVE, SAVE As oder SAVE COPY As (Abb. 3.6) verwenden (oder eine der üblichen Tastenkombinationen dafür). Die Bedeutung und Unterschiede der einzelnen Befehle sollten bekannt oder nachvollziehbar sein.
 2. Auswahl RUN und dann RUN MODULE oder die Taste F5 (Abb. 3.7). Damit führen Sie den Python-Code direkt aus der IDLE-IDE heraus aus. Aber vorher muss dieser mit einem Dateinamen versehen und gespeichert sein. Sie erhalten also einen Dialog, über den Sie entscheiden können, ob Sie den aktuellen Stand¹ des Quellcodes speichern wollen (Abb. 3.8). Und wenn noch kein Dateiname vergeben wurde, erhalten Sie den üblichen Speichern-Dialog des Betriebssystems.
- Speichern Sie die Datei unter *print1.py*.

¹ Sollte bereits ein Dateiname vorhanden sein, können Sie auch einen älteren Stand des Quellcodes (der zuletzt gespeicherte) ausführen. Aber das ist selten sinnvoll.

Abb. 3.7 Den Python-Code ausführen

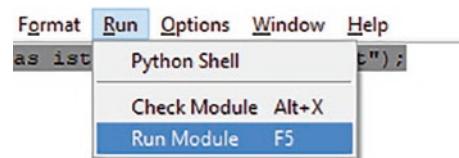


Abb. 3.8 Vor dem Ausführen speichern

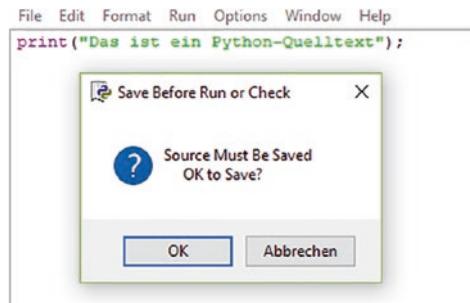
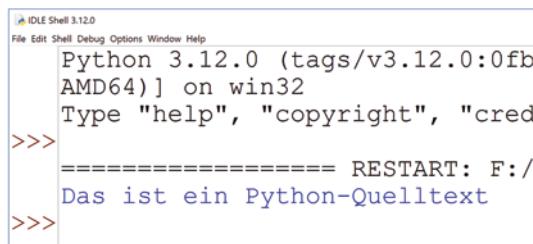


Abb. 3.9 Die Ausgabe des Programms



Sie sehen danach die Ausgabe des Texts in IDLE – genau genommen der integrierten Python-Shell (Abb. 3.9).

- Beachten Sie das Semikolon am Ende der Anweisung. Python erzwingt in der Regel kein Semikolon, um eine Anweisung zu beenden.² Aber es ist auch nicht verboten. Deshalb notieren es manche Python-Programmierer am Ende jeder Anweisung. Man sollte sich auf jeden Fall auf einen Stil festlegen. Entweder setzen Sie grundsätzlich ein Semikolon am Ende einer Anweisung, oder Sie lassen es immer weg, wenn nicht eine spezielle Situation dazu zwingt.

²Wie etwa Java, C, C++, C# und viele andere Sprachen.

Abb. 3.10 Zwei *print*-Anweisungen

```
===== RESTART: F:/PythonQu...
Das ist ein Python-Quelltext
Hier folgt die 2 Python-Anweisung
>>>
```

Erstellen wir auf die beschriebene Weise noch ein neues Programm mit Namen *print2.py*. Das soll folgenden Quelltext enthalten:

```
print("Das ist ein Python-Quelltext");
print("Hier", "folgt", "die", 2, "Python-Anweisung");
```

Es gibt an dem kleinen Programm bereits einige interessante Stellen, an denen man einiges zu Python lernen kann (Abb. 3.10).

- Es wurden zwei *print()*-Anweisungen angewendet. Diese wurden durch ein Semikolon getrennt und nacheinander vom Python-Interpreter abgearbeitet.
 - Ein Aufruf der *print()*-Funktion erzeugt einen Zeilenvorschub.
 - Die zweite Anweisung hat mehrere Parameter beim Aufruf der *print()*-Funktion verwendet.
 - Die Parameter bei der zweiten *print()*-Funktion werden zur Ausgabe verknüpft und der Text gemeinsam ausgegeben.
 - Die Parameter der zweiten *print()*-Funktion haben verschiedene Datentypen. Auch wenn dieser Begriff „Datentyp“ noch nicht eingeführt wurde, kann man ohne große Erklärung erkennen, dass es bei der zweiten Anweisung Stringliterale (in Hochkommata eingeschlossen) und Zahlen (Integerliterale) gibt, die gemeinsam verwendet werden. Das ist in Python oft möglich, denn Python ist lose typisiert. Was das bedeutet, wird jedoch noch detailliert behandelt.
- Es wurde schon erwähnt, ist aber so wichtig, dass noch mehrfach darauf eingegangen wird – Python beachtet Groß- und Kleinschreibung. Man nennt dies **case-sensitiv**. Das gilt auch für die Built-in-Funktionen, und deshalb wäre die Schreibweise *Print()* oder *PRINT()* falsch (Abb. 3.11)! Merken Sie sich, dass Built-in-Funktionen immer kleingeschrieben werden, und in der Regel auch alle anderen Anweisungen in Python.

3.3.3.1 Platzhalter in Strings und die f-string-Syntax

In Python findet man bei Strings in der *print*-Funktion oft ein vorangestelltes *f* und im Inneren geschweifte Klammern mit Bezeichnern darin. Dabei beziehen sich die vorangestellten „*f*“ und die geschweiften Klammern in der *print*-Funktion auf die sogenannte **f-string-Syntax**. Der Präfix „*f*“ steht für „formatiert“, und die Strings selbst werden **formatierte Zeichenketten** genannt.

```
Traceback (most recent call last):
  File "F:/PythonQuellcodes/kap3/print2.py", line 2, in <module>
    Print("Hier","folgt","die",2,"Python-Anweisung");
NameError: name 'Print' is not defined
>>> |
```

Abb. 3.11 Selbst nur ein Großbuchstabe beim Bezeichner print sorgt für einen Fehler

Dieses Konzept ermöglicht es, Strings auf eine einfache und bequeme Weise zu erstellen, indem Werte, Variablen oder Ausdrücke in einen String eingefügt werden. Man vermeidet damit String-Verkettung, bei der Strings mittels des Plus-Operators verbunden werden und die teils etwas aufwendig ist. Insbesondere spart man sich bei der f-string-Syntax eine Typkonvertierung, sofern ein Wert kein String ist. Das führt im Moment noch zu weit, aber darauf kommen wir zurück.

Die Verwendung von f-Strings macht das Erstellen von formatierten Strings in Python viel leserlicher und einfacher, da man die Werte direkt in den Text einbetten kann. Dies ist besonders nützlich, wenn man viele Variablen oder Ausdrücke in einen String einfügen muss.

Doch zurück zu den Details der f-string-Syntax. Ein f-string ist ein String, der spezielle Platzhalter für Werte oder Ausdrücke enthält, die später in den String eingefügt werden. Diese Platzhalter sind von geschweiften Klammern umgeben, und innerhalb dieser Klammern kann man im Grunde beliebige Python-Ausdrücke verwenden.

Hier ist ein einfaches Beispiel, das etwas vorgreift und eine Variable verwendet (*f_strings.py*):

```
x = 1
print(f'Zahl: {x}')
```

► Eine **Variable** kann man sich erst einmal (vereinfacht) als eine Stelle im Hauptspeicher des Rechners vorstellen, die über einen Bezeichner (Namen) anzusprechen ist und wohin man Werte schreiben kann. Diese Werte können später wieder ausgelesen werden.

Mit *x* wird eine Variable bezeichnet. Deren Wert wird bei der *print*-Funktion im Rahmen des Textes ausgegeben, weil im String die Variable in den geschweiften Klammern steht.

3.3.3.2 Den Separator bei *print()* anpassen

Ihnen sollte aufgefallen sein, dass bei der Angabe von mehreren Parametern mit *print()* Python diese bei der Ausgabe automatisch durch ein Leerzeichen verbunden hat (Abb. 3.10). Dieses Leerzeichen ist ein Separator, der bei *print()* als Vorgabewert verwendet wird. Über den Optionsparameter *sep* (für Separator) kann dieser mit einer

Abb. 3.12 Anpassung des Separators der print()-Funktion

```
Das****ist***einPython-Quelltext
DasisteinPython-Quelltext
Das#ist#einPython-Quelltext
>>> |
```

Zuweisung über das Gleichheitszeichen angepasst oder auch weggelassen werden. Sie geben einfach das oder die gewünschte(n) Zeichen als weiteren Parameter zusätzlich an.

- ▶ Beachten Sie, dass diese Art der Wertzuweisung bei Attributen in Python sehr häufig für die Konfiguration verwendet wird. Dieses Beispiel mit dem Parameter *sep* kann also verallgemeinert betrachtet werden. Man hat in Python sehr oft ein Default- beziehungsweise Vorgabeverhalten, und wenn dieses modifiziert werden soll, dann wird einem bestimmten Attribut ein neuer Wert zugewiesen.

Erstellen wir auf die beschriebene Weise noch ein neues Programm mit Namen *print3.py*. Das soll folgenden Quelltext enthalten:

```
print("Das", "ist", "ein" "Python-Quelltext",sep="****");
print("Das", "ist", "ein" "Python-Quelltext",sep=" ");
print("Das", "ist", "ein" "Python-Quelltext",sep="#");
```

Sie sehen, dass bei der ersten Ausgabe die drei Sternzeichen zwischen den Wörtern auftauchen, bei der zweiten kein Leerraum und bei der dritten Ausgabe das Hash-Zeichen (Abb. 3.12).

3.3.4 Die `input()`-Funktion

Um das EVA-Prinzip vollständig umzusetzen, brauchen wir in Python noch eine Eingabemöglichkeit. Und dazu kann man eine weitere Built-in-Funktion anwenden, die sich *input()* nennt. Wir wollen nun ein Programm erstellen, in dem der Anwender eine Eingabe vornehmen kann, die dann in die Ausgabe eingearbeitet wird.

Erstellen wir auf die oben beschriebene Weise noch ein neues Programm mit Namen *input1.py*. Das soll folgenden Quelltext enthalten:

```
name=input("Geben Sie Ihren Namen an");
print("Hallo", name);
```

Python betrachtet wieder jede einzelne Zeile als Anweisung und arbeitet jede einzelne Zeile hintereinander sequenziell ab. Ein Anwender kann zuerst nach Aufforderung seinen Namen eingeben, dieser wird dann in die Ausgabe übernommen (Abb. 3.13). Beachten

Abb. 3.13 Entgegennahme einer Eingabe

```
Geben Sie Ihren Namen anHerby
Hallo Herby
>>>
```

Sie, dass die *input*-Funktion einen Parameter haben kann, der in der Konsole vor der Eingabe angezeigt wird.

Beachten Sie ebenso, dass bei dem Beispiel in der zweiten Zeile der zweite Parameter bei der *print()*-Funktion **nicht** als String notiert ist und nicht in Hochkommata eingeschlossen wird. Das ist dann ein Variablenname. Diese Variable wurde zuvor in der ersten Zeile deklariert.

Bei *input()* setzt man innerhalb der Klammern als Parameter eine Zeichenkette hinein, die beim Aufruf ausgegeben wird. Damit kann man den Benutzer auffordern, eine Zeichenkette einzugeben. Die Funktion unterbricht bis zur Bestätigung einer Eingabe durch den Anwender die Abarbeitung des Programms und liefert dann einen sogenannten Rückgabewert, der in dem Beispiel der Variablen *name* zugewiesen wird, welche wie gesagt in der ersten Zeile deklariert wurde.

► Der **Rückgabewert** ist das Ergebnis der Funktion. In Python muss aber nicht jede Funktion einen Rückgabewert liefern.

Der Rückgabewert ist in dem Fall der Text, den der Benutzer eingegeben hat, und der wird dann mit der nächsten Anweisung über die Verwendung der Variablen wieder ausgegeben.

Genau genommen liefert *input()* einen sogenannten **Eingabestrom**.

Nun sollte Ihnen bei der Ausgabe aufgefallen sein, dass die Eingabe des Anwenders ziemlich unschön an den Text „geklebt“ wird, der bei der *input()*-Funktion zur Eingabe auffordert. Die *input()*-Funktion erzeugt also im Gegensatz zur *print()*-Funktion keinen Zeilenumbruch. Wir wollen das Beispiel modifizieren (*input2.py*) und die Anzeige der Eingabe in eine neue Zeile verlagern. Der Quellcode sieht nun so aus:

```
name=input("Geben Sie Ihren Namen an\n");
print("Hallo", name);
```

Nun wird die Rückmeldung von Python doch schöner aussehen (Abb. 3.14). Aber haben Sie überhaupt die Modifikation bemerkt? Ich habe sie bewusst nicht hervorgehoben. Am Ende des Strings in Zeile 1 steht eine sogenannte **Escape-Sequenz**. Damit kann man Zeichen maskieren (in einem speziellen Code notieren). Das ist insbesondere für Sonderzeichen notwendig, und \n steht für einen Zeilenumbruch.

Abb. 3.14 Entgegennahme einer Eingabe mit „schönerer“ Formatierung

```
Geben Sie Ihren Namen an
Herby
Hallo Herby
>>> |
```

3.3.5 Eine kurze Übersicht aller Built-in Functions

Nachfolgend finden Sie alphabetisch sortiert eine Tabelle aller Built-in-Funktionen von Python (Tab. 3.1). Für Details sei auf die Dokumentation unter <https://docs.python.org/3/library/functions.html> verwiesen. Wir haben uns in dem Buch ja auf die Fahne geschrieben, nicht jedes Detail zu untersuchen, sondern immer den grundsätzlichen Ansatz.

Dennoch sollen ein paar weitere Beispiele für den Umgang mit den Built-in Functions folgen. Dabei greifen wir an einigen Stellen zwar etwas vor, aber die Listings sollten mit den Erklärungen gut nachvollziehbar sein. Wir deuten damit auch schon einmal ein paar Techniken an, die noch genauer ausgearbeitet werden.

3.3.5.1 Mathematische Umwandlungen

- Erstellen Sie den folgenden Code (*umwandeln.py*):

```
zahl=input("Geben Sie eine Zahl ein\n");
print("Oktal entspricht die dezimale Zahl", zahl, "dem Wert",
      oct(int(zahl)));
```

In dem Beispiel kommen die Built-in Functions *int()* und *oct()* zum Einsatz. Damit wandelt man als Parameter übergebene Werte um. Man nennt dieses Verfahren **Casting** oder **Typumwandlung**. Was steckt dahinter?

Nun, die Funktion *input()* liefert einen String als Datentyp. Der Rückgabewert der Funktion ist also ein Text. In dem Beispiel wird ein Anwender aufgefordert, eine Zahl einzugeben. Aber auch wenn er das macht, ist das rein formal ein Text. Angenommen, ein Anwender gibt nach der Aufforderung die Zahl **11** ein. Dann liefert *input()* aber "**11**". Das ist ein Text, dessen Inhalt nur (zufällig) aus einer Zahl besteht.

Um nun damit zu rechnen oder bestimmte mathematische Dinge zu tun, kann es notwendig sein, die Zahl aus dem Text „herauszuschälen“. Das macht man in Form dieses Castings. Die Built-in Function *int()* macht genau so etwas – aus dem Text wird eine Zahl „herausgeschält“, sofern das möglich ist. Diese Zahl (der Rückgabewert von *int()*) wird nun von der zweiten Funktion *oct()*, die um die Funktion *int()* gelegt wurde, in eine oktale Darstellung der Zahl umgewandelt (Abb. 3.15).

Tab. 3.1 Built-in Functions in Python

<code>abs()</code>	<code>divmod()</code>	<code>input()</code>	<code>open()</code>	<code>staticmethod()</code>
<code>all()</code>	<code>enumerate()</code>	<code>int()</code>	<code>ord()</code>	<code>str()</code>
<code>any()</code>	<code>eval()</code>	<code>isinstance()</code>	<code>pow()</code>	<code>sum()</code>
<code>basestring()</code>	<code>execfile()</code>	<code>issubclass()</code>	<code>print()</code>	<code>super()</code>
<code>bin()</code>	<code>file()</code>	<code>iter()</code>	<code>property()</code>	<code>tuple()</code>
<code>bool()</code>	<code>filter()</code>	<code>len()</code>	<code>range()</code>	<code>type()</code>
<code>bytearray()</code>	<code>float()</code>	<code>list()</code>	<code>raw_input()</code>	<code>unichr()</code>
<code>callable()</code>	<code>format()</code>	<code>locals()</code>	<code>reduce()</code>	<code>unicode()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>long()</code>	<code>reload()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>map()</code>	<code>repr()</code>	<code>xrange()</code>
<code>cmp()</code>	<code>globals()</code>	<code>max()</code>	<code>reversed()</code>	<code>zip()</code>
<code>compile()</code>	<code>hasattr()</code>	<code>memoryview()</code>	<code>round()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hash()</code>	<code>min()</code>	<code>set()</code>	
<code>delattr()</code>	<code>help()</code>	<code>next()</code>	<code>setattr()</code>	
<code>dict()</code>	<code>hex()</code>	<code>object()</code>	<code>slice()</code>	
<code>dir()</code>	<code>id()</code>	<code>oct()</code>	<code>sorted()</code>	

```
Geben Sie eine Zahl ein
11
Oktal entspricht die dezimale Zahl 11 dem Wert 0o13
>>>
```

Abb. 3.15 Mathematische Funktionen

► Zahlen können in verschiedenen Zahlensystemen dargestellt werden. Beim „normalen“ Rechnen verwendet man das Dezimalsystem, das bereits kleine Kinder lernen. In der IT werden aber auch oft Systeme wie das Dualsystem, Hexadezimalsystem oder eben Oktalsystem verwendet. Das Oktalsystem hat die Basis 8 und kennt nur acht Ziffern zur Darstellung einer Zahl: 0, 1, 2, 3, 4, 5, 6 und 7. Jede größere Zahl muss daraus zusammengesetzt werden.

Sie sehen an dem Beispiel bereits, dass man Aufrufe von Funktionen mit Rückgabewert an jeder Stelle notieren kann, an denen ein Wert (Literal) notiert werden kann, etwa als Parameter einer anderen Funktion.

Der gesamte Prozess wird allerdings zu einem Fehler führen, wenn der Anwender keine reine Zahl eingibt (Abb. 3.16). Die Eingabe ist also nicht gegen Fehleingaben abgesichert. Aber für die Demonstration der beiden Funktionen und die Typumwandlung genügt das Beispiel.

```
Geben Sie eine Zahl ein
42 Birnen
Traceback (most recent call last):
  File "F:/PythonQuellcodes/kap3/umwandeln.py", line 2, in <module>
    print("Oktal entspricht die dezimale Zahl", zahl, "dem Wert", oct(int(zahl))
);
ValueError: invalid literal for int() with base 10: '42 Birnen'
>>>
```

Abb. 3.16 Die Eingabe war keine reine Zahl

3.3.5.2 Evaluierung

Lassen Sie uns jetzt noch die Summe von mehreren Variablen und numerischen Literalen bilden. Das macht das folgende Programm *summe.py*:

```
z1=2;
z2=4;
z3=7;
print(eval("1+2+3+z1+z2+z3"));
```

Die Ausgabe von dem Programm wird **19** sein. Das ergibt sich einfach aus **1+2+3+2+4+7**. Aber die Sache ist dennoch sehr, sehr bemerkenswert. Denn der evaluierte Ausdruck ist ein Stringliteral. Im Inneren des Stringliterals befinden sich sogenannte Operatoren mit ihren Operanden. Das Zeichen + ist ein mathematischer Operator, den Sie intuitiv aus der Schulmathematik verstehen. Aber dass diese mathematische Operation der Addition überhaupt ausgeführt wird, ist nicht trivial. Die Werte wie auch die Operatoren stehen ja in einem String. Aber die Built-in Function *eval()* ist genau dafür gedacht. Strings werden damit bezüglich enthaltender Operationen evaluiert. Und auf noch etwas muss ausdrücklich die Beachtung gelenkt werden – die Operanden können auch Variablen sein. Genau genommen wird jeder gültige Python-Code damit ausgeführt, was die Funktion nicht ungefährlich macht, wenn freie Benutzereingaben damit evaluiert werden.

3.3.5.3 Absolutwertberechnung

Das nachfolgende kleine Listing (*absolut.py*) berechnet zum Abschluss der Übungen den Absolutwert einer Zahl. Dazu kommt einfach die Built-in-Funktion *abs()* zum Einsatz (Abb. 3.17):

```
zahl=-42;
print("Der Absolutwert von", zahl, "ist", abs(zahl));
```

Abb. 3.17 Berechnung des Absolutwerts

```
Der Absolutwert von -42 ist 42
>>>
```



Grundlegende Begriffe – Kommentare, SheBang und Strukturanalysen

4

4.1 Was behandeln wir in diesem Kapitel?

Bevor es nun genauer an die konkreten Details in Python geht, sollen in dem Kapitel einige wichtige Begriffe und Vorgänge in der Programmierung allgemein, aber natürlich an Python angepasst, geklärt werden, die zwar eher zum Hintergrundwissen denn zur Praxis zählen. Aber das Wissen darum macht deutlich, wie gewisse Dinge ablaufen.

4.2 Token und Parser

Token bedeutet übersetzt Zeichen oder Merkmal. Wenn ein Python-Interpreter eine `.py`-Datei in Maschinenanweisungen übersetzen will, muss er vorher zunächst herausfinden (oder mitgeteilt bekommen), welche Token oder Symbole im Code benutzt werden und sie in sinnvolle Zusammenhänge setzen.

Einen Token kann man deshalb auch als **Sinnzusammenhang** verstehen. So ist etwa in der menschlichen Sprache ein Wort nicht nur die Summe seiner Buchstaben oder Zeichen beziehungsweise akustisch ein Klang, sondern besitzt einen konkreten Sinnzusammenhang, den das interpretierende System (der menschliche Geist) mit einer bestimmten Bedeutung assoziiert. Allerdings muss das interpretierende System sowohl die Sprache als auch den konkreten Begriff verstehen, sonst bleibt ein Token einfach nur die Summe seiner Buchstaben oder Zeichen beziehungsweise im akustischen Fall ein Lautgebilde ohne Bedeutung. Stellen Sie sich vor, dass Sie ein Wort in einer Ihnen fremden Sprache hören. Für Sie ist das dann kein Token, aber für jemanden, der diese Sprache

versteht, schon. Und wenn Sie sich das Wort übersetzen lassen, dann wird es auch für Sie in Zukunft ein Token sein.

4.2.1 Zerlegen von Quelltext

Der gesamte Teil der Assoziation mit gültigen Token bei einer Programmiersprache wird bei der Übersetzung eines Quelltextes vom sogenannten **Parser** übernommen. Dieser kann ein eigenständiges Programm sein, aber oft ist er auch in einen Interpreter oder Compiler bereits integriert. Allgemein versteht man in der Informatik unter einem Parser (engl. *to parse*, analysieren) ein Computerprogramm (oder einen Teil eines Programms), das für die Zerlegung und Umwandlung einer beliebigen Eingabe in ein für die Weiterverarbeitung brauchbares Format zuständig ist.

Wenn der Quellcode vom Python-Interpreter übersetzt werden soll, muss er ihn also vorher von seinem zugeordneten Parser in einzelne sinnvolle Bestandteile (eben Token) zerlegen lassen. Dabei wird oft auch die Reihenfolge der Token geändert. Nur dann kann der Quellcode übersetzt werden.

Der gesamte Quelltext muss sich also in logisch sinnvolle Einheiten zerlegen und in gültige Arten von Python-Token einordnen lassen. Die Sprachelemente werden dabei auf ihre Richtigkeit geprüft. Alle Operationen mit einem Token werden mit dem spezifischen Wert des Tokens durchgeführt, also nicht mit den Buchstaben und Zeichen des Tokens selbst, sondern mit dem, was Python mit dem Token assoziiert. Es gibt in Python wie in den meisten anderen Computersprachen fünf Arten von Token:

1. Bezeichner oder Identifier
2. Schlüsselwörter
3. Literale
4. Operatoren
5. Trennzeichen

4.2.1.1 Leerzeichen und Trennzeichen

Beginnen wir die Erklärungen von hinten. Denn die da genannten **Trennzeichen** als auch **Leerzeichen** sind wichtige Möglichkeiten, um Token zu trennen. Das ist an vielen Stellen notwendig, kann aber auch an anderen Stellen rein zur Übersicht des Quellcodes verwendet werden.

Leerzeichen sind in den meisten Programmiersprachen wie Kommentare (Abschn. 1.3) zu sehen¹ und keine klassische Token, obwohl sie auf Quelltextebene wie Token notiert werden. In Python ist das etwas anders, zumindest, wenn Leerzeichen am

¹ Haben also außer einer Trennung von Token keine „Bedeutung“.

Anfang einer Quellcodezeile beziehungsweise vor einem Ausdruck stehen – was wir schon gesehen haben. Denn Einrückungen (also Leerzeichen am Beginn der Zeile) haben eine definierte Bedeutung.

Unter **Trennzeichen** versteht man neben Leerzeichen alle einzelnen Symbole, die dazu benutzt werden, andere Token zu trennen und Zusammenfassungen von Code anzuzeigen. Dabei sind Trennzeichen gar nicht so trivial.

- Es gibt in Python etwa das **Komma**, das in mehreren Zusammenhängen als Begrenzer verwendet wird.
- Der **Punkt** wird zum einen als **Dezimalpunkt**, zum anderen über die Punktnotation als Trennzeichen in der OOP benutzt.
- Die **runde Klammer** wird sowohl für eine Parameterliste einer Methode oder Funktion als auch zur Festlegung eines Vorrangs für Operationen in einem Ausdruck benutzt.
- Die **eckigen Klammern** werden in Python bei Datenstrukturen verwendet.

4.2.1.2 Operatoren

Operatoren sind Token, die mit ihren Operanden Operationen ausführen, etwa mathematische Berechnungen, Wertzuweisungen, Vergleiche etc. Wir gehen detailliert auf die Operatoren unter Python noch ein.

4.2.1.3 Literale

Literale sind Werte, was schon kurz angedeutet wurde. Wir sind dem Begriff bereits mehrfach begegnet (Stringliterale und Integerliterale). Aber man muss sich klarmachen, dass es auch explizit Token (sinnbehaftete Ausdrücke) sind.

4.2.1.4 Schlüsselwörter

Unter dem Begriff **Schlüsselwörter** fasst man alle Wörter zusammen, die ein essenzieller Teil der Sprachdefinition sind und die in Python eine besondere Bedeutung haben. Auch die Behandlung von Schlüsselwörtern wird auf verschiedene Abschnitte des Buches verteilt und an den passenden Stellen genauer abgehandelt.

4.2.1.5 Bezeichner

Mit dem Token-Typ **Bezeichner** oder **Identifier** berühren wir ein Thema, das auf verschiedene Abschnitte des Buches verteilt wird. Es handelt sich hier um die Namen für Klassen, Objekte, Variablen, Funktionen, Konstanten etc. Wir gehen an den passenden Stellen genauer darauf ein.

4.3 Kommentare

Bei der Zerlegung von Quellcode in Token werden **Kommentare** aus dem Text entfernt. Kommentare sind also keine Token im eigentlichen Sinn. Sie passen dennoch in das Gesamtkonzept der Token, weil sie auf Quelltextebene wie Token notiert werden. Und zudem kann ein Kommentar zwar für das eine System kein Token sein, für ein zweites System jedoch schon.

Das klingt abstrakt, aber denken Sie noch einmal an die Situation, dass Sie ein Wort in einer Ihnen fremden Sprache hören. Für Sie ist das dann kein Token. Jetzt wird aus Ihnen der Python-Interpreter. Kommentare sind für ihn keine Token und werden von dem zugeordneten Parser entfernt. Aber wenn nun ein anderes interpretierendes System den Quellcode ansieht, kann dieses diese Zeichenkombination schon als Token verstehen, etwa ein Tool, das aus dem Quellcode eine Dokumentation erstellt.

Doch wie kann man nur einen Kommentar definieren?

- ▶ Ein **Kommentar** ist eine Zeichenfolge im Quellcode, die die Ausführung eines Programms nicht beeinflusst, aber auch keinen Fehler auslöst. Ausnahme ist die **SheBang**, die einen besonderen Kommentar darstellt, der unter Unix-artigen Systemen eine Bedeutung hat (Abschn. 1.4).

4.3.1 Kommentare in Python

In Python-Quellcode kann man eine Zeile als Kommentar kennzeichnen und dafür sorgen, dass der Python-Interpreter eine Zeile überspringt, indem ganz am Anfang der Zeile eine Raute gesetzt wird. Dann wird vom Interpreter alles, was dahintersteht, ignoriert.

Das ist dann ein **einzeiliger** Kommentar.

- ▶ Das Kommentarzeichen kann aber auch hinter Anweisungen notiert werden und damit eine Anweisung davor einfach kommentieren. Die Zeichen davor sind aber normaler Programmcode.

Für **mehrzeilige** Kommentare setzt man drei einfache oder auch doppelte Hochkommata. Diese müssen dann aber auch wieder abgeschlossen werden.

Kommentare sind also in Python wirklich nicht schwer, aber ein Beispiel soll die Thematik dennoch praktisch verdeutlichen.

- Erstellen Sie das Python-Programm *mitKommentar.py* mit folgendem Quelltext:

```
# Eingabe der Anrede
anr=input("Geben Sie die Anrede an\n");
# Eingabe des Vornamens
vn=input("Geben Sie den Vornamen an\n");
# Eingabe des Nachnamens
nn=input("Geben Sie den Nachnamen an\n");
"""

Und jetzt erfolgt die Ausgabe.
Zuerst das Wort "Hallo", dann der Vorname und
zuletzt der Nachname
"""

print("Hallo", anr, vn, nn); # Das ist die Ausgabe
```

- ▶ Beachten Sie im Editor die farbliche Abgrenzung von Kommentaren (Abb. 4.1). Das Syntax-Highlighting macht deutlich, dass Kommentare eine besondere Struktur sind.
- Führen Sie das Programm direkt aus IDLE aus, indem Sie auf den RUN-Befehl im Menü gehen und dann RUN MODULE auswählen (Abb. 4.2). Natürlich können Sie auch eine andere Entwicklungsumgebung nutzen.

Die erste, dritte und fünfte Zeile im Quelltext zeigen einen einzeiligen Kommentar, während in der siebten Zeile der Beginn und in der elften Zeile das Ende des mehrzeiligen Kommentars markiert werden. Alles von Zeile 7 bis Zeile 11 zählt zum mehrzeiligen Kommentar. Hinter der `print`-Anweisung steht noch ein weiterer einzeiliger Kommentar direkt in der Zeile.

Die mehrzeiligen Kommentare tauchen in der Hilfekonsole von Python auf, wenn Sie mit `help()` etwa Hilfe zu einer Klasse aufrufen und diese entsprechend dokumentiert haben.

The screenshot shows the Python IDLE editor interface. At the top is a menu bar with File, Edit, Format, Run, Options, Window, and Help. Below the menu is the Python source code. The code uses color-coded syntax highlighting: red for comments, green for strings, blue for keywords like print, and black for variables and operators. The code itself is identical to the one shown in the text above, including the multi-line comment block and the final single-line comment after the print statement.

```
File Edit Format Run Options Window Help
# Eingabe der Anrede
anr=input("Geben Sie die Anrede an\n");
# Eingabe des Vornamens
vn=input("Geben Sie den Vornamen an\n");
# Eingabe des Nachnamens
nn=input("Geben Sie den Nachnamen an\n");
"""

Und jetzt erfolgt die Ausgabe.
Zuerst das Wort "Hallo", dann der Vorname und
zuletzt der Nachname
"""

print("Hallo", anr, vn, nn); # Das ist die Ausgabe
```

Abb. 4.1 Der Quellcode mit den Kommentaren

Abb. 4.2 Das Programm wurde ausgeführt

```
Type "copyright", "credits" or
>>>
=====
RESTART: F:\Python\...
Geben Sie die Anrede an
Herr
Geben Sie den Vornamen an
Rudi
Geben Sie den Nachnamen an
Rueffel
Hallo Herr Rudi Rueffel
>>> |
```

4.4 SheBang und eine Python-Datei direkt ausführen

Wenn Python auf einem Computer korrekt installiert ist, können Sie eine Python-Datei direkt ausführen. Unter einem Windows-Rechner brauchen Sie bloß einen Doppelklick auf die `.py`-Datei auszuführen. Unter Umständen müssen Sie noch Python als „App“ auswählen, aber grundsätzlich geht das. Beachten Sie aber, dass eine Konsolenanwendung so ausgeführt wird, dass das Terminal unmittelbar nach Ende des Programms wieder geschlossen wird.

- ▶ Wenn Sie als letzte Anweisung in dem Python-Quellcode einfach nur zusätzlich `input()` notieren, wird das Terminal offen gehalten, bis Sie die Eingabeaufforderung betätigen.

Bei macOS führen Sie die Datei ganz einfach mit einem Python-Launcher aus.

4.4.1 SheBang als besonderer Kommentar

Unter Linux sollte als erste Zeile im Quellcode allerdings eine sogenannte SheBang notiert werden. Die sieht etwa so aus:

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

Das gibt bei einem Unix-artigen Betriebssystem an, wo der Python-Interpreter zu finden ist. Dazu muss aber die Python-Datei zusätzlich als **executable** (ausführbar) markiert werden. In anderen Betriebssystemen wird diese SheBang-Zeile als normaler Kommentar verstanden und einfach ignoriert. Wir haben hier also einen Fall, in dem eine bestimmte Kommentarform auf einem Betriebssystem als Token verstanden wird, auf anderen aber nicht.



Anweisungen – Dem Computer Befehle geben

5

5.1 Was behandeln wir in diesem Kapitel?

Anweisungen bezeichnen beim Programmieren allgemein eine Syntaxstruktur, die bestimmte Dinge ausführt. Man unterteilt Anweisungen in der Programmierung grundsätzlich nach ihrer Art, und in diesem kompakten Kapitel wollen wir uns einen kleinen Überblick über diese verschiedenen Arten verschaffen.

5.2 Was sind Anweisungen?

Wenn Sie einem Computer Befehle geben wollen, schreiben Sie Anweisungen. Damit werden bestimmte Dinge ausgeführt. Wir haben in sämtlichen Codebeispielen bisher schon mit Anweisungen gearbeitet, obgleich das so nicht ausdrücklich thematisiert wurde.

5.2.1 Eine Frage der Reihenfolge

Anweisungen werden in der Regel einfach der Reihe nach ausgeführt. Man nennt das eine sequenzielle Abarbeitung und die Folge selbst eine **Sequenz**. Betrachten Sie den Quellcode, den wir in der Art schon kennen:

```
anr=input("Geben Sie die Anrede an\n");
vn=input("Geben Sie den Vornamen an\n");
```

```
nn=input("Geben Sie den Nachnamen an\n");
print("Hallo", anr, vn, nn);
```

Nacheinander werden die Anweisungen in dem Quellcode abgearbeitet.

Aber es gibt auch Anweisungen, die nicht der Reihe nach abgearbeitet werden, sondern aufgrund einer bestimmten Situation.

- Die Abarbeitung der Anweisungen nennt man den **Programmfluss**.

So eine Situation, in der der Programmfluss von der sequenziellen Abarbeitung abweicht, liegt bei Kontrollflussanweisungen oder Ausnahmeanweisungen vor. Darauf gehen wir detailliert noch ein.

5.3 Anweisungsarten

Man unterteilt Anweisungen in der Programmierung grundsätzlich nach ihrer Art.

5.3.1 Blockanweisung

Blockanweisungen erlauben eine Zusammenfassung von größeren Teilen des Quellcodes zu Blockstrukturen. Außerdem werden Blockanweisungen dazu genutzt, den Geltungsbereich von Variablen und Funktionen einzuschränken.

Eine Sprache, die die Strukturierung mit Blockanweisungen ermöglicht, wird als **strukturierte** Programmiersprache bezeichnet.

In einem Anweisungsblock steht eine Reihe von anderen Anweisungen. Anstelle einer einzelnen Anweisung kann immer jeder sinnvolle Block stehen.

- Blöcke können beliebig ineinander geschachtelt werden. Sie müssen allerdings die Schachtelung wieder sauber schließen. Viele Fehler beruhen auf geöffneten und nicht korrekt geschlossenen Blöcken.
- In Python wird die Strukturierung in Blockanweisungen durch **Einrückung** von **vier Zeichen** (Vorgabewert) vorgenommen.
- Die Tiefe der Einrückung kann verändert werden. Sie muss nur innerhalb von jedem Block identisch sein. In IDLE finden Sie dazu die Anpassung in den Optionen. Aber in der Regel bleibt man bei der Einrücktiefe 4. Dies nutzen in der Regel alle Entwicklungsumgebungen zu Python als Vorgabewert.

Das Verwenden von Leerzeichen zur Bildung von Blockanweisungen unterscheidet das Strukturierungsprinzip von Python deutlich von vielen anderen Programmiersprachen. Wie bereits beschrieben, strukturieren andere Programmiersprachen ihre Programmblöcke durch Schlüsselwörter oder geschweifte Klammern. Leerzeichen oder Einrückungen sind für die Compiler und Interpreter der meisten Programmiersprachen ohne jede Bedeutung. Dennoch wird Programmierern aber immer empfohlen, Blöcke durch einheitliche Einrückungen besser verständlich zu machen.

In Python geht man diese Empfehlung konsequent weiter und gibt führenden Leerzeichen eine Bedeutung. Die Einrückung von Zeilen und die Benutzung von Leerzeichen am Anfang von Zeilen dienen hier als Strukturierungselement, sodass Programmierer zu übersichtlichem Code „gezwungen“ werden. Dafür spart man sich die zusätzlichen Schlüsselwörter beziehungsweise Trennzeichen.

5.3.2 Kontrollflussanweisungen

Ein häufiges Strukturierungselement im Quellcode sind **Kontrollflussanweisungen**, die sich aus einem Anweisungskopf und einem Anweisungskörper zusammensetzen, etwa so etwas:

```
a = 8;
if a == 0:
    print("nein")
elif a == 8:
    print("ja")
else:
    if(a == 5):
        print("sonst")
```

Wesentlich sind hierbei der Doppelpunkt am Ende des Anweisungskopfes und die gleichmäßige Einrückung der zugehörigen Anweisungen. Da diese Art der Anweisungen sehr wichtig, aber auch komplex ist, kommen wir darauf später genauer zu sprechen. Hier soll nur die eine Entscheidungsstruktur als ein Vertreter einer Kontrollflussanweisung an sich vorgestellt werden.

- ▶ Beachten Sie, dass in Python der Blockanweisung oft ein Doppelpunkt in der vorherigen Zeile vorangeht. Dieser gehört dann zu der vorangehenden Anweisung, wie etwa einer Entscheidungsanweisung.

5.3.3 Deklarationsanweisung

Deklarationen dienen der Einführung einer Variablen, einer Datenstruktur, einer Klasse, einer Funktion etc. Zum Teil wurden sie schon behandelt, und wir werden bei Bedarf darauf zu sprechen kommen.

5.3.4 Ausdrucksanweisung

Ein weiterer Fall von uns schon bekannten Anweisungen sind Ausdrucksanweisungen (obwohl sie nicht unter dem Namen aufgetaucht sind). Ausdrucksanweisungen werden in Programmiersprachen häufig verwendet. Es gibt etwa Zuordnungen, aber auch Aufrufe von Funktionen. Diese Anweisungsart kennen wir bereits oder machen uns keine konkreten Gedanken bei der Verwendung.

Was aber zusammenfassen wäre, sind die allgemeinen **Regeln** für Ausdrucksanweisungen unter Python:

- Ausdrucksanweisungen enden mit dem Ende der Zeile. Wenn sie auf mehrere Zeilen verteilt werden sollen, müssen die Folgezeilen eingerückt werden. Genau genommen spricht man in Python von **logischen Zeilen**, durch die Ausdrucksanweisungen unterteilt werden. Das Ende einer logischen Zeile wird durch das Token NEWLINE dargestellt. Statements können keine logischen Zeilengrenzen überschreiten, außer wenn NEWLINE von der Syntax erlaubt ist (zum Beispiel zwischen Anweisungen in zusammengesetzten Anweisungen). Eine logische Zeile wird aus einer oder mehreren physischen Zeilen aufgebaut, indem sie den expliziten oder impliziten Zeilenverbindungsregeln folgt. Eine physische Zeile ist eine Folge von Zeichen, die durch ein Zeilenende beendet wird. In Quelldateien können beliebige der Standardplattformabschlusssequenzen verwendet werden.
- Eine Ausdrucksanweisung wird immer vollständig durchgeführt, bevor die nächste Anweisung ausgeführt wird.
- Eine Zuweisungsanweisung kann einen Ausdruck rechts vom Gleichheitszeichen (dem Zuweisungsoperator) stehen haben. Dieser Ausdruck kann jede andere gültige Ausdrucksanweisung sein.
- Eine Ausdrucksanweisung kann aus Verschachtelungen bestehen. Klammern können zur Festlegung der Reihenfolge dienen, in der die einzelnen Unteranweisungen bewertet werden.

5.3.5 Die leere Anweisung *pass*

Es gibt in Python eine leere Anweisung – *pass*. Diese tut nichts und dient als Platzhalter. Sie ist gar nicht so unsinnig, wie es vielleicht im ersten Moment erscheint. Es gibt beispielsweise folgende sinnvolle Anwendungen:

- Man kann Kennzeichen setzen.
- Man kann eine leere Anweisung bei Bedarf mit Debugging-Befehlen zur Suche nach Fehlern füllen. Wenn die Debugging-Befehle nicht gebraucht werden, kommentiert man sie aus und lässt die leere Anweisung stehen.
- Man kann eine leere Anweisung schon einmal prophylaktisch in einen Source einfügen und erst dann mit Befehlen füllen, wenn man sich richtig dieser Stelle widmen will. Das hilft erheblich gegen Vergesslichkeit.
- Unter Umständen wird bei einer Struktur eine Anweisung benötigt, aber man will gar nichts konkret ausführen.



Datentypen, Variablen und Literale – Die Art der Information

6

6.1 Was behandeln wir in diesem Kapitel?

Das EVA-Prinzip sagt aus, dass bei einem typischen Programm **Daten** eingegeben, verarbeitet und in Form eines Ergebnisses wieder ausgegeben werden. Aber was sind Daten eigentlich? Und wie liegen sie im Rahmen eines Programms vor? Um die Beantwortung dieser Fragen dreht es sich in diesem Kapitel.

6.2 Variablen

Variablen sind benannte Stellen im Hauptspeicher eines Computers, denen zur Laufzeit eines Programms/Skripts temporär Werte zugeordnet und wieder ausgelesen werden können. Diese Zuordnung eines Werts erfolgt in Python über ein Gleichheitszeichen. Das Gleichheitszeichen ist der sogenannte **Zuweisungsoperator**.

6.2.1 Variablen deklarieren

Das Deklarieren (Anlegen) von Variablen erfolgt in Python einfach, indem Sie einen bisher noch nicht verwendeten Bezeichner im Quellcode notieren und dann einen **Wert** (also ein Literal) zuweisen. Für die Benennung von Variablen sollten Sie in Python gewisse Regeln einhalten:

- Sonderzeichen außer dem Unterstrich sind grundsätzlich verboten.
- Eine Variablenbezeichnung darf zwar Zahlen enthalten, aber nicht am Beginn des Bezeichners.
- Seit Python 3 wird Unicode unterstützt. Somit kann der Bezeichner auch Unicode-Zeichen enthalten. Man vermeidet aber dennoch deutsche Umlaute oder das ß. Und auch sonst ist es etablierter Programmierstil, dass man sich auf die Buchstaben aus dem ASCII-Zeichensatz beschränkt.
- Die Länge eines Bezeichners ist im Prinzip nicht begrenzt, sollte aber ein sinnvolles Maß nicht überschreiten. Es gilt, einen Kompromiss aus Aussagefähigkeit und möglichst geringer Anzahl an Zeichen zu finden.

Man sollte also für Bezeichner in der Regel nur Buchstaben aus dem ASCII-Zeichensatz, Zahlen oder den Unterstrich für diese Bezeichner verwenden.

Der Unterstrich am Beginn und Ende eines Bezeichners (auch mehrfach) hat in Python noch einige besondere Bedeutungen. Deshalb sollte man darauf verzichten, wenn man diese Bedeutungen nicht explizit verwenden will oder sich auch gar nicht darüber im Klaren ist.

Hier sehen Sie ein paar Variablen, die so in einem Quellcode angelegt werden können:

Beispiel

```
A = 1.  
b2 = 2.  
meineVar = 3.14.  
meine_var = 2.71. ◀
```

- **Warnung** Auch bei den Bezeichnern von Variablen gilt selbstverständlich die Unterscheidung in Groß- und Kleinschreibung. Oft nutzt man bei Bezeichnern die **Camelnotation** (auch CamelCase-Notation oder Kamel- oder Höckerschrift genannt). Das ist eine Namenskonvention in der IT, bei der ein Name einer Variablen, Funktion, Klasse, etc. aus mehreren Wörtern gebildet wird. Dabei wird jedes neue Wort mit einem Großbuchstaben begonnen. Die Wörter werden direkt aneinandergereiht, es werden keine Leerzeichen oder Unterstriche verwendet. Ein großer Vorteil dieser Notation ist die Möglichkeit eines aussagekräftigen Bezeichners in jedem beliebigen Computersystem, ohne dabei eine Namensbegrenzung zu überschreiten. Dabei werden bei der Camelnotation zwei Varianten unterschieden.

Bei der **lowerCamelCase**-Variante¹ wird der erste Buchstabe der Variablen kleingeschrieben, während bei der **UpperCamelCase**-Variante² (auch

¹ Auch der Name der Notation selbst beginnt hier mit einem Kleinbuchstaben.

² Und hier beginnt der Name der Notation selbstbewusst mit einem Großbuchstaben.

PascalCase) alle Wörter innerhalb der Bezeichnung mit einem Großbuchstaben beginnen.

Die Camelnotation ist in den meisten Programmiersprachen Usus, aber gelegentlich findet man auch eine Schreibweise, bei der zusammengesetzte Bezeichner mit einem Unterstrich getrennt/zusammengefügt werden (**Under-score-Notation** oder oft auch **Snake Case** bzw. `snake_case` genannt). Gerade in Python ist diese sogar oft zu finden.

Ich rate zur Camelnotation bei der Vergabe sämtlicher Bezeichner, wenn Sie weitgehend synchron mit den meisten anderen Programmiersprachen bleiben wollen. In diesem Buch wird diese Notation der Regelfall sein. Welche Variante Sie wählen, ist egal oder wird von den globalen Regeln im Projekt bestimmt. Aber Sie sollten auf jeden Fall die Konvention konsequent durchziehen.

Machen Sie sich klar, dass die einer Variablen zugewiesenen Literale nicht nur Werte darstellen, sondern auch Datentypen (Abschn. 1.2), die sich aus der Art der Werte ergeben.

6.2.2 Variablen im Quellcode verwenden

Angesprochen werden deklarierte Variablen über den Bezeichner, den sie vorher zugewiesen bekommen haben. Dazu werden wir gleich noch ausführliche Beispiele sehen.

6.3 Die Datentypen in Python

Es ist bei einer Programmierung elementar wichtig zu wissen, von was für einem **Typ** eine Variable ist. Die Wahl des Typs einer Variablen (auch Datentyp genannt) hängt unter anderem davon ab, was man mit der Variablen machen möchte. Umgekehrt legt die Art des Datentyps fest, was man dann mit der Variablen machen kann. Eine Variable mit Text als Inhalt ist beispielsweise von einem ganz bestimmten Datentyp. Sie werden mir sicher Recht geben, dass die Division von zwei Texten ziemlich sinnlos ist. Es gibt aber auch Datentypen, in denen man Zahlen darstellen kann. Damit würde eine Division schon eher Sinn ergeben. Warum ein Programmierer also wissen sollte, was für ein Typ von Wert in einer Variablen drinsteht, wird damit deutlich.

Auf der technischen Seite gibt es ebenfalls diverse Gründe, warum eine Variable einen Datentyp benötigt. Im Wesentlichen muss das System wissen, wie viel Speicher für eine Variable oder eine direkte Information wie eine Zahl (ein Literal) eingeplant werden muss.

6.3.1 Lose Typisierung und Typumwandlung in Python

Es ist nun Segen wie auch Fluch von Python und anderen Programmiersprachen dieses Typs, dass Sie als Programmierer Details zu Datentypen gar nicht so genau zu wissen

Abb. 6.1 Die Variable wechselt automatisch den Typ

Die Antwort ist
42
>>>

brauchen. Denn Python verfolgt bei der Zuweisung von Werten zu Variablen und auch der Deklarierung selbst das Konzept der sogenannten **losen Typisierung** (loose typing).

Dies bedeutet, dass von einem Python-Programmierer zwar die Bezeichner von Variablen festgelegt, dabei jedoch die Datentypen nicht (direkt) deklariert werden (können!).³ Dadurch bekommt eine Variable erst dann einen bestimmten Datentyp zugewiesen, wenn ihr der Wert eines bestimmten Typs zugewiesen wurde. Die eigentliche Typfestlegung erfolgt damit im Hintergrund vollkommen automatisch und ohne explizites Zutun des Programmierers.

Ein wesentlicher Vorteil der losen Typisierung ist neben der einfachen Einführung einer Variablen, dass Sie den Datentyp jederzeit ohne Aufwand ändern können. Eine explizite **Typumwandlung** (**Casting**, wie es in der Fachsprache heißt) ist in vielen Fällen damit nicht notwendig. Wenn Sie beispielsweise einer Variablen den Wert einer Zeichenkette zuweisen, können Sie ihr später eine Zahl zuweisen und verändern damit automatisch ihren Typ. Welcher Wert auch immer in der Variablen enthalten ist, er definiert den Datentyp. Dies ist bei vielen konventionellen Programmiersprachen (wie C und C++ und Java) anders. Dort muss der Programmierer auf jeden Fall den Datentyp festlegen, bevor in einer Variablen ein Wert gespeichert werden kann. In der so festgelegten Variablen kann dann auch nur ein Wert des einmal fixierten Typs gespeichert werden. Jede Wertzuweisung eines anderen Datentyps wird einen Fehler erzeugen. Man nennt so etwas dann statische Typisierung von Variablen.

- Erstellen Sie ein neues Python-Programm mit Namen *datentypen1.py* und folgendem Quellcode:

```
var1 = "Die Antwort ist";
print(var1);
var1 = 42;
print(var1);
```

Sie sehen, dass die Variable *var1* im Laufe der Abarbeitung der Anweisungen Werte mit verschiedenen Datentypen aufnehmen konnte (Abb. 6.1).

³Auf anderem Weg jedoch schon.

Lose Typisierung macht den Einstieg in eine Sprache wie Python (oder auch JavaScript oder PHP) für Einsteiger einfach und Quellcode oft viel kompakter, hat aber nicht nur Vorteile. Sonst würden die mächtigeren Programmiersprachen nicht den Aufwand mit der statischen Typisierung betreiben. Lose Typisierung erhöht die Fehlerwahrscheinlichkeit in komplexeren Quellcodes und macht die Wartung unter Umständen aufwendig. Merken Sie sich schon an dieser Stelle: Obwohl Python es zulässt, sollten Sie eine Variable am besten nie im Typ verändern, indem Sie Literale oder Rückgabewerte von Funktionen unterschiedlichen Typs zuweisen. Das ist schlechter Programmierstil, der im letzten Beispiel nur demonstriert wurde, um die grundsätzliche Möglichkeit in Python zu zeigen.

Python ist zwar lose typisiert und hält den Programmierer bei der Deklaration von der konkreten Angabe und dem Zugriff auf Datentypen meist fern. Das bedeutet aber nicht, dass nicht intern mit Datentypen gearbeitet wird. Python ist sogar **typsicher**, weil Variablen ihren Typ nicht mehr ändern können, wenn er einmal ausdrücklich (nicht nur durch Wertzuweisung) festgelegt wurde.

- ▶ Mit der Built-in Function `type()` können Sie den Typ eines Parameters (eine Variable, aber auch ein Wert) bestimmen. Das werden wir in der Folge noch genauer sehen.

6.3.2 Die Python-Datentypen

Im folgenden Abschnitt werden die zum Python-Standard gehörenden Datentypen kurz vorgestellt.

6.3.2.1 **None – Der Nichts-Typ**

Durch den Datentyp *None* wird in Python symbolisiert, dass eine Variable (noch) keinen Wert beinhaltet. Dies ist beispielsweise sinnvoll, wenn man eine Variable definieren, ihr aber erst später einen konkreten Wert zuweisen will. Ein anderer Anwendungsfall wäre die Rückgabe eines Ergebniswerts einer Funktion.

Um einer Variablen den Wert *None* zuzuweisen, gibt man Folgendes ein:

```
a = None
```

- Erstellen Sie ein neues Python-Programm mit Namen *datentypen2.py* und folgendem Quellcode:

```
var1 = None;
print(var1);
var1 = 42;
print(var1);
var1 = None;
print(var1);
```

Abb. 6.2 Verwendung von
None

```
None
42
None
>>>
```

Als Ergebnis wird zuerst *None*, dann der Wert 42 und dann wieder *None* ausgegeben (Abb. 6.2).

- ▶ **Tipp** Der **Wert** *None* kann stets wie eine Konstante verwendet werden. Er kann in Vergleichen verwendet werden, ob eine Variable einen Wert beinhaltet oder nicht, oder auch als Parameter an passende Funktionen übergeben werden.

Noch einmal zur Erinnerung – Python beachtet Groß- und Kleinschreibung.
Die Schreibweisen *none* oder *NONE* sind also für den Nulltyp **falsch!**

6.3.2.2 True und False – Boolesche Werte mit Typ *bool*

Eine boolesche Variable kann nur *True* (wahr) oder *False* (falsch) als Werte annehmen. Beide sind im Sinn von Python Token, und beachten Sie auch jetzt wieder, dass die Groß- und Kleinschreibung genau eingehalten werden muss und *True* und *False* **großgeschrieben** werden!⁴

- ▶ Von Python wird *True* als 1 beziehungsweise *False* als 0 interpretiert, sodass sich auch mit Variablen des Datentyps *bool* rechnen lässt. Umgekehrt gilt jeder (!) numerischer Wert ungleich 0 als *True*. Dieses Verhalten ist aus vielen älteren Sprachen bekannt und recht bequem, gilt aber auch als eine ganz große Schwäche in Hinsicht auf Sicherheit, Stabilität, Zuverlässigkeit und Wartbarkeit. Python entschärft allerdings diese Probleme, worauf wir noch zurückkommen.

Der boolesche Wahrheitswert eines beliebigen Ausdrucks kann mittels der Standard-Funktion *bool()* ermittelt werden, die wir im folgenden Beispiel verwenden wollen.

- Erstellen Sie ein neues Python-Programm mit Namen *datentypen3.py* und folgendem Quellcode:

⁴Diese vielen Wiederholungen mögen nerven, aber falsche Groß- und Kleinschreibweise ist gerade für Einsteiger eine der größten Fehlerquellen.

```
v1 = True;
v2 = True;
v3 = False;
print(v1 + v2);
print(v1 * v3);
print(bool(v1));
print(bool(5));
print(bool(-42));
print(bool(0));
```

In Abb. 6.3 sehen Sie die Ausgabe:

Sie erkennen den Zusammenhang zwischen den numerischen Werten und deren Interpretation als Wahrheitswerte sowie umgekehrt die Möglichkeiten für das mathematische Rechnen mit Wahrheitswerten.

6.3.3 Zahlen – *int*, *float* und *complex*

In Python unterscheidet man drei Varianten von Zahlen:

- ganze Zahlen
- Gleitkommazahlen
- komplexe Zahlen

6.3.3.1 Ganze Zahlen

Ganzzahlige Werte⁵ werden in Python durch den Datentyp *int* repräsentiert.

Beispiel

n1 = 42 ◀

Das zugewiesene Integerliteral legt den Datentyp fest.

Abb. 6.3 Umgang mit
Wahrheitswerten

```
2
0
True
True
True
False
>>>
```

⁵Also ohne Nachkommateil.

```
Python 3.6.2 (v3.6.2:5fd33b5, Jul  8 2017, 04:14:34) [MSC v.1900 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 1l
  File "<stdin>", line 1
    1l
      ^
SyntaxError: invalid syntax
>>> -
```

Abb. 6.4 Bei dieser Python-Installation unter Windows funktioniert die Darstellung als long integer literal nicht

6.3.3.1.1 Große Ganzzahlen

Zahlenliterale können in Python im Prinzip durch ein nachgestelltes *l* oder *L* als „long integer literal“ gekennzeichnet werden. Damit können große Ganzzahlen in Python dargestellt werden, die nicht in einen *int*-Datentyp passen.

Beispiel

7559229999914264593543950336L ◀

- ▶ Es gibt auf einigen Python-Plattformen (insbesondere neueren Versionen 3.x) Probleme mit der Darstellung als long integer literal (Abb. 6.4), während das bei alten Plattformen wunderbar funktioniert (Abb. 6.5)!⁶ Sie sollten diese Auszeichnung vermeiden, wenn Sie sich der Unterstützung nicht sicher sind. Auch die Verwendung der Built-in Function *long()* ist von diesem uneinheitlichen Verhalten betroffen (Abb. 6.6 und 6.7) und sollte in modernen Codes vermieden werden.⁷

⁶Sie können die Unterstützung ganz einfach testen, indem Sie in der Python-Konsole *1 L* eingeben.

⁷Geben Sie in der Python-Konsole einfach *long(1)* ein. Wenn Sie eine Fehlermeldung erhalten, funktioniert die Funktion grundsätzlich nicht.

Abb. 6.5 Bei der Python-Installation in der Version 2.7 unter Linux gibt es mit der Darstellung als long integer literal keine Probleme

```
[GCC 5.4.0 20110609]
Type "help", "copyright", "credits" or "license" for more information.
>>> 1L
1L
>>>
```

Abb. 6.6 Keine Probleme mit der Funktion long() unter Linux

```
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> long(1)
1L
>>>
```

```
>>> long(1)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    long(1)
NameError: name 'long' is not defined
>>>
```

Abb. 6.7 Unter dieser Python-Installation lässt sich die Funktion long() nicht einsetzen

6.3.3.2 Gleitkommazahlen

Zahlen mit Nachkommastellen (**Gleitkommazahlen** oder Gleitpunktzahlen beziehungsweise Floating-Point-Zahlen⁸ genannt) werden in Python durch den Datentyp *float* repräsentiert. Die Nachkommastellen eines Gleitkommaliterals werden dabei – wie im englischen Sprachraum üblich – nicht durch ein Komma, sondern durch einen Punkt von dem ganzzahligen Anteil getrennt. Zudem ist es möglich, sehr große oder sehr kleine *float*-Zahlen mittels *e* oder *E* in Exponentialschreibweise anzugeben. Die Zahl hinter dem *e* gibt dabei an, um wie viele Stellen der Dezimalpunkt innerhalb der Zahl verschoben wird.

⁸Wegen der Verwendung des Punkts anstelle von einem Komma im angloamerikanischen System.

Beispiel

N1 = 4.3e5 # Entspricht 430.000. ◀

- ▶ Das Rechnen mit Gleitkommazahlen nennt man Gleitkommaarithmetik. Grundsätzlich ist Gleitkommaarithmetik ein kritischer Vorgang, bei dem es oft zu Rundungsproblemen kommt. Man versucht komplexere Gleitkommaarithmetik weitgehend zu vermeiden.
- ▶ Für das **Runden** von Gleitkommazahlen gibt es eine Built-in Function mit Namen *round()*. Hier kann man auch die Anzahl der Stellen angeben, auf die gerundet werden soll.

6.3.3.3 Komplexe Zahlen

Komplexe Zahlen bestehen aus einem Realteil und einem Imaginärteil. Der Imaginärteil besteht aus einer reellen Zahl, die mit der imaginären Einheit *i* multipliziert wird. Die imaginäre Einheit zum Quadrat ergibt – 1. Oder umgekehrt beschrieben: Die Wurzel aus – 1 ergibt die imaginäre Einheit.

- ▶ Beachten Sie, dass die imaginäre Einheit in Python durch ein *j* statt einem *i* dargestellt wird. Daran habe ich als Mathematiker schwer zu schlucken, aber diese Darstellung kommt aus der Elektrotechnik.

Um eine komplexe Zahl zu definieren, gibt man in Python etwa Folgendes ein:

Beispiel

N1 = 1 + 4j. ◀

Wir wollen die numerischen Datentypen nun in einem Beispiel verwenden und dabei auch Built-in Functions von Python verwenden, die explizit mit diesen Datentypen korrespondieren.

- Erstellen Sie ein neues Python-Programm mit Namen *datentypen4.py* und folgendem Quellcode:

```
n1 = 42;  
print(n1);  
print(float(n1));  
n2 = 4.3e5; # 430000  
print(n2);  
print(int(n2));  
n3 = 3.14;
```

```
print(n3);
print(int(n3));
print(round(n3,1));
n4 = 1 + 4j;
print(n4);
print(abs(n4));
```

Die Ausgabe wird sein (Abb. 6.8):

Aber wie ergeben sich diese Resultate? Das ist teils nicht ganz trivial. Schauen wir uns die Details genauer an:

- Die erste Ausgabe ist einfach. Die Variable *n1* wird durch Zuweisung des numerischen Literals 42 zu einem *int*-Datentyp und der Wert wird unverändert ausgegeben.
- Bei der zweiten Ausgabe wurde aber der *int*-Datentyp über die Built-in Function *float()* in einen Gleitkommatyp gewandelt. Der Wert ändert sich natürlich nicht, aber in der Ausgabe erkennen Sie den ergänzten Nachkommateil, der den Wert 0 haben muss.
- Die dritte Ausgabe gibt dezimal den Wert der Gleitkommavariablen aus, die in der Exponentialschreibweise deklariert wurde.
- In der nächsten Ausgabe wurde der Nachkommanteil der Zahl einfach mit *int()* abgeschnitten. Auch wenn damit nur ein Nullwert wegfällt – das ist dann explizit ein *int*-Datentyp, der ausgegeben wird.
- Ausgabe 5 ist die ganz normale Darstellung einer Gleitkommazahl – genau so, wie sie deklariert wurde.
- In der sechsten Ausgabe wurde der Nachkommanteil der Zahl wieder mit *int()* abgeschnitten, und hier fällt wirklich Information weg.
- Statt dem Wandeln des Datentyps schneidet *round()* einfach nach der Stelle eine Gleitkommazahl ab, die als zweiter Parameter angegeben wird. Das führt dazu, dass in dem Beispiel eine Nachkommastelle erhalten bleibt.
- Die vorletzte Ausgabe zeigt die Darstellung einer komplexen Zahl in der Konsole.

Abb. 6.8 Numerische Datentypen

```
42
42.0
430000.0
430000
3.14
3
3.1
(1+4j)
4.123105625617661
>>> |
```

- Mit komplexen Zahlen kann man entsprechend den mathematischen Regeln rechnen. So kann man etwa den Absolutwert bilden. Das wird für die letzte Ausgabe mit der Built-in Function `abs()` gemacht.

6.3.4 Zeichenketten -(str) und Zeichenliterale

Zeichenketten beziehungsweise Strings oder Zeichenkettenliterale sind eine Folge von Zeichen, die wahlweise in einfachen oder doppelten Anführungszeichen geschrieben werden und vom Typ `str` sind. Beispiel:

Beispiel

```
s1='English as She Is Spoke'  
s2="The Lumberjack-Song" ◀
```

Die einzelnen Zeichen innerhalb einer Zeichenkette sind **Zeichenliterale**, die auch **maskiert** werden können. Das bedeutet, dass man eine codierte Darstellung verwendet – die **Escape-Darstellung**.

- Die Escape-Zeichenliterale beginnen immer mit dem Backslash-Zeichen.
- Diesem folgt eines der Zeichen b, t, n, f, r, ',' oder \ oder eine Serie von Ziffern, denen ein Zeichen für die Art der Darstellung vorangeht.

In der nachfolgenden Tabelle (Tab. 6.1) sehen Sie einige Beispiele für besondere Zeichen in verschiedenen Darstellungen.

- Erstellen Sie ein neues Python-Programm mit Namen `datentypen5.py` und folgendem Quellcode (Abb. 6.9):

```
print("Hier folgt ein Tabulator\tund dann der Rest");  
print("Hier folgt ein Zeilenumbruch\nund dann der Rest");  
print("Hier folgt ein maskiertes Zeichen in Hexadezimal-Darstellung:\t\x65");  
print("Hier folgt ein maskiertes Zeichen in Oktal-Darstellung:\t\555");
```

Tab. 6.1 Wichtige Escape-Sequenzen von Zeichenliteralen

Escape-Literal	Bedeutung
\'	Einfaches Anführungszeichen
\"	Doppeltes Anführungszeichen
\\\	Backslash
\b	Rückschritt (Backspace)
\f	Formularvorschub (Formfeed)
\n	Neue Zeile (New line)
\r	Wagenrücklauf (Return)
\t	Tabulatur (Tab)
\a	ASCII Bell – Klingeln
\uxxx	Ein Zeichen in Unicode (UTF-16) mit drei hexadezimalen Zahlen. Die Darstellung macht auf einigen Python-Plattformen Schwierigkeiten
\Uxxxxxxxx	Ein Zeichen in Unicode (UTF-32) mit acht hexadezimalen Zahlen. Die Darstellung macht auf einigen Python-Plattformen Schwierigkeiten
\ooo	Ein Zeichen in der Oktal-Darstellung mit drei oktalnen Zahlen
\xhh	Ein Zeichen in der Hex-Darstellung mit zwei hexadezimalen Zahlen

```

Hier folgt ein Tabulator      und dann der Rest
Hier folgt ein Zeilenumbruch
und dann der Rest
Hier folgt ein maskiertes Zeichen in Hexadezimal-Darstellung:   e
Hier folgt ein maskiertes Zeichen in Oktal-Darstellung: ü
>>> |

```

Abb. 6.9 Zeichenketten und Sonderzeichen

6.4 Den Datentyp bestimmen und umwandeln

Python kümmert sich im Hintergrund weitgehend automatisch um die Verwaltung von Datentypen. Aber es kann dennoch in gewissen Situationen wichtig oder gar notwendig sein, dass Sie den Datentyp genau kennen und auch gezielt umwandeln. Zum Teil haben wir uns damit schon beschäftigt, aber jetzt schauen wir uns das Konzept genauer an.

6.4.1 Den Datentyp mit `type()` dynamisch bestimmen

In Skripten beziehungsweise Programmen oder auch im interaktiven Modus kann der Datentyp eines Objekts oder einer Variablen jederzeit mittels der Built-in Function `type()` ermittelt werden. Das könnte man so machen (*datentypen6.py*):

```
a = "Der Papagei ist tot";
print(type(a));
b = 1;
print(type(b));
c = 1.0;
print(type(c));
d = True;
print(type(d));
print(1 == True);
e = 1 + 3j;
print(type(e));
f = 4e2;
print(type(f));
```

Sie erkennen an der Ausgabe (Abb. 6.10), dass `type()` die Datentypen bestimmt. Aber Ihnen sollte noch etwas auffallen, auch wenn die genaue Bedeutung vielleicht noch nicht ganz klar wird. Wenn man unter die Oberfläche schaut, muss man deutlich sagen, dass es in Python in Wirklichkeit gar keine primitiven Datentypen gibt. In Python ist intern alles ein Objekt. Selbst ganze Zahlen oder boolesche Variablen sind in Python Objekte, auch wenn die Art des Zugriffs wie bei primitiven Datentypen erfolgt. Darauf gehen wir natürlich noch genauer ein, aber die Ausgaben des letzten Beispiels verdeutlichen das schon jetzt. Es ist weder Zufall noch bedeutungslos, dass immer „`class`“ in den spitzen Klammern bei dem Datentyp auftaucht.

6.4.2 Implizite und explizite Typumwandlung

Wir sind schon damit in Berührung gekommen, dass man unter **Casting** beziehungsweise **Typkonvertierung** die Umwandlung von einem Datentyp in einen anderen Datentyp versteht. Python wandelt oft Datentypen im Hintergrund automatisch um.

Diese impliziten Ad-hoc-Konvertierungen ohne ausdrückliches Zutun des Programmierers sind sehr bequem, aber es gibt Situationen, in denen eine explizite Konvertierung notwendig ist, um absichtlich den Datentyp eines Wertes zu verändern. Für eine **explizite Typumwandlung** stellt Python Built-in Functions zur Verfügung, deren Namen sich an den Datentypen orientieren und von denen wir zwei Funktionen bereits eingesetzt haben:

Abb. 6.10 Typüberprüfungen

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'bool'>
<class 'complex'>
<class 'float'>
>>>
```

- Mit `int()` wandelt man einen Parameter in einen `int`-Datentyp.
- Mit `float()` wandelt man einen Parameter in einen `float`-Datentyp.
- Mit `str()` wird ein Parameter in einen String überführt.

Darüber hinaus gibt es noch einige weitere Konvertierungsfunktionen. Etwa für Listen, Tupel etc., was derzeit noch nicht behandelt wurde.

- Beachten Sie, dass sich bei einer expliziten Umwandlung der Parameter auch wirklich in den Zieltyp wandeln lassen muss. Sie können keinen beliebigen Text in eine Zahl wandeln, aber auch keine komplexe Zahl in eine Ganzzahl oder Gleitkommazahl.
- Erstellen Sie ein neues Python-Programm mit Namen `datentypen7.py` und folgendem Quellcode:

```
v1 = 1;
v2 = 3.14;
v3 = "42";
print(type(v1));
print(type(v2));
print(type(v3));
print(type(float(v1)));
print(type(float(v3)));
print(type(int(v2)));
print(type(int(v3)));
print(type(str(v1)));
print(type(str(v2)));
```

An der Ausgabe mit der Typüberprüfung (Abb. 6.11) können Sie genau sehen, dass und wie die verschiedenen Datentypen ineinander umgewandelt wurden.

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'float'>
<class 'float'>
<class 'int'>
<class 'int'>
<class 'str'>
<class 'str'>
>>>
```

Abb. 6.11 Typumwandlungen in verschiedene Richtungen



Ausdrücke, Operatoren und Operanden – Die Verarbeitung von Daten

7

7.1 Was behandeln wir in diesem Kapitel?

Die Eingabe und Ausgabe wird in Python im Wesentlichen durch Built-in Functions bereitgestellt. Wenn das EVA-Prinzip auch die Verarbeitung von Daten beinhaltet, dann muss es dafür Mechanismen geben. Auf der untersten Ebene macht man das mit sogenannten **Ausdrücken**, die wiederum aus **Operatoren** und **Operanden** aufgebaut werden. Um dieses Thema kümmern wir uns in diesem Kapitel.

7.2 Ausdrücke

Unter einem **Ausdruck** versteht man allgemein das, was einen Wert in einer Programmierung repräsentiert. Da gibt es einmal den festen Wert, wie er durch ein Literal repräsentiert wird. Oder der Wert entsteht durch eine Berechnung oder eine Ermittlung eines Werts durch die Steuerung des Programmflusses (eine Operation).

Ausdrücke können Literale, Konstanten, Variablen, Schlüsselwörter, Operatoren und andere Ausdrücke beinhalten. Wir haben in unseren bisherigen Beispielen natürlich schon diverse Python-Ausdrücke verwendet, ohne uns um die Tatsache groß zu kümmern. Aber das steht jetzt auf dem Stundenplan.

Man kann Ausdrücke am einfachsten folgendermaßen definieren:

- ▶ Ausdrücke sind das Ergebnis der Verbindung von Operanden und Operatoren über die syntaktischen Regeln der Sprache.

Ausdrücke werden also für die Durchführung von Operationen (Manipulationen) an Variablen oder Werten verwendet. Dabei sind Spezialfälle wie arithmetische Konstanten beziehungsweise Literale kein Widerspruch, sondern hier kommt eine sogenannte leere Operation zur Anwendung.

7.3 Operationen mit Operatoren und Operanden

In jeder Programmiersprache kann man Operationen ausführen. Ein **Operator** ist dabei ein Symbol beziehungsweise Token (Sinnzusammenhang). Er gibt an, welche Operation in einem Ausdruck ausgeführt werden soll. Die Elemente (Variablen oder Literale), mit denen eine solche Operation durchgeführt wird, nennt man **Operanden**. Im numerischen Fall sind das zum Beispiel Zahlen. Aber es gibt nicht nur numerische Operanden und Operatoren.

Python verfügt zwar über die üblichen Operatoren, aber aufgrund des Konzepts der Einfachheit und Übersichtlichkeit ist die Menge gegenüber vielen anderen Programmiersprachen reduziert.

Man unterscheidet die Operatoren nun nach ihrem Typ.

7.3.1 Arithmetische Operatoren

Arithmetische Operatoren (auch **mathematische Operatoren** genannt) benutzen immer einen oder zwei Zahlenoperanden und liefern als Ergebnis einer arithmetischen Operation (Berechnung) eine neue Zahl als Wert. So etwas kennt man aus der Schulmathematik, und auch die Operatoren sind vermutlich weitgehend vertraut. Es gibt aber Feinheiten zu beachten. Tab. 7.1 gibt die arithmetischen Python-Operatoren an.

Mit den arithmetischen Operatoren können Sie wie aus der Mathematik gewohnt Berechnungen durchführen. Wenn die Berechnung mehrere Operatoren verwendet, gilt (natürlich) die Punkt-vor-Strich-Regel (Abschn. 1.4).

Beispiel

erg = 1 + 2 * 3 ◀

Das wäre ein weiteres Beispiel mit einem vollständigen Python-Listing, in dem aber die trivialen mathematischen Operationen nicht vorkommen, sondern nur die, welche besonderer Aufmerksamkeit bedürfen (*op1.py*):

```
a = 11
b = 0.5
c = 3
```

Tab. 7.1 Die arithmetischen Python-Operatoren

Operator	Bedeutung	Beschreibung	Beispiel
-	Subtraktion und Negation	Mit zwei numerischen Operanden führt der Operator die Subtraktion von zwei Zahlen oder numerischen Variablen aus. Er dient aber bei einem Operanden auch zur einstelligen arithmetischen Negierung. Der Operator wird zur Negierung eines Ausdrucks verwendet, indem er einfach dem Ausdruck vorangestellt wird. Wenn er vor einem Ausdruck steht, wird der Wert negativ.	6 - 1 -5
%	Modulo	Diese Operation gibt den Rest einer Division zurück. Die Restwertberechnung ist in Python sogar für Gleitkommazahlen definiert! Sie sollten davon allerdings Abstand nehmen, denn es kann leicht zu Rundungsproblemen kommen (Abb. 7.1).	15 % 9
*	Multiplikation	Der Operator definiert die Multiplikation von zwei Zahlen oder numerischen Variablen.	2 * 3
/	Division	Der Operator definiert die Division von zwei Zahlen oder numerischen Variablen.	6 / 2
+	Addition und Gegenteil der Negation	Mit zwei numerischen Operanden führt der Operator die Addition von zwei Zahlen oder numerischen Variablen aus. Das Ergebnis der Operation ist die Summe der Operanden. Mit einem nachgestellten Operanden ist der Operator nur aus Symmetriegründen vorhanden, um das Gegenteil der Negation formulieren zu können.	1 + 1 + 1
**	Potenz	Der erste Operator wird mit dem zweiten Operator zur Potenz genommen	3 ** 2
//	Ganzzahldivision	Bei dieser Form der Division wird das Ergebnis eine ganze Zahl sein. Ein eventueller Nachkommateil wird abgeschnitten	7 // 2

```
d = 0.7
print(a % c)
print(a % b)
print(a % d)
print(a ** c)
print(a ** b)
print("a" + "b")
print(a + 2.9)
print(1 + 2.9)
print(False + True)
print(a + True)
```

Das wäre die Ausgabe (Abb. 7.1):

Abb. 7.1 Mathematische Operationen

```
2
0.0
0.5000000000000007
1331
3.3166247903554
ab
13.9
3.9
1
12
>>> |
```

```
0.0
0.5000000000000007
1331
3.3166247903554
ab
13.9
3.9
1
12
```

Das Listing enthält diverse bemerkenswerte Stellen. Deshalb gehen wir jede einzelne Operation durch:

1. Die erste Ausgabe rechnet 11 modulo 3. Die 3 geht dreimal in die Zahl 11. 3×3 ergibt 9, und dieser Wert wird „vergessen“. Als Rest bleibt also der Wert 2.
2. Die zweite Ausgabe nutzt die Modulooperation mit Gleitkommazahlen. Das ist in Python ja wie gesagt erlaubt. Die Zahl 11 lässt sich durch den Wert 0.5 teilen, und deshalb bleibt als Rest 0.0. Beachten Sie, dass der Zieltyp ein Gleitkommatyp ist, auch wenn einer der Operanden (das Literal 11) eine ganze Zahl war.
3. Für die dritte Ausgabe machen wir das Gleiche wie für die zweite Ausgabe. Nur passt dieses Mal der zweite Operand **nicht** (!) ohne Rest in den ersten Operanden. Es bleibt also ein von 0.0 verschiedener Rest. Auch das ist legitim. Aber Sie erkennen deutlich die Rundungsprobleme. Wegen dieser Probleme sollte man lieber auf Gleitkommaarithmetik verzichten oder diese nur mit großer Vorsicht anwenden.¹
4. Die Ausgabe Nummer 4 berechnet einfach die Potenz des ersten Operanden (der Basis) mit dem zweiten Operanden als Exponenten.
5. Auch die Ausgabe Nummer 5 berechnet die Potenz. Aber der Exponent ist dieses Mal eine Gleitzahl – der Wert 0.5. Damit ziehen wir die Quadratwurzel aus der Zahl 11. Mit Exponenten zwischen 0 und 1 kann man also Wurzeln ziehen, wie auch vielleicht noch aus der Schulmathematik bekannt.

¹Darauf wurde schon mehrfach hingewiesen und hier sehen Sie einen ganz praktischen Beweis.

6. Wenn Sie den Plusoperator auf Strings anwenden, ist das keine (!) arithmetische Operation, sondern einen String-Verknüpfung. Das sehen Sie in der nächsten Ausgabe.
7. In den folgenden beiden Operationen werden Literale verwendet. Das geht natürlich auch. Zuerst werden eine Variable und ein Literal verbunden. In der nächsten Ausgabe wird die Summe von zwei Literalen gebildet.
8. Man kann mit booleschen Literalen *True* und *False* rechnen, denn diese repräsentieren ja in Python die Zahlen 1 und 0. Die letzten beiden Ausgaben zeigen das.

Wegen der Punkt-vor-Strich-Regel, die ein Spezialfall einer Prioritätenreihenfolge der Operatoren darstellt (Abschn. 1.4.1), würde bei dem nächsten Beispiel der Wert der Variablen *erg* 37 sein.

Beispiel

`erg = 2 + 5 * 7` ◀

- ▶ **Die Verwendung von Klammern ist in Ausdrücken erlaubt und sehr oft sinnvoll, um die Übersicht zu erhöhen. Vor allem können Sie damit die Reihenfolge der Bewertung verändern.**

Beispiel

`erg = (2 + 5) * 7` ◀

Der Wert von *erg* wird 49 sein.

7.3.2 Der String-Verkettungsoperator

Der Plusoperator wird in Python auch zur String-Verkettung verwendet. Das verbindet zwei Strings zu einem einzigen String, wie wir auch schon im letzten Beispiel gesehen haben.

- ▶ Sobald einer der Operanden des Plusoperators kein String ist, wird in Python **keine** automatische String-Verkettung vorgenommen. In vielen anderen Sprachen ist so ein Verhalten der Fall. In Python müssen Sie gegebenenfalls einen unpassenden Operanden erst mit der Built-in Function *str()* in einen String wandeln. Sonst kommt es zum Fehler.

Tab. 7.2 Die Zuweisungsoperatoren

Operator	Bedeutung	Beispiel
<code>%=</code>	Modulo- und Zuweisungsoperator	Die Anweisung <code>a %= 5</code> nimmt den Wert von <code>a</code> Modulo 5. Die alternative Schreibweise wäre <code>a = a % 5</code> .
<code>*=</code>	Multiplikations- und Zuweisungsoperator	Die Anweisung <code>a *= 5</code> multipliziert den Wert der Variablen <code>a</code> mit dem Wert 5. Die alternative Schreibweise wäre <code>a = a * 5</code> .
<code>/=</code>	Divisions- und Zuweisungsoperator	Die Anweisung <code>a /= 5</code> teilt den Wert von <code>a</code> durch 5. Die alternative Schreibweise wäre <code>a = a / 5</code> .
<code>+ =</code>	Additions- und Zuweisungsoperator. Der Operator kann auch in Verbindung mit Strings verwendet werden. Dann handelt es sich um eine Zuweisung mit String-Verkettung	Die Anweisung <code>a += 5</code> erhöht den Wert von der Variablen <code>a</code> um den Wert 5. Die alternative Schreibweise wäre <code>a = a + 5</code> .
<code>=</code>	Einfacher Zuweisungsoperator	Die Anweisung <code>a = 5</code> weist <code>a</code> den Wert 5 zu.
<code>-=</code>	Subtraktions- und Zuweisungsoperator	Die Anweisung <code>a -= 5</code> reduziert den Wert von <code>a</code> um 5. Die alternative Schreibweise wäre <code>a = a - 5</code> .
<code>**=</code>	Exponentialrechnung mit Zuweisung	Die Anweisung <code>c **= a</code> ist äquivalent zu <code>c = c ** a</code> .
<code>//=</code>	Ganzzahldivisions- und Zuweisungsoperator	Die Anweisung <code>a //= 5</code> teilt den Wert von <code>a</code> durch 5 und „vergisst“ einen Nachkommateil.

7.3.3 Zuweisungsoperatoren

Es gibt neben dem einfachen Zuweisungsoperator in Python die **arithmetischen Zuweisungsoperatoren**. Diese sind als Abkürzung für arithmetische Operationen (dies bedeutet mathematische Berechnungen) mit einer nachfolgenden Zuweisung zu verstehen und bauen auf dem Gleichheitsoperator mit dem vorangestellten Token für die arithmetische Operation auf (Tab. 7.2).

- Erstellen Sie eine neue Datei mit Namen `op2.py`. Geben Sie den Quelltext ein:

```
a = 11
a *= 2
print(a)
a %= 13
print(a)
```

```
a **= 2
print(a)
```

Die Ausgabe wird sein:

```
22
9
81
```

- In der ersten Operation wird der Wert 11 mit dem Wert 2 multipliziert. Das ergibt 22.
- In der zweiten Operation wird der Wert 22 modulo dem Wert 13 genommen. Als Rest bleibt der Wert 9.
- Die dritte Operation nimmt den Wert 9 zum Quadrat, was 81 ergibt.

7.3.4 Boolesche Operatoren (Vergleichsoperatoren)

Sie können bei logischen Ausdrücken nicht nur die Gleichheit von zwei Seiten überprüfen, sondern diverse andere Bedingungen, etwa ob der Wert auf einer Seite mit numerischen Operanden kleiner oder größer ist. Dazu stehen Ihnen unter Python die folgenden

Tab. 7.3 Vergleichsoperatoren

Operator	Bedeutung	Beschreibung	Beispiele	Ergebnis
<code>!=</code>	Ungleichheit	Ein Vergleich, ob zwei Operanden im Wert nicht identisch sind. Der Vergleich liefert dann True, wenn zwei Ausdrücke rein im Wert nicht gleich sind.	<code>8 != 4 + 4</code> <code>7 != 4 + 4</code>	False True
<code><</code>	Kleiner als	Der Vergleich, ob der erste Operand kleiner als der zweite Operand ist.	<code>5 < 6</code>	True
<code><=</code>	Kleiner als oder gleich	Der Vergleich, ob der erste Operand kleiner oder gleich dem zweiten Operanden ist.	<code>6 <= 6</code>	True
<code>==</code>	Gleichheit	Der Vergleich, ob zwei Operanden im Wert identisch sind. Das doppelte Gleichheitszeichen darf nicht mit dem Zuweisungsoperator (<code>=</code>) verwechselt werden!	<code>4 == 5</code> <code>"4" == 4</code> <code>4 == 2 + 2</code>	False True True
<code>></code>	Größer als	Der Vergleich, ob der erste Operand größer als der zweite Operand ist.	<code>5 > 6</code>	False
<code>>=</code>	Größer als oder gleich	Der Vergleich, ob der erste Operand größer oder gleich dem zweiten Operanden ist.	<code>5 >= 5</code> <code>6 >= 5</code> <code>4 >= 5</code>	True True False

Abb. 7.2 Vergleiche

```
False
True
True
True
False
False
True
False
True
True
False
>>> |
```

logischen Vergleichsoperatoren zur Verfügung (Tab. 7.3). In jedem Fall erhalten Sie entweder *True* oder *False* als Ergebnis.

- Erstellen Sie eine neue Datei mit Namen *op3.py* und dem Quelltext:

```
print(8 != 4 +4)
print(7 != 4 +4)
print(5 < 6)
print(6 <= 6)
print(4 == 5)
print("4" == 4)
print(4 == 2 + 2)
print(5 > 6)
print(5 >= 5)
print(6 >= 5)
print(4 >= 5)
```

Die Ergebnisse sollten offensichtlich und nicht bemerkenswert sein (Abb. 7.2). Mit einer Ausnahme – denn der Vergleich "4" == 4 liefert *False*. Python nimmt also **keine** implizite Typkonvertierung vor. In vielen anderen lose typisierten Sprachen (etwa JavaScript oder PHP) ist das anders.

7.3.5 Logische Operatoren

Die logischen Operatoren sind Vergleichsoperatoren sehr ähnlich, dienen jedoch der **Verknüpfung** von booleschen Werten beziehungsweise booleschen Ausdrücken, wie sie beispielsweise über die zuvor aufgeführten Vergleichsoperatoren gebildet werden (Tab. 7.4). Das Ergebnis einer solchen Verknüpfung ist immer erneut ein boolescher Wert.

Tab. 7.4 Logische Operatoren

Operator	Bedeutung	Beschreibung	Beispiele	Ergebnis
and	Logisches Und (And)	Der Operator vergleicht zwei Ausdrücke und liefert nur dann True, wenn beide verglichenen Ausdrücke beziehungsweise Operanden wahre Ergebnisse liefern. Sonst erhalten Sie False.	(4+4 == 8) and (2+3 == 5)	True
or	Logisches Oder (Or)	Der Operator vergleicht zwei Ausdrücke und liefert bereits dann True zurück, wenn einer der beiden verglichenen Ausdrücke ein wahres Ergebnis liefert. Ebenso liefert der Operator True, wenn beide wahr sind. Dies ist also kein ausschließendes Oder (XOR). Nur wenn beide Operanden unwahr sind, liefert der Operator False.	(4+4 == 8) or (2+3 == 5)	True
not	Logisches Nicht (Not)	Der Operator dreht einen logischen Wert (oder meist das Ergebnis einer nachgestellten booleschen Operation) um. Aus True wird False und aus False wird True.	not(4+4 == 8)	False

- Erstellen Sie eine neue Datei mit Namen *op4.py* und dem Quelltext:

```
print( (4 + 4 == 8) and (2 + 3 == 5) )
print( (4 + 4 == 8) and (1 + 3 == 3) )
print((4 + 4 == 8) or (2 + 3 == 5) )
print((4 + 4 == 7) or (2 + 3 != 5) )
print(not(1 + 4 == 5))
```

Die Ausgabe wird das sein:

True

False

True

False

False

Versuchen Sie die Ergebnisse nachzuvollziehen, insbesondere, was der boolesche Wert der einzelnen verknüpften Teilbedingungen ist, wann beide *True* liefern müssen und wann bereits einmal das Teilergebnis *True* genügt, damit das Gesamtergebnis *True* ist.

7.3.6 Die Membership-Operatoren

Pythons Membership-Operatoren (Mitgliedschaftsoperatoren) testen die Mitgliedschaft eines Operanden in einer Sequenz. Das kann zum Beispiel ein String sein, wie wir ihn schon behandelt haben, aber auch eine Liste oder ein Tupel, was bisher noch nicht eingeführt wurde.

Es gibt zwei Mitgliedsoperatoren.

- Der Operator *in* testet, ob sich der erste Operand in dem zweiten Operanden (der Sequenz) befindet.
- Der zweite Mitgliedsoperator ist streng genommen kein eigener Operator, sondern die Kombination aus *not* und *in* und testet, ob sich der erste Operand **nicht** in dem zweiten Operanden befindet.

Erstellen Sie eine neue Datei mit Namen *op5.py* und dem Quelltext:

```
suchausdruck = "@"  
  
sequenz="ralph_steyer@gmx.de"  
  
print(suchausdruck in sequenz)  
print(suchausdruck not in sequenz)
```

Die erste Ausgabe wird *True* ergeben, denn das Zeichen @ ist Teil des zu untersuchenden Strings. Entsprechend wird die zweite Ausgabe *False* ergeben.

7.3.7 Identitätsoperatoren

Identitätsoperatoren vergleichen die Speicherplätze zweier Objekte. Es gibt zwei Identitätsoperatoren, wobei der zweite wie eben nur die Kombination mit *not* und damit die Negation der ersten Version darstellt.

Der Operator *is* liefert True, wenn die Operanden auf beiden Seiten des Operators auf das gleiche Objekt verweisen, und *is not* entsprechend im umgekehrten Fall. Da bisher noch keine Objekte behandelt wurden, wird hier auf ein Beispiel verzichtet.

Tab. 7.5 Binäroperatoren

Operator	Beschreibung
<<	Operator für bitweise Verschiebung nach links
>>	Operator für bitweise Verschiebung nach rechts
&	Bitweiser AND-Operator
	Bitweiser OR-Operator
^	Bitweiser XOR-Operator – das ausschließende Oder (engl. "exclusive OR")
~	Bitweises NICHT – auch bitweiser Komplement-Operator genannt

Tab. 7.6 Priorität der Python-Operatoren

Operator	Beschreibung
**	Exponentialoperator
~ + -	Komplement
* / % //	Punktrechnung
+ -	Strichrechnung
>> <<	Rechts und links bitweise verschieben
&	Bitweise 'UND'
^	Bitweise exklusives 'ODER' und regelmäßiges 'ODER'
<=<>>=	Vergleichsoperatoren
== !=	Gleichstellungsbetreiber
= %= /= // -= -= + *= ** =	Zuweisungsoperatoren
is, is not	Identitätsoperatoren
in, not in	Mitgliedschaftsoperatoren
not or and	Logische Operatoren

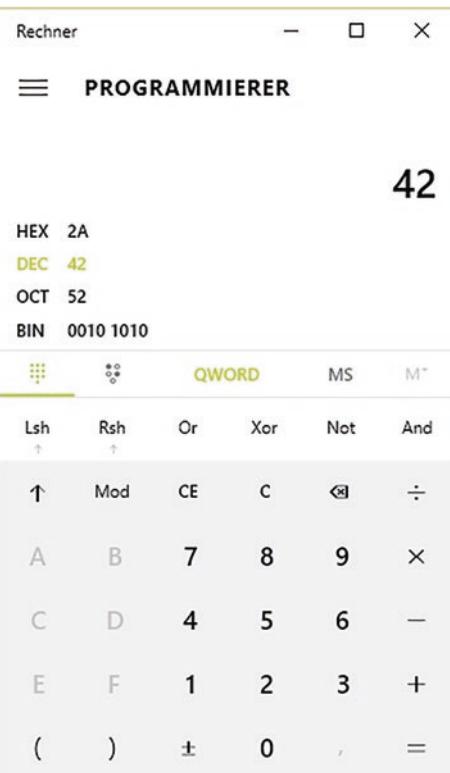
7.3.8 Bitweise Operatoren

Jedes Zeichen auf dem Computer wird bekanntlich intern in Form des Binärsystems aus Nullen und Einsen dargestellt, was den Wert in einer numerischen Zeichencodierung wie ASCII entspricht. Python kennt wie viele andere Programmiersprachen Operatoren, welche eine Manipulation von Werten auf Bit-Ebene erlauben. Bitweise Operatoren (Binäroperatoren) ermöglichen Operationen auf Basis der Binärdarstellung von Zeichen (Tab. 7.5).

7.3.8.1 Verschiebungsoperatoren

Schauen wir zuerst auf die Verschiebungsoperatoren. Um die Wirkung solcher Operatoren deutlich zu machen, betrachten wir die Binärdarstellung eines Zeichens.

Abb. 7.3 Umrechnung von Werten in verschiedene Zahlensysteme



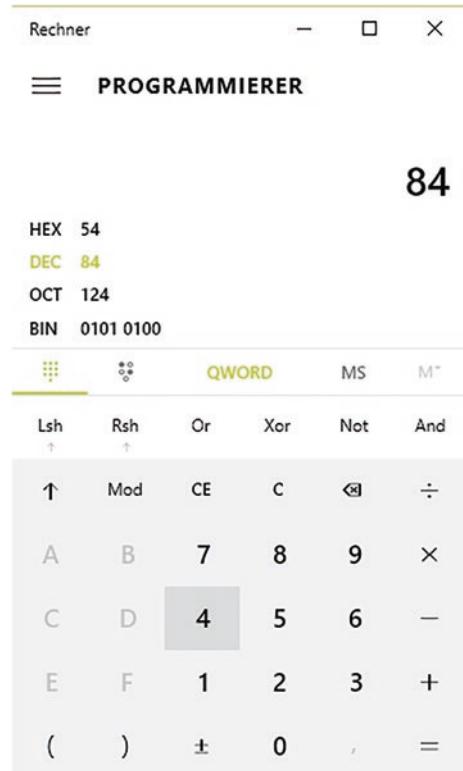
Nehmen wir die Zahl 42 als Beispiel. Deren Binärdarstellung ist *00.101.010*. Das können Sie leicht mit einem Taschenrechner umrechnen, wenn Sie sich nicht selbst die Mühe einer Umrechnung des Zahlensystems machen wollen (Abb. 7.3).

Nun bedeutet die Anwendung der Operation $<< 1$, dass alle Bits um eine Stelle nach links verschoben werden. Von rechts kommt eine 0 dazu. Das führt zu der Binärdarstellung *01.010.100* und das ist dezimal 84, was zur Not der Taschenrechner „beweist“ (Abb. 7.4).

Das bedeutet nun, dass $<< 1$ (eine Verschiebung der Bits um eine Stelle nach links) nichts weiter als die Verdoppelung des Wertes darstellt. Das ist auch aufgrund der Basis 2 des Binärsystems klar. Zwangsläufig führt eine Verschiebung um 2 Stellen nach links ($<< 2$) zu einer Vervierfachung.

Wie geht es nun in die andere Richtung?

Der Ausdruck $>> 1$ verschiebt die Bits um eine Stelle nach rechts. Das bedeutet, dass bei der Binärdarstellung der Zahl 42 (*00.101.010*) das am weitesten rechts stehende Bit verschwindet und links eine binäre 0 dazukommt. Das führt in der Binärdarstellung zum Wert *00.010.101*. Dezimal entspricht das dann der Zahl 21. Offensichtlich steht der Aus-

Abb. 7.4 Was ist 84 binär?

druck $>> 1$ für eine Halbierung des Wertes. Wenn wir jetzt bei der Binärdarstellung der Zahl 42 (00.101.010) zwei Stellen nach rechts schieben ($>> 2$), wird geviertelt.

Beachten Sie, dass bei einer Verschiebung nach rechts eine **Ganzzahldivision** durchgeführt wird! Es bleibt kein Rest beziehungsweise der Rest wird „vergessen“.

- Erstellen Sie das Beispiel mit Namen *op6.py* und dem Quelltext:

```

z1 = 42
a = z1 << 1
b = z1 >> 1
c = z1 << 2
d = z1 << 3
print("Der Wert von 42 << 1: " , a)
print("Der Wert von 42 >> 1: " , b)
print("Der Wert von 42 << 2: " , c)
print("Der Wert von 42 << 3: " , d)
z2 = 21
    
```

Abb. 7.5 Binäre Verschiebung

```
Der Wert von 42 << 1: 84
Der Wert von 42 >> 1: 21
Der Wert von 42 << 2: 168
Der Wert von 42 << 3: 336
Der Wert von 21 >> 1: 10
>>>
```

```
print("Der Wert von 21 >> 1: " , z2 >> 1)
```

Die Ausgabe (Abb. 7.5) belegt noch einmal die oben getroffenen Erklärungen. Beachten Sie aber besonders die letzte Ausgabe. Denn 21 wird zwar halbiert, aber eben in Form einer Ganzzahldivision. Und deshalb kommt 10 und nicht 10.5 (beachten Sie, dass der Punkt zur Trennung verwendet wird – in der Schulmathematik wird das als 10,5 dargestellt) als Ergebnis heraus.

7.3.8.2 Bitweise arithmetische Operatoren

Betrachten wir nun die bitweisen arithmetischen Operatoren. Diese dienen der Verknüpfung von zwei Operanden auf Bit-Ebene. Das bedeutet, dass das Bit in einem Operanden mit dem Bit an der gleichen Stelle des zweiten Operanden verknüpft wird.

Um die Sache zu verstehen, schauen wir uns an, was auf Binärebene passiert, wenn zwei Bits aufeinandertreffen. Das bitweise UND funktioniert wie folgt:

- 0 & 0 = 0
- 0 & 1 = 0
- 1 & 0 = 0
- 1 & 1 = 1

Das Ergebnis zweier mit UND verknüpfter binärer Zahlen führt also dazu, dass nur bei den Bits, ana denen in beiden Operanden übereinstimmend eine 1 steht, nach der Operation wieder eine 1 zu finden ist.

Beispiel

Operation	Dezimaldarstellung	Binärdarstellung
	10	0000 1010
&	12	0000 1100
Ergebnis	8	0000 1000



Das bitweise ODER folgt nachstehendem Schema:

- $0 \mid 0 = 0$
- $0 \mid 1 = 1$
- $1 \mid 0 = 1$
- $1 \mid 1 = 1$

Hier gibt es nur dann 0, wenn die verglichenen Bits in den zwei Operanden beide eine 0 aufweisen. Beispiel:

Beispiel

Operation	Dezimaldarstellung	Binärdarstellung
	10	0000 1010
	12	0000 1100
Ergebnis	14	0000 1110



XOR hat das folgende Rechenschema:

- $0 \wedge 0 = 0$
- $0 \wedge 1 = 1$
- $1 \wedge 0 = 1$
- $1 \wedge 1 = 0$

Es wird eine 1 im Ergebnis gesetzt, wenn die Bits sich unterscheiden. Andernfalls ist das Resultat 0.

Beispiel

Operation	Dezimaldarstellung	Binärdarstellung
	10	0000 1010
\wedge	12	0000 1100
Ergebnis	6	0000 0110



Das bitweise NICHT (\sim) ist die Negation eines jeden Bits. Es wird in der Bitdarstellung eines Operanden jede 0 durch eine 1 und jede 1 durch eine 0 ersetzt. Die Wirkung dieses Vorgangs auf die Zahl besteht darin, dass ihr Vorzeichen umgedreht (aus einer positiven Zahl wird also eine negative) und der Wert 1 subtrahiert wird. Dieser bitweise Komplementoperator unterscheidet sich deutlich von den anderen bitweisen Operatoren. Er legt eine einstellige bitweise Operation fest. Das bedeutet, er hat nur einen – nachgestellten – Operanden, genau wie ein negatives Vorzeichen bei einer normalen Zahl.

Beispiel

Operation	Dezimaldarstellung	Binärdarstellung
\sim	10	0000 1010
Ergebnis	-11	1111 0101



Schauen wir uns die bitweisen Operationen in einem gemeinsamen Beispiel *op7.py* an:

```

z1 = 10
z2 = 12
a = z1 & z2
b = z1 | z2
c = z1 ^ z2
d = ~z1
print("Der Wert von 10 & 12: " , a)
print("Der Wert von 10 | 12: " , b)
print("Der Wert von 10 ^ 12: " , c)
print("Der Wert von ~ 10: " , d)

```

Die Ausgabe wird sein:

Der Wert von 10 & 12: 8

Der Wert von 10 | 12: 14

Der Wert von 10 ^ 12: 6

Der Wert von ~ 10: -11

Zu dem Listing sind aufgrund der vorbereitenden Ausführungen keine weiteren Erklärungen notwendig, aber Sie sollten die Ausgabe ansehen, die die obigen Ausführungen belegt.

7.4 Python-Sonderfälle bei der Auswertung

Es gibt in Python Syntaxkonstruktionen, die man so in anderen Sprachen gar nicht oder selten findet. In diesem Abschnitt lernen Sie ein paar kennen.

7.4.1 Verkettete Auswertung

Angenommen, Sie wollen in einem Vergleich mehrere Bedingungen prüfen. Zum Beispiel, ob der Wert einer Variablen größer einem Minimum und kleiner einem Maximum ist. Das wären dann zwei Vergleiche, die man etwa so testen kann (*op8.py*):

```
z1 = int(input("Geben Sie eine Zahl zwischen 0 und 10 ein:\n"))
print(0 < z1 and z1 < 11)
```

Es soll in dem Skript zuerst eine Zahl eingegeben werden. Dabei nutzen wir in der ersten Zeile die Built-in Function `int()`, um den String, der von `input()` geliefert wird, in eine Zahl zu wandeln. Beachten Sie, dass die Eingabe nicht gegen Fehleingaben (beispielsweise Buchstaben) abgesichert ist.

In der zweiten Zeile wird nun in zwei Bedingungen geprüft, ob die Eingabe größer als 0 ist (erster Vergleich) und ob die Eingabe kleiner als 11 ist (zweiter Vergleich). Die beiden Bedingungen liefern jeweils *True* oder *False* und werden mit dem Verknüpfungsoperator `and` verbunden, sodass genau ein Wahrheitswert das Ergebnis des gesamten Ausdrucks ist.

Wird beispielsweise 4 eingegeben, wird die Ausgabe *True* sein. Wird beispielsweise 12 eingegeben, ist die Ausgabe *False*.

Die Syntax in der zweiten Zeile wird man so oder so ähnlich in nahezu allen Programmiersprachen schreiben.

Nur Python erlaubt eine Verkürzung, in der auf den Verknüpfungsoperator und das mehrfache Notieren der zu testenden Variablen bzw. des Ausdrucks verzichtet wird. Das kann man dann so schreiben (*op9.py*):

```
z1 = int(input("Geben Sie eine Zahl zwischen 0 und 10 ein:\n"))
print(0 < z1 < 11)
```

Der zu testende Ausdruck wirkt quasi nach links und rechts. Sie können so auch längere oder auch komplexere Vergleiche formulieren, aber ich warne davor, dass Sie sich mit dieser Schreibweise einen Programmierstil angewöhnen, mit dem Sie in vielen anderen Sprachen Probleme bekommen werden.

7.4.2 Der Walross-Operator

Python ist mit dem Ziel angetreten, maximal einfach zu sein. Ein zentraler Aspekt dieses hehren Anspruchs basiert darauf, auf diverse Sprachelemente anderer Sprachen zu verzichten. Doch auch Python ist nicht davor gefeit, dass neue Versionen der Sprache Funktionalitäten bis hin zu neuen Syntaxstrukturen hinzufügen.

Bei Python 3.8 ist die Hinzufügung von **Zuweisungsausdrücken** eine solche syntaktische Erweiterung, da sie mit einem neuen Operator einhergeht. Dieser wird umgangssprachlich **Walross-Operator** (engl. "walrus operator") genannt, da die `:=`-Syntax den Augen und Stoßzähnen eines seitwärts gerichteten Walrosses ähnelt.

Dabei wird die Schreibweise in Form von `:=` vielen Programmierern bekannt vorkommen, denn sie gibt es in einigen älteren Programmiersprachen für die reine Zu-



```

Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct
2 2023, 13:03:39) [MSC v.1935 64 bit (AMD6
4)] on win32
Type "help", "copyright", "credits" or "li
cense()" for more information.
>>> a = 1
>>> (a := 2)
2
>>>

```

Ln: 6 Col: 0

Abb. 7.6 Zuweisungsoperator vs. Walross-Operator

weisung. Nur ist diese neue Syntax in Python explizit für die Zuweisung von Variablen inmitten von Ausdrücken – und nur da – gedacht. Oder kurz gesagt – sie verbindet eine Zuweisung direkt mit der Auswertung. Deshalb wird der `:=`-Operator offiziell als **Zuweisungsausdrucksoperator** bezeichnet. Ein weiterer Begriff, der für Zuweisungsausdrücke verwendet wird, ist **benannte Ausdrücke**.

Grundsätzlich hat sich in vielen alten Sprachen ein Problem ergeben, wenn man etwa in *if*-Bedingungen eine Zuweisung statt eines Vergleichs vorgenommen hat, die von einer Programmiersprache bei einer Auswertung als wahr oder falsch interpretiert wurde. Hier greift auch die schon angesprochene Problematik, dass Werte ungleich 0 dort als wahr gewertet wurden und versehentliche bzw. falsche Zuweisungen statt eines Vergleichs zu einem fehlerhaften Fortsetzen des Programms geführt haben.

Deshalb wurde diese gleichzeitige Zuweisung und Auswertung in vielen modernen Sprachen unterbunden. Auch Python würde bei Verwendung des klassischen Zuweisungsoperators innerhalb der Bedingung einer *if*-Anweisung oder eines ähnlichen Ausdrucks einen Fehler auslösen. Dabei ist der Fehler nicht das Problem, sondern die Lösung, weil oft logisch falsche Programmabläufe damit unterbunden werden. Dies ist die weiter oben schon angedeutete Entschärfung der Problematik in Python, obwohl in Python Werte ungleich 0 oder dem Leerstring als wahr verstanden werden.

Doch führt diese Einschränkung dazu, dass man bei einem Vergleich und einer Zuweisung mehrere Anweisungen für eine Situation braucht, die man mit dem Operator `:=` verkürzen kann. Er gibt im Gegensatz zu herkömmlichen Zuweisungsanweisungen den zugewiesenen Wert und nicht *True* oder *False* (es sei denn, *True* oder *False* wurden zugewiesen) zurück. In der Python-Shell können Sie die Wirkung im Vergleich zu dem normalen Zuweisungsoperator mit einem ganz einfachen Test sehen (Abb. 7.6).

Zeile 1 der Shell zeigt eine traditionelle Zuweisungsanweisung, bei der Variablen *a* der Wert 1 zugewiesen wird. Sie sehen aber explizit keine Ausgabe in der Folgezeile. Das bedeutet, dass die Zuweisung nicht mit einer gleichzeitigen Auswertung samt Rückgabe des zugewiesenen Werts gekoppelt ist.

Als Nächstes verwenden wir aber einen Zuweisungsausdruck (beachten Sie, dass dieser in der Situation zwingend in runde Klammern notiert werden muss), um der Variablen `a` den Wert 2 zuzuweisen. Dieser wird direkt zurückgegeben und damit in der Shell unmittelbar angezeigt.

Um jetzt den Nutzen dieses Operators genauer zu sehen, benötigen wir allerdings Syntaxstrukturen, die bisher noch nicht behandelt wurden. Deshalb verschieben wir konkrete Anwendungen und weitere Beispiele erst einmal.

7.5 Operatorvorrang und Ausdrucksbewertung

Operatorvorrang bedeutet die Beachtung der **Priorität** von Python-Operatoren. Denn diese sind entsprechend geordnet.

7.5.1 Die Priorität der Python-Operatoren

Die Operatoren in Python haben eine festgelegte Rangordnung, die immer dann angewandt wird, wenn in einem Ausdruck mehrere Operatoren verwendet und keine Klammern gesetzt werden (Klammern sind in der Rangfolge mit aufgenommen und haben die höchste Priorität). Als Beispiel kennen Sie bereits die Punkt-vor-Strich-Regel. Die Prioritäten sind von der höchsten Wertigkeit bis zur niedrigsten abwärts angegeben.

7.5.2 Bewertung von Ausdrücken

Wenn Sie einen Ausdruck mit unterschiedlichen Operatoren haben, muss Python entscheiden, wie Ausdrücke bewertet werden. Dazu wird zuerst der gesamte Ausdruck analysiert. Hier ist das Beispiel von Punkt-vor-Strich-Rechnung aus der Mathematik wieder sinnvoll. Python hält sich strikt an Regeln der Operatorvorrangigkeit. Je höher ein Operator priorisiert ist, desto eher wird er bewertet. Die multiplikativen Operatoren haben Vorrang vor den additiven Operatoren. Mit anderen Worten: In einem zusammengesetzten Ausdruck, der sowohl multiplikative als auch additive Operatoren enthält, werden die multiplikativen Operatoren zuerst bewertet. Dies ist übrigens einfach die Punkt-vor-Strich-Rechnung.

Beispiel

`a = b - c * d / e` ◀

Weil in dem Ausdruck die Operatoren `*` und `/` Vorrang haben, werden die beiden Unterausdrücke zuerst berechnet. Erst danach wird die Subtraktion ausgeführt. Immer wenn

Sie die Bewertungsreihenfolge von Operatoren in einem Ausdruck ändern wollen, müssen Sie Klammern benutzen. Klammern haben eine höhere Priorität als arithmetische Operatoren. Jeder Ausdruck in Klammern wird zuerst bewertet.

Beispiel

$$a = (b - c) * d / e \blacktriangleleft$$

Hier würde zuerst die Subtraktion von b und c durchgeführt, bevor die multiplikativen Operationen damit durchgeführt werden.

Die sogenannte **Operatorassoziativität** ist die einfachste der Bewertungsregeln, aber sie kann von Bedeutung sein. Es geht darum, dass alle arithmetischen Operatoren Ausdrücke in der Vorgabe von links nach rechts bewerten (assoziiieren). Das heißt, wenn der selbe Operator oder Operatoren gleicher Priorität in einem Ausdruck mehr als einmal auftauchen – wie beispielsweise der + -Operator bei dem Ausdruck $1+2+3+4+5$ – dann wird der am weitesten links erscheinende zuerst bewertet, gefolgt von dem rechts daneben und so weiter. Unterziehen wir folgende arithmetische Zuweisung einer näheren Betrachtung:

Beispiel

$$x = 1 + 2 + 3 + 4 + 5 \blacktriangleleft$$

In diesem Beispiel wird der Wert des Ausdrucks auf der rechten Seite des Gleichheitszeichens zusammengerechnet und der Variablen x auf der linken Seite zugeordnet. Für das Zusammenrechnen des Werts auf der rechten Seite bedeutet die Tatsache, dass der Operator + von links nach rechts assoziiert, dass der Wert von $1+2$ zuerst berechnet wird. Erst im nächsten Schritt wird zu diesem Ergebnis dann der Wert 3 addiert und so weiter, bis zuletzt die 5 addiert wird. Anschließend wird das Resultat dann der Variablen x zugewiesen. Immer, wenn Operatoren derselben Priorität mehrfach benutzt werden, können Sie die Assoziativitätsregel anwenden.



Kontrollstrukturen – Die Steuerung des Programmflusses

8

8.1 Was behandeln wir in diesem Kapitel?

Die Anweisungen in einem Programm können zwar sequenziell abgearbeitet werden. Aber das macht man nur bei sehr einfachen Programmen und Initialisierungen. So gut wie immer wird der Programmfluss durch gewisse Faktoren beeinflusst. Dazu dienen die Kontrollstrukturen, die wir in diesem Kapitel betrachten.

8.2 Was sind Kontrollstrukturen?

Kontrollstrukturen sind klassische Anwendungen von logischen oder vergleichenden Operatoren und spezielle Anweisungsschritte in einer Programmiersprache, mit denen ein Programmierer Entscheidungen über den weiteren Ablauf eines Programms oder Skripts vorgeben kann, wenn bestimmte Bedingungen eintreten. Das nennt man den **Programmfluss**.

Es gibt in der Regel drei Arten von Kontrollflussanweisungen:

- **Entscheidungsanweisungen.** Diese suchen aufgrund einer Bedingung einen Programmfluss heraus. Man spricht hier auch von einem **Zweig**, denn der Programmfluss verzweigt in verschiedene Varianten.
- **Schleifen** beziehungsweise **Iterationsanweisungen**. Diese wiederholen eine bestimmte Anzahl an Anweisungen.
- **Sprunganweisungen.** Diese verlassen eine Struktur.

8.3 Die Kontrollstrukturen in Python

In Python haben Sie weitgehend die gleichen Arten Kontrollstrukturen zur Verfügung, die es in fast allen Programmiersprachen in gleicher oder ähnlicher Form gibt. Kontrollstrukturen sind für pure Anfänger zwar nicht vollkommen einfach, aber wenn man mit irgendeiner Programmiersprache zumindest etwas Erfahrung hat, sind sie in anderen Programmiersprachen – insbesondere Python – nahezu trivial. Die Besonderheit bei Python ist nur, dass man **Anweisungsblöcke** beziehungsweise **Blockanweisungen** mit einem **Doppelpunkt** beginnt und die **Tiefe der Einrückung** den Block begrenzt (siehe Abschn. 5.3.1). Aus den meisten anderen Programmiersprachen ist man geschweifte Klammern oder explizite Schlüsselwörter wie *Begin* und *End* gewohnt. Das wurde in Python gestrichen, was die Syntax vereinfachen soll. Überhaupt wurde auch die Anzahl der erlaubten Kontrollstrukturen in Python entschlackt und auf Kontrollstrukturen verzichtet, die man aus den meisten anderen Programmiersprachen gewohnt ist. Auch hier greift wieder das Credo, dass Python einfach und klar sein soll. Es gibt dessen ungeachtet auch in Python grundsätzlich die oben genannten drei Arten von Kontrollflussanweisungen.

8.3.1 Entscheidungsanweisungen

Betrachten wir zuerst die **Entscheidungsanweisungen**. Davon gab es in Python bis zur Version 3.9 nur eine Variante – die *if*-Anweisung. In neuen Versionen von Python (ab 3.10) gibt es nun aber auch eine *match-case*-Anweisung als Alternative.

8.3.1.1 Die if-Entscheidungsanweisungen

Über die *if*-Kontrollstruktur entscheiden Sie, ob eine nachfolgende Anweisung oder ein Block auszuführen ist. Allgemeine Syntax ist folgende:

Beispiel

if Bedingung:

Anweisungen A ◀

Natürlich gibt es auch die Möglichkeit eines alternativen Programmflusses. Dieser wird wie eigentlich jede Programmiersprache mit *else* eingeleitet.

Beispiel für eine schematische Syntax:

Beispiel

if Bedingung:

Anweisungen A

else:

alternative Anweisungen ◀

- Die Bedingung muss in Python – im Gegensatz zu vielen anderen Programmiersprachen – **nicht** in Klammern gesetzt werden. Sie **darf** aber (meist) geklammert werden, was auch nicht in allen Sprachen erlaubt ist. Allgemein ist es in Python eher unüblich, die Bedingung zu klammern. Beachten Sie allerdings, dass in Python Klammern auch an einigen Stellen verboten (!) sind, an denen man es möglicherweise nicht erwartet – etwa bei Bedingungen einer *for*-Schleife.

Wenn eine Bedingung wahr ist, werden die Anweisungen A (entweder eine einzelne Anweisung oder eine Blockanweisung) ausgeführt, ansonsten die alternativen Anweisungen, wobei die *else*-Anweisung wie gesagt optional ist. Unter Bedingung versteht man einen Vergleich, der entweder wahr (*True*) oder falsch (*False*) sein kann, oder den booleschen Wert selbst. Zur Definition des Vergleichs verwendet man Vergleichsoperatoren und bei Bedarf logische Operatoren. Es gibt die *if*-Bedingung wie gesagt mit oder ohne nachgestellten *else*-Zweig. Wenn er da ist, kann darin wie oben eine Folge von alternativen Anweisungen stehen, die immer dann ausgeführt werden, wenn die Bedingung den Wert *False* liefert.

Wenn bereits der *if*-Zweig ausgeführt wurde, wird der *else*-Zweig nicht ausgeführt. Allgemein wird das Ausführen von einem Zweig der Konstruktion dazu führen, dass andere Zweige nicht ausgeführt werden. Wenn der *else*-Zweig fehlt, bewirkt die gesamte Struktur nur etwas, wenn die Bedingung den Wert *True* liefert. Im Falle des Wertes *False* passiert gar nichts.

Sie können als erste Anweisung hinter *else* wieder direkt eine *if*-Anweisung notieren, wenn der nachfolgende Programmfluss bedingt ist. Das wird in Python allerdings etwas „eigenartig“ notiert. Nach *else* folgt der Doppelpunkt, und die nachfolgende *if*-Anweisung muss eingerückt werden, etwa so:

Beispiel

else:

if Bedingung: ◀

- Es gibt in Python ein eigenes Schlüsselwort *elif*, was eine Kurzform von *else if* darstellt und besser anzuwenden ist.

So sieht das in einem vollständigen Listing aus (*kontroll1.py*):

```
a = 3
if a <= 2:
    print("Kleiner 3")
elif a == 3:
    print("Der Wert ist 3")
else:
    if a == 4:
        print("Der Wert ist 4")
```

Die Ausgabe wird „*Der Wert ist 3*“ sein.

- ▶ Sie dürfen nach dem Doppelpunkt hinter der Bedingung kein Semikolon notieren. In vielen Sprachen ist so etwas möglich. Das ist dann einfach eine leere Anweisung. In Python müssten Sie explizit die Anweisung *pass* notieren, wenn Sie so etwas machen wollten. Aber das ist meist wenig sinnvoll.

Wenn Sie eine *else:if*-Anweisung notieren, können Sie danach auf der gleichen Ebene wie *else* keine weitere *else*-Anweisung notieren. Nach *elif* hingegen können weitere *elif* oder auch eine *else*-Anweisung auf der gleichen Ebene folgen. Diese Aussage ist bewusst etwas „mysteriös“ formuliert, um eine Besonderheit von Python wieder sehr deutlich zu machen. Und diese ist so wichtig, dass wir diese Situation detailliert ausarbeiten.

Betrachten Sie den folgenden Quellcode:

```
a = 3
if a <= 2:
    print("Kleiner 3")
elif a == 3:
    print("Der Wert ist 3")
else:
    if a == 4:
        print("Der Wert ist 4")
else:
    print("Anderer Wert");
```

Es wurde im Vergleich zu dem vorherigen Beispiel einfach ein *else*-Zweig hinzugefügt. Aber wenn Sie das Beispiel ausführen wollen, erhalten Sie einen Fehler (Abb. 8.1).

Was ist das Problem? Ist *else* grundsätzlich nach einer vorangehenden *else*-Anweisung verboten, auch wenn diese mit einem direkten *if* danach als bedingt notiert wurde und eigentlich eine *elif*-Anweisung darstellt?

Wir sind gerade an einer Stelle, die in vielen Programmiersprachen etwas kritisch ist. Es geht um die Frage, zu welchem *if* das letzte *else* gehört. Das Stichwort ist „ver-schachtelte *if*-Anweisungen“.

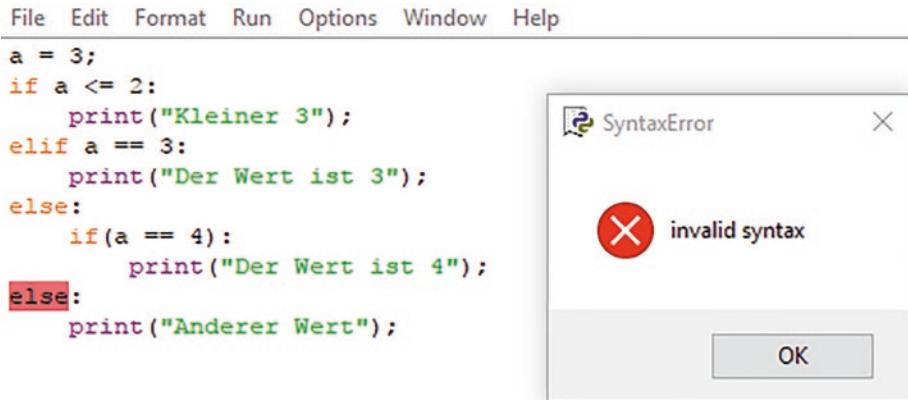


Abb. 8.1 Da stimmt etwas mit else nicht

Das *else* könnte zu dem *if* in Zeile 2 (dem Beginn der gesamten Struktur) oder aber dem *if* in Zeile 7 (nach dem *else*) gehören.

Python geht ziemlich radikal vor und lässt hier keinen Spielraum zu, was auch Teil des Konzepts der Einfachheit und Klarheit ist. Um die offene Frage aber zu lösen, machen wir vorher noch einen weiteren Fehler. Betrachten Sie diese geänderte Variante:

```
a = 3
if a <= 2:
    print("Kleiner 3")
elif a == 3:
    print("Der Wert ist 3")
else:
    if a == 4:
        print("Der Wert ist 4")
    else:
        print("Anderer Wert")
```

Auch jetzt scheitert die Ausführung (Abb. 8.2). Aber was ist denn der Unterschied der beiden (bisher noch fehlerhaften) Quellcodes? Die **Einrückung** von *else* und dem zugehörigen Block! Und da liegt der Hund begraben.

Das ist die korrekte Variante des Quellcodes (*kontroll2.py*):

```
a = 3
if a <= 2:
    print("Kleiner 3")
elif a == 3:
    print("Der Wert ist 3")
```

```

if a <= 2:
    print("Kleiner 3");
elif a == 3:
    print("Der Wert ist 3");
else:
    if(a == 4):
        print("Der Wert ist 4");
    else:
        print("Anderer Wert");

```

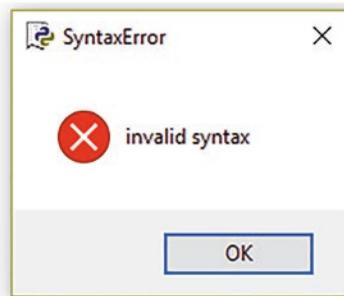


Abb. 8.2 Da stimmt immer noch etwas nicht mit else

```

else:
    if a == 4:
        print("Der Wert ist 4")
    else:
        print("Anderer Wert")

```

Das abschließende *else* muss in der gleichen Spalte wie das vorangehende *if* stehen. Und damit ist die Frage von oben beantwortet, wozu das *else* gehört. Es gehört nicht zum einleitenden *if*, sondern es kann nur zu dem *if* notiert werden, das zu der *else:if*-Struktur gehört. Wie angedeutet, ist diese Zuordnung zwar einfach und klar, aber dennoch etwas diffizil, zumal *elif* in die gleiche Spalte notiert wird wie das einleitende *if*.

Spielen wir noch ein kurzes Beispiel mit dieser Entscheidungsstruktur durch (*kontrol3.py*):

```

erg = int(input("Geben Sie eine Zahl zwischen 1 und 10 ein:\n"))
print(f"Sie haben {erg} eingaben! ")
if erg == 1:
    print("The Lumberjack-Song")
    print("Holzfäller-Lied")
elif erg == 2:
    print("The Funniest Joke in the World")
    print("Der tödlichste Witz der Welt")
elif erg == 3:
    print("Dirty Hungarian Phrasebook")
    print("Versautes ungarisch-englisches Wörterbuch")
    print("(English As She Is Spoke)")
elif erg > 3 and erg <= 10:
    print("*** Gerade in Arbeit ***")
else:
    print("Dazu hat Monty Python nichts zu sagen")

```

```
Geben Sie eine Zahl zwischen 1 und 10 ein:  
2  
Sie haben 2 eingegeben!  
The Funniest Joke in the World  
Der tödlichste Witz der Welt  
>>>  
===== RESTART: F:/PythonQuellcodes/kap  
Geben Sie eine Zahl zwischen 1 und 10 ein:  
3  
Sie haben 3 eingegeben!  
Dirty Hungarian Phrasebook  
Versautes ungarisch-englisches Wörterbuch  
(English As She Is Spoke)  
>>>  
===== RESTART: F:/PythonQuellcodes/kap  
Geben Sie eine Zahl zwischen 1 und 10 ein:  
5  
Sie haben 5 eingegeben!  
*** Gerade in Arbeit ***  
>>>  
===== RESTART: F:/PythonQuellcodes/kap  
Geben Sie eine Zahl zwischen 1 und 10 ein:  
12  
Sie haben 12 eingegeben!  
Dazu hat Monty Python nichts zu sagen  
>>>  
===== RESTART: F:/PythonQuellcodes/kap  
Geben Sie eine Zahl zwischen 1 und 10 ein:  
1  
Sie haben 1 eingegeben!  
The Lumberjack-Song  
Holzfäller-Lied  
>>>
```

Abb. 8.3 Alle möglichen Programmflüsse wurden ausprobiert

In dem Programm soll ein Anwender eine Zahl zwischen 1 und 10 (inklusive) eingeben. In dem Beispiel wird also eine Zahl zwischen 0 und 10 entgegengenommen und dieser Wert der Variablen *erg* zugewiesen. Dabei muss die Eingabe von einem String in eine Zahl umgewandelt werden. Das Programm wählt dann in Abhängigkeit davon einen der möglichen Zweige des Programmflusses aus (Abb. 8.3).

Zuerst wird der Wert der eingegebenen Zahl wieder ausgegeben. In der folgenden *if*-Konstruktion wird überprüft, ob der Wert größer als 10 ist und in Abhängigkeit davon eine entsprechende Meldung ausgegeben. Ist diese Bedingung nicht erfüllt, wird die nächste Überprüfung ausgeführt. In Abhängigkeit davon wird dann gegebenenfalls eine entsprechende Meldung ausgegeben. Oder es geht zur nächsten Überprüfung etc.

Es gibt aber jetzt noch ein paar bemerkenswerte Stellen im Quellcode, auf die ich Ihre Aufmerksamkeit lenken möchte:

- In einem Zweig können mehrere Anweisungen stehen, aber auch nur eine Anweisung. In dem Beispiel sehen Sie die verschiedenen Varianten. Der Block geht immer genau so weit, wie neue Anweisungen in der Spalte des Blocks beginnen. Es gibt keine explizite Anweisung für das Ende des Blocks.
- Wenn man mehrere Dinge in einer Bedingung überprüfen will, muss man diese mit einem Verknüpfungsoperator verbinden. Es muss immer genau eine Bedingung formuliert werden, die entweder *True* oder *False* liefert. In dem Beispiel kommt *and* zum Einsatz.
- Das Programm ist nicht gegen Fehleingaben abgesichert, um es nicht zu komplex werden zu lassen.¹ Wenn keine Zahl eingegeben wird, gibt es einen Fehler. Es wird dann aber auch nicht der abschließende *else*-Zweig erreicht.

8.3.1.2 Die **match-case**-Anweisung und Pattern-Matching

Für viele Jahre wurde es als ein Highlight bzw. Vorteil von Python propagiert, dass es nur eine Entscheidungsanweisung gibt. Der Verzicht auf sonst übliche Alternativen in anderen Sprachen wurde als Errungenschaft gefeiert, eine einfache und klare Sprache bereitzustellen und damit einen verständlichen und einheitlichen Programmierstil zu erzwingen.

Nun stellt sich die Frage, warum man in Python 3.10 mit der Einführung der neuen *match-case*-Anweisung für eine kompaktere Mehrfachauswahl diese Argumentation aufgibt?

Es gibt einige Gründe, die mehr oder weniger schwer wiegen.

- Gerne wird angeführt, dass es eben in den meisten anderen Programmiersprachen Anweisungen wie *match-case* (meist *switch-case* von der Syntax) gibt und Umsteiger diese gerne nutzen wollen bzw. ein Umstieg aus anderen Sprachen leichter fällt.
- Auch die leichtere Portierbarkeit von Quellcode einer anderen Sprache in Python wird als Argument angeführt.
- Stärker wiegt aus meiner Sicht das Argument, dass man mit dieser Form einer Entscheidungsstruktur Mehrfachauswahl teils kompakter schreiben kann als mit *if-elif-else*.

Dennoch rechtfertigt für mich persönlich keines der genannten Argumente bereits die „Aufblähung“ der Syntax von Python um ein zusätzliches Sprachkonstrukt. Aber ich habe das Hauptargument noch nicht genannt – das sogenannte **Pattern-Matching**. Dessen Möglichkeiten rechtfertigen auch für mich letztendlich die Einführung der neuen Syntax.

Doch jetzt müssen wir erst einmal diese neuen Anweisungen konkret sehen und beginnen mit der Idee, der grundsätzlichen Syntax und ein paar ganz einfachen Beispielen.

¹Außerdem haben wir noch nicht alle Techniken zur Verfügung, um das sinnvoll zu machen.

Um ganz einfach zu beginnen – die Python-Anweisung *match-case* wurde grundsätzlich eingeführt, um das Arbeiten mit mehreren Bedingungen in einer klareren und strukturierteren Weise zu ermöglichen. Angenommen, es gibt eine Variable, und auf den möglichen Werten dieser Variable basierend sollen verschiedene Aktionen ausgeführt werden. Die *match-case*-Anweisung erlaubt es meist kompakter als *if-elif-else*, verschiedene Fälle (Werte) der Variablen zu überprüfen und für jeden Fall eine bestimmte Aktion auszuführen.

Hier sehen Sie erst einmal die schematische Syntax:

Beispiel

Match Testwert:

```
case Testfall1:  
    Anweisungen  
case Testfall2:  
    Anweisungen  
...  
case Testfall_n:  
    Anweisungen  
case _:  
    alternative Anweisungen ◀
```

Sie haben einen Testwert vorliegen und können nacheinander verschiedene Testfälle angeben, um Übereinstimmungen zu finden.

Beachten Sie, dass das optionale *case _* für *other* (in vielen anderen Sprachen *default*) steht, was Sie auch nutzen können. Beim Unterstrich handelt es sich genaugenommen um eine Wildcard, worauf wir bei der Behandlung des Pattern-Matchings noch eingehen. Das entspricht dem *else*-Fall bei einer *if*-Anweisungen. In einfacheren Worten: *match-case* hilft, sehr kompakt und übersichtlich eine Entscheidung zu treffen und verschiedene Dinge basierend auf den möglichen Werten einer Variable zu tun. Auch wenn es oft mittels *if-elif-else* ebenso gut möglich ist.

Hier ist erst einmal ein sehr einfaches und grundlegendes Beispiel für die Anwendung von *match-case* (*match1.py*):

Beispiel

```
x = 2  
match x:  
case 1:  
    print("x ist gleich 1")  
case 2:  
    print("x ist gleich 2")
```

```

case 3:
print("x ist gleich 3")
case _:
print("x hat einen anderen Wert") ◀

```

In diesem Beispiel wird *match x* verwendet, um den Wert der Variable *x* (eine Zahl) mit verschiedenen Fällen zu vergleichen. Je nachdem, welchen Wert *x* hat, wird die entsprechende Aktion ausgeführt. Wenn keiner der Fälle zutrifft, wird der Zweig *case _* verwendet, um eine Standardaktion auszuführen.

Verdeutlichen wir uns die Anweisung mit weiteren konkreten Beispielen und bauen erst einmal auf dem ersten Beispiel mit einer Abwandlung auf (*match2.py*).

Beispiel

```

b = input("Geben Sie an, was Sie machen wollen\nx:\t Ende\ns:\tSpei-
chern\nnd:\tDrucken\n").lower()
match b:
case 'x':
print('Programm verlassen.')
case 's':
print('Daten speichern.')
case 'd':
print('Drucken.')
case _:
print('Kein Treffer.') ◀

```

In diesem Beispiel wird mit einer freien Benutzereingabe gearbeitet und auf Übereinstimmung mit einem spezifischen Zeichen (also einem Character bzw. String der Länge 1) getestet. Sonst ist das Beispiel weitgehend identisch zum ersten Beispiel.

Betrachten wir diese weitere Abwandlung (*match3.py*).

Beispiel

```

b = input("Geben Sie an, was Sie machen wollen:\nEnde:\t\tEnde\
nSpeichern:\tSpeichern\nDrucken:\tDrucken\n").lower()
match b:
case 'ende':
print('Programm verlassen.')
case 'speichern':
print('Daten speichern.')
case 'drucken':
print('Drucken.')
case other:
print('Kein Treffer.') ◀

```

Die Testfälle können auch beliebige Strings sein. Das war in anderen Programmiersprachen bei vergleichbaren Syntaxstrukturen lange Zeit nicht möglich, wobei diese Art der Anwendung mittlerweile Usus ist.

Doch dieses dritte Beispiel geht bereits in die Richtung des Pattern-Matchings, was letztendlich die *match-case*-Anweisung erst wirklich begründet und vor allen Dingen sehr spannend macht. Nur benötigen wir für das effektive Verständnis von Pattern-Matching sequenzielle Datenstrukturen. Deshalb soll die detaillierte Behandlung verschoben werden, bis dieses Thema durchgesprochen wurde.

8.3.2 Iterationsanweisungen

Kommen wir nun zu den **Schleifen** beziehungsweise **Iterationsanweisungen** in Python. Diese wiederholen Anweisungen so lange, bis ein Abbruchkriterium erfüllt ist.

- Schleifen lassen sich auch verschachteln. Das bedeutet, dass Sie innerhalb einer Schleife eine weitere Schleife ausführen.

8.3.2.1 Die while-Schleife

Die *while*-Schleife führt am Anfang eines Blocks eine Prüfung einer Bedingung durch. Die nachfolgenden Anweisungen werden so lange durchlaufen, bis die Bedingung nicht mehr erfüllt ist. Die Syntax sieht schematisch so aus:

Beispiel

while Bedingung:

Anweisungen ◀

Wenn eine Bedingung beim Erreichen der Schleife nicht mehr erfüllt ist, werden die Anweisungen im Block überhaupt nicht durchgeführt. Normalerweise kontrolliert man die Anzahl der Wiederholungen mit einer **Zählvariablen**.

Die *while*-Schleife benötigt (in der Regel) in ihrem Inneren eine Möglichkeit, mit der die Schleife abgebrochen oder die Voraussetzungen für einen neuen Durchlauf verändert werden können (etwa durch Änderung der Zählvariablen).

Andernfalls erzeugen Sie eine **Endlosschleife**.

- Eine **Endlosschleife** ist eine nicht endende Abarbeitung der Anweisungen im Inneren der Schleife. Das kann zwar manchmal sinnvoll sein, ist aber normalerweise nicht gewünscht. Allgemein sind Endlosschleifen ohne ein explizites Abbruchkriterium selten sinnvoll. Man kann allerdings Endlosschleifen sinnvoll verwenden, wenn man in der Schleifenbedingung das Abbruchkriterium nicht angeben will oder kann, aber im Inneren der Endlosschleife ein Abbruchkriterium formuliert.

Der Abbruch einer *while*-Schleife erfolgt normalerweise darüber, dass die in der Bedingung geprüfte Variable im Inneren so verändert wird, dass irgendwann die Bedingung nicht mehr erfüllt ist. Sie sehen es in der nachfolgenden Übung.

```
i = 0;
print("Zahl\t\t2. Pot\t\t3. Pot\t\t4. Pot")
while i < 9:
    i+=1
    print(i ,"\t|\t", i ** 2, "\t|\t", i ** 3, "\t|\t", i ** 4)
print("Schleife beendet")
```

Das Beispiel erzeugt eine Ausgabe mit mehreren Zeilen. Die einzelnen Zeilen samt Inhalt werden dynamisch mit Python erzeugt. Zuerst ist der Wert der Zählvariablen *i* für unsere Schleife 0. Die Bedingung in der *while*-Überprüfung ist also wahr und die Anweisungen werden ausgeführt. Nach der Erhöhung des Wertes werden der Wert des Zählers und die 2., 3. und 4. Potenz der Zahl ausgegeben.

Das Durchlaufen der Schleife wird so lange fortgesetzt, bis die Variable den Wert 10 erreicht. Dann ist die Bedingung in der *while*-Überprüfung falsch und die Anweisungen der Schleife werden nicht mehr ausgeführt. Es geht weiter mit der ersten Anweisung hinter der Schleife.

Beachten Sie auch den Einsatz von Escape-Sequenzen (Tabulator), um die Ausgabe aufzubereiten (Abb. 8.4). Diese kommen auch in der ersten Ausgabezeile zum Einsatz, die außerhalb der Schleife die „Überschrift“ der tabellenartigen Ausgabe der Werte generiert.

- ▶ Eine Schleife kann ebenfalls durch den Einsatz von einer Sprunganweisung wie *break* beendet werden (aus dem Block der Schleife heraus). Nach dem Sprung aus der Schleife mit *break* geht es im Programmfluss hinter der Schleife weiter. Das ist insbesondere in Verbindung mit dem Prüfen einer Bedingung über *if* oft sinnvoll. Diese Bedingung im Inneren ist in der Regel nicht mit der Bedingung im Schleifenkopf identisch. Dazu kommen wir gleich.

Zahl	2. Pot	3. Pot	4. Pot
1	1	1	1
2	4	8	16
3	9	27	81
4	16	64	256
5	25	125	625
6	36	216	1296
7	49	343	2401
8	64	512	4096
9	81	729	6561
Schleife beendet			
>>>			

Abb. 8.4 Ausgabe der Schleife

8.3.2.2 Die for-Schleife

Eine weitere Schleifenform wird über das Schlüsselwort *for* eingeleitet. Die allgemeine Syntax sieht schematisch so aus:

Beispiel

for x in Datenstruktur:

Anweisungen ◀

Wie Sie schon an der schematischen Beschreibung sehen, ist diese Schleife zur Anwendung bei Datenstrukturen gedacht. Diese Datenstrukturen zählen in Python zu den Highlights. Schleifen mit *for* sind insbesondere im Zusammenhang mit Arrays, Mengen, Listen, Tupeln etc. von Bedeutung und werden auch in dem Zusammenhang erst mit Beispielen unterfüttert, da wir diese Datenstrukturen noch nicht behandelt haben. Aber eine Datenstruktur kennen wir schon – den String. Denn das ist im Sinne von Python eine Datenstruktur. Das wäre eine kleine Anwendung der Schleife (*kontroll5.py*):

```
text = "The Funniest Joke in the World"
zv = 0
for x in text:
    zv+=1
    print("Zeichen ", zv, ":\t", x)
print("Schleife beendet")
```

Die Datenstruktur *text* (ein String aus einer gewissen Anzahl an Zeichenliteralen) wird Zeichen² für Zeichen durchlaufen. Das jeweilige Zeichen wird dann in einer eigenen Zeile ausgegeben (Abb. 8.5). Wenn das letzte Element erreicht und verarbeitet wurde, wird die Schleife beendet.

- ▶ Die Zählvariable *zv* ist explizit nicht notwendig, da die Bedingung zum Abbruch (im Gegensatz zur *while*-Schleife) nicht daran gekoppelt ist. Eine solche Zählvariable wird aber oft auch bei der Schleifenform „mitgeführt“, wenn man eine gewisse Logik daran koppeln will. In dem Beispiel wird damit einfach gezählt, welches Element gerade in Arbeit ist.³

²Genauer ist „Element“.

³Was in dem Fall äquivalent zur Anzahl der bisherigen Schleifendurchläufe ist.

Abb. 8.5 Die for-Schleife

```

Zeichen 1 : T
Zeichen 2 : h
Zeichen 3 : e
Zeichen 4 :
Zeichen 5 : F
Zeichen 6 : u
Zeichen 7 : n
Zeichen 8 : n
Zeichen 9 : i
Zeichen 10 : e
Zeichen 11 : s
Zeichen 12 : t
Zeichen 13 :
Zeichen 14 : J
Zeichen 15 : o
Zeichen 16 : k
Zeichen 17 : e
Zeichen 18 :
Zeichen 19 : i
Zeichen 20 : n
Zeichen 21 :
Zeichen 22 : t
Zeichen 23 : h
Zeichen 24 : e
Zeichen 25 :
Zeichen 26 : W
Zeichen 27 : o
Zeichen 28 : r
Zeichen 29 : l
Zeichen 30 : d
Schleife beendet
>>>

```

8.3.3 Sprunganweisungen

Python stellt eine gewisse Anzahl an Sprunganweisungen zur Verfügung. Diese verlassen eine umgebende syntaktische Struktur und führen dazu, dass mit der direkten Anweisung hinter der syntaktischen Struktur weitergemacht wird. Der Begriff einer „syntaktischen Struktur“ ist recht abstrakt, aber eine solche syntaktische Struktur kennen Sie – die Schleifen. Allgemein wird eine Schleife beendet, wenn die überprüfte Bedingung nicht mehr erfüllt ist. Schleifen sind aber recht kritische Stellen in jedem Programm oder Skript. Es kann leicht zu Endlosschleifen oder ungewollten Durchläufen kommen. Oder Sie brauchen mehrere Bedingungen, bei denen eine Schleife oder ein Schleifendurchlauf zu beenden ist.

8.3.3.1 Abbruch mit break

Die Anweisung *break* verlässt eine Syntaxstruktur sofort, wenn diese Stelle im Quelltext erreicht wird. Das kann man in Schleifen nutzen (*kontroll6.py*).

```
i = 0
while True:
    i+=1
    print(i ,"\t:\t", i * i)
    if (i > 9): break
print("Schleife beendet")
```

Mit `while True:` wird in dem Beispiel eine Endlosschleife erzeugt. Aber mit `if (i > 9):break` wird diese verlassen, wenn die Bedingung erfüllt ist.

Beachten Sie, dass die Anweisung `break` unmittelbar dem Doppelpunkt folgt. Das kann man in Python dann machen, wenn nur genau eine Anweisung statt eines Anweisungsblocks notiert wird.

8.3.3.2 Fortsetzen mit `continue`

Neben `break` gibt es im Zusammenhang mit Schleifen noch die Sprunganweisung `continue`. Sie können damit an einer bestimmten Stelle innerhalb des Schleifenblocks unmittelbar den nächsten Schleifendurchlauf erzwingen und die nachfolgenden Anweisungen innerhalb des Schleifenblocks ignorieren (`kontroll7.py`).

```
i = 0
while i < 10:
    i+=1
    if i % 2 != 0: continue
    print(i)
print("Schleife beendet")
```

- ▶ Die Verwendung von `continue` findet man in der Praxis nicht sonderlich oft. Alternativ kann man fast immer Bedingungen so formulieren, dass man darauf verzichten kann.

Das Beispiel gibt nur die geraden Zahlen der Zählvariablen einer Schleife aus. Die erste Anweisung des Schleifenblocks wird für jeden Durchlauf ausgeführt. Danach wird unter Verwendung des Modulo-Operators und des Ungleich-Operators in einer `if`-Bedingung getestet, ob die Zählvariable ungerade ist. Falls der Test `True` liefert, wird `continue` ausgeführt und sofort ein weiterer Durchlauf der Schleife gestartet. Der Wert der Zählvariablen wird dabei natürlich um 1 erhöht. Die folgende Zeile im Block der Schleife wird dabei übersprungen. Falls die Zählvariable jedoch gerade ist, wird die Anweisung ausgeführt (Ausgabe von dem Wert der Zählvariablen).

8.3.3.3 Rückgabe mit `return`

Im Zusammenhang mit Funktionen gibt es die Sprunganweisung `return`. Diese verlässt eine Funktion und gibt in der Regel einen Rückgabewert zurück. Wir werden diese Sprunganweisung bei den Funktionen behandeln.

8.3.3.4 Unterbrechung mit `raise`

Die vierte Sprunganweisung bei Python nennt sich *raise*. Damit werfen Sie eine sogenannte Ausnahme aus, die den normalen Programm- beziehungsweise Skriptablauf unterbricht und zu einer Behandlungsroutine springt. Wir besprechen die Ausnahmebehandlung in einem späteren Kapitel mit fortgeschrittenen Python-Techniken.



Funktionen in Python – Modularisierung mit „Unterprogrammen“

9

9.1 Was behandeln wir in diesem Kapitel?

Mit Built-in Functions von Python als eine Art Unterprogramme des APIs haben wir schon mehrfach Berührung gehabt. Aber natürlich können Sie auch eigene Funktionen in Python erstellen, um Ihre Programme zu strukturieren und zu modularisieren. Wenn man von einigen Besonderheiten in Python absieht (siehe Abschn. 1.4), ist die Deklaration und Verwendung von eigenen Funktionen in Python sogar äußerst einfach.

9.2 In Python eigene Funktionen deklarieren – Das Schlüsselwort `def`

Die Deklaration einer eigenen Funktion beginnt in Python mit dem Schlüsselwort `def`. Dem Schlüsselwort folgen der Name der Funktion und runde Klammern, in die bei Bedarf Parameter eingefügt werden. Das ist in Python ganz so wie in nahezu allen Programmiersprachen, die Funktionen oder Methoden deklarieren lassen. Die Besonderheit in Python ist wieder der nun folgende Doppelpunkt, mit dem der Anweisungsblock eingeleitet wird.

Schematisch sieht die Deklaration einer Funktion so aus:

Beispiel

`def functionsname(Parameterliste):`

.... ◀

Ein Beispiel für die Deklaration einer Funktion ohne Parameter könnte so aussehen:

```
def ausgabe():
    print("Ein Kratzer?! Ihr Arm ist ab!")
    print("Nein, das stimmt nicht.")
```

9.2.1 Übergabewerte

Über **Funktionsparameter** oder kurz Parameter (manchmal auch Argumente genannt, wobei das eigentlich die Aufrufsituation beschreibt) übergibt man Werte an eine Funktion. Bei der Deklaration schreibt man in Python die Namen als kommataseparierte Liste hin.

- ▶ Die Parameter sind innerhalb der Funktion in Python **lokale** Variablen.¹

Hier wäre ein Beispiel mit Parametern, die dann in der Funktion als Variablen verwendet werden:

```
def multiplizieren(a, b):
    print(a * b)
```

In den meisten Programmiersprachen unterscheidet man nun im Wesentlichen zwischen zwei Formen der Parameterübergabe:

- Call-by-value
- Call-by-reference

9.2.1.1 Wertübergabe (Call-by-value)

Beim Aufruf der Funktion stellt bei Call-by-value jedes Argument nur einen **Wert** dar, der in der Funktion verwendet werden kann. Wenn hier eine Variable notiert wird, wird nur der Wert und nicht die Variable selbst übergeben. Es wird quasi eine Kopie der Variable übergeben.

9.2.1.2 Referenzübergabe (Call-by-reference)

Bei der Referenzübergabe wird eine Referenz auf die Speicheradresse (oft Pointer oder Zeiger genannt) übergeben. Man nutzt als Parameter dann nur Variablen – also einen Ausdruck, deren Adresse berechnet werden kann. Das bewirkt, dass sich alle

¹ Das ist in nahezu allen Programmiersprachen der Fall.

Änderungen innerhalb der Funktion an diesem Parameter auch im aufrufenden Teil des Programmes auswirken.

9.2.1.3 Was gilt in Python? Call-by-Object

In Python gilt von der Wirkung her Call-by-value. Es wird also eine Kopie übergeben und Änderungen in der Funktion verändern nicht die globale Variable. Nur ist das nicht ganz so trivial, denn Python benutzt einen Mechanismus, den man als „Call-by-object“ oder auch „Call-by-object-reference“ oder „Call-by-sharing“ bezeichnet. Beim Aufruf einer Funktion werden bei Python nicht Zeiger auf Variablen, sondern Zeiger auf die zugrunde liegenden Objekte übergeben. Insofern könnte man von einer Art Referenzübergabe sprechen. Zuerst verhält sich Python wie bei „Call-by-reference“, im Ergebnis jedoch wie „Call-by-value“, weil das Objekt ausgewertet wird. Der Vorteil ist, dass man beliebige Ausdrücke übergeben kann.

9.2.2 Rückgabewerte

Eine Funktion kann auch einen Rückgabewert liefern. Das ist ein Ergebnis. Der Rückgabewert wird mit dem Schlüsselwort *return* zurückgegeben. Das ist in der Regel (siehe Abschn. 1.4.4) die letzte Anweisung im Funktionsblock, sofern das nicht bedingt gemacht wird.

Schematisch sieht das so aus:

```
def functionsname(Parameterliste)
```

```
....Anweisungen  
return Ergebnis ◀
```

Hier wäre ein Beispiel mit Rückgabewert:

```
def multiplizieren(a, b):  
    return a * b
```

Eine Funktion ohne Rückgabewert wurde früher **Prozedur** genannt. In modernen Sprachen unterscheidet man sprachlich nicht mehr zwischen Funktionen mit oder ohne Rückgabewert.

9.3 Funktionen aufrufen

Die Deklaration von Funktionen ist nur die eine Seite der Medaille. Man muss sie erst aufrufen, damit sie aktiv werden. Funktionen werden dabei einfach über ihren Namen und gegebenenfalls Übergabewerte aufgerufen. Funktionsaufrufe sind also in dem Sinn „normale“ Anweisungen.

- Sie können in einer Funktion eine andere selbstdefinierte² Funktion aufrufen. Beachten Sie, dass diese aber **vor dem Aufruf** in einer anderen Funktion deklariert werden muss. Andernfalls erhalten Sie einen *NameError*.

Hier ist ein Python-Programm (*Funktionen1.py*) mit der Deklaration von vier Funktion und deren folgendem Aufruf in verschiedenen Konstellationen:

```
def multiplizieren(a, b):
    print(a * b)
def multi(a, b):
    return a * b
def ausgabe():
    print("Ein Kratzer?! Ihr Arm ist ab!")
    print("Nein, das stimmt nicht.")
def rechnen(a, b, c):
    if c == "*":
        return multi(a, b)
    elif c == "+":
        return a + b
    elif c == "-":
        return a - b
    elif c == "/":
        return a / b
ausgabe()
multiplizieren(6,7)
print(rechnen(7,6,"*"))
print(rechnen(22,20,"+"))
```

- Zuerst wird einfach eine Funktion *multiplizieren(a, b)* deklariert, welche die beiden übergebenen Parameter multipliziert und direkt ausgibt. Es gibt keinen Rückgabewert.
- Danach wird eine Funktion *multi(a, b)* deklariert, welche ebenso die übergebenen beiden Parameter multipliziert, aber nicht ausgibt. Stattdessen wird das Ergebnis der Multiplikation als Rückgabewert geliefert.
- Die dritte Funktion *ausgabe()* hat weder Parameter noch einen Rückgabewert. Hier wird einfach eine Ausgabe ausgeführt.
- Die vierte Funktion *rechnen(a, b, c)* zeigt den wahrscheinlich interessantesten Aufbau. Denn es gibt drei Parameter, und mit dem dritten Parameter wird der interne Programmfluss der Funktion gesteuert. Damit wird diese Funktion einen **bedingten Rückgabewert** liefern, denn die gewählte *return*-Anweisung hängt vom Wert des dritten Parameters ab. Was weiter interessant ist: Die Funktion ruft zur Berechnung der Multiplikation die Funktion *multi()* auf, die oberhalb dieser Stelle deklariert wurde.

²Und natürlich auch eine Standardfunktion des API.

Die weiteren Aufrufe der Funktionen werden nun sequenziell notiert und müssen keinesfalls in der Reihenfolge der Deklaration erfolgen. Letzteres ist zwar trivial, soll aber dennoch erwähnt werden.

Die Ausgabe des Programms sehen Sie in Abb. 9.1.

9.3.1 Stehen in Python global deklarierte Variablen in der Funktion zur Verfügung?

Es stellt sich bei Funktionen auch die Frage, ob global deklarierte Variablen in der Funktion zur Verfügung stehen. Die Antwort ist nicht ganz trivial und muss genauer untersucht werden. Denn es hängt von der konkreten Situation ab.

9.3.1.1 Deklaration vor dem Funktionsaufruf

Sofern globale Variablen **vor dem Aufruf** der Funktion deklariert wurden, stehen sie in der Funktion zur Verfügung, aber nur als Kopie. Zumindest wirkt es so, denn Änderungen an den Variablen in der Funktion wirken sich nicht auf globale Variablen aus. Tatsächlich „zaubert“ Python etwas im Hintergrund, denn globale Variablen lassen sich erst einmal in Funktionen nur lesen, aber nicht ändern. Sobald man den Bezeichner einer globalen Variablen nutzt und einen Wert zuweist, wird automatisch eine lokale Variablen angelegt, die die gleichbenannte globale Variable verdeckt. Das werden wir uns noch in einem anderen Zusammenhang ansehen (Abschn. 9.7.1). Der folgende Code (*GlobaleVariablen.py*) funktioniert:

```
a = 42
def ausgabe():
    print(a)
ausgabe()
print(a)
```

Die Ausgabe ist das:

```
42
42
```

In der Funktion und außerhalb steht die Variable *a* zur Verfügung, denn sie wurde vor dem Funktionsaufruf deklariert. Aber auch das funktioniert:

Abb. 9.1 Die Ausgabe des Programms

```
Ein Kratzer?! Ihr Arm ist ab!
Nein, das stimmt nicht.
42
42
42
>>>
```

```
def ausgabe():
    print(a)
a = 42
ausgabe()
print(a)
```

Zwar steht die **Deklaration** der Funktion *ausgabe()* **vor** der Deklaration der globalen Variable, aber der **Aufruf** der Funktion **dahinter**.

9.3.1.1.1 Das Schlüsselwort **global**

Wie gesehen, stehen globale Variablen in der Funktion nur als Kopie bzw. zum Auslesen des Werts der globalen Variablen zur Verfügung, und Änderungen an den Variablen in der Funktion wirken sich nicht auf globale Variablen aus. Jetzt gibt es aber noch das Schlüsselwort *global*, mit dem Sie in der Funktion bekanntgeben, dass explizit auf die globale Variable zugegriffen wird – ggfls. auch schreibend. Das ist ein Verhalten, wie man es etwa in PHP auch kennt. Beispiel (*GlobaleVariablen2.py*):

```
a = 42
def ausgabe():
    global a
    print(a)
    a = 3
    print(a)
ausgabe()
print(a)
```

Das funktioniert, weil in der Funktion *a* global ist, und Änderungen daran wirken sich auf die globale Variable aus:

```
42
3
3
```

9.4 Rekursion

Selbstverständlich unterstützt Python auch sogenannte **rekursive Funktionsaufrufe** – auch **Selbstaufrufe** genannt. Das Adjektiv „rekursiv“ stammt vom lateinischen Verb „recurrere“, was „zurücklaufen“ bedeutet. Rekursive Aufrufe haben eine gewisse Ähnlichkeit mit Schleifen, da auch damit Anweisungen wiederholt werden, und sie sind nicht ganz trivial. Beachten Sie aber, dass rekursive Aufrufe zwar ähnlich wie Schleifen funktionieren, aber die rekursiven Funktionsaufrufe intern auf dem sogenannten **Stack** (Stapel) des Computers abgelegt und von dort wieder genommen werden, während Schleifen damit nichts zu tun haben und über einen Speicherbereich mit Namen **Heap**

(Haufen) arbeiten. Dieser Unterschied kann sich auf die Performance auswirken, aber vor allen Dingen ist die Reihenfolge einer Verarbeitung von Bedeutung. Denn während eine Schleife chronologisch bei jedem Durchlauf Anweisungen ausführt, erfolgt die Abarbeitung bei einer Rekursion nach dem **FILO**-Prinzip.

FILO steht für „first in last out“. Das klingt jetzt vielleicht etwas seltsam, aber wenn Sie an einen realen Kartenstapel denken, wird die Sache leicht klar. Stellen Sie sich vor, Sie legen Karten einzeln auf einem Kartenstapel aufeinander. Wenn Sie die Karten wieder einzeln wegnehmen, nehmen Sie sie von oben weg, bis die letzte Karte entnommen wurde und der Stapel weg ist. Die Karte, die als erstes („first in“) auf den Stapel gelegt wurde, ist also vor dem „Abarbeiten“ des Stapels ganz unten und wird als letzte Karte verarbeitet („last out“).

- ▶ **Vorsicht** Eine rekursive Funktion muss terminieren, damit man sie in einem Programm benutzen kann. Eine rekursive Funktion terminiert (endet), wenn mit jedem Rekursionsschritt das Problem reduziert wird und sich in Richtung einer Abbruchbedingung bewegt. Das ist der Fall, wenn die Funktion sich nicht mehr selbst aufruft. Sollten zu viele Aufrufe auf den Stack gelegt werden (etwa weil man kein vernünftiges Abbruchkriterium für die Anzahl der Selbstaufrufe definiert hat), wird man in Python einen *RecursionError* erhalten. Das erkennt man an einer Meldung der Art:

RecursionError: maximum recursion depth exceeded in comparison.

Nun ist die FILO-Reihenfolge bei Rekursion nicht immer von Bedeutung. Im einfachen Fall ruft sich eine Funktion im Laufe der Abarbeitung bloß selbst wieder auf und macht irgendetwas, das vom vorherigen Aufruf nicht abhängt, oder liefert keine Ausgabe, die den aktuellen Status anzeigt. Aber wenn ein Aufruf von dem vorherigen abhängt, dann ist die Reihenfolge natürlich massiv von Bedeutung. Gerade wenn eine rekursive Funktion einen Rückgabewert liefert, ist das FILO-Prinzip wichtig. Die nachfolgenden Beispiele zeigen sowohl die einfache Situation von unabhängigen Aufrufen als auch von einfachen abhängigen Aufrufen als auch einen klassischen Algorithmus für Rekursion, der mit Rückgabewerten der rekursiven Funktion arbeitet – die Fakultätsberechnung. Diese ist meist das Standardbeispiel für Rekursion beim Erlernen einer Programmiersprache.

Beispiel 1 (*Rekursion.py*):

```
i = 0
def fkt():
    global i
    i += 1
    if i < 10:
        fkt()
fkt()
print(i)
```

Für den Rekursivauftrag, bei dem die Reihenfolge der Abarbeitung im Grunde uninteressant ist, definieren wir eine globale Variable i , die rekursiv erhöht werden soll. Die Funktion $fkt()$

- nutzt das Schlüsselwort *global*, um auf die globale Variable i zuzugreifen,
- erhöht i um den Wert 1 und
- ruft sich dann wieder auf, bis die globale Variable i den Wert 10 erreicht hat.

Danach springt der Programmfluss hinter den Aufruf der Funktion $fkt()$, und der Wert der globalen Variable i , der zu dem Zeitpunkt 10 ist, wird ausgegeben. Grundsätzlich ist also an der Stelle nur die Tatsache interessant, dass sich eine Funktion wieder selbst aufruft und es am Ende ein Ergebnis gibt.

Nun wollen wir aber ohne die globale Variable arbeiten und der Funktion einen Parameter übergeben. Beachten Sie diese Variante des Listings (*Rekursion2.py*):

```
def fkt(i):
    if i < 10:
        i += 1
        fkt(i)
    print(i)
fkt(0)
```

In diesem Beispiel wird der Funktion $fkt()$ beim ersten Aufruf der Wert 0 übergeben. Da damit die Bedingung der *if*-Anweisung im Inneren der Funktion erfüllt ist, werden die Erhöhung und der Selbstaufruf durchgeführt. Damit bekommt der zweite Aufruf der Funktion aber den Wert 1 übergeben. Die Selbstaufrufe werden so lange auf den Stack gelegt, bis die Bedingung der *if*-Anweisung nicht mehr erfüllt ist. Erst dann wird die Anweisung hinter dem Block der *if*-Anweisung ausgeführt. Diese gibt den Wert der lokalen Variable i aus und das ist zu dem Zeitpunkt der Wert 10.

Beachten Sie, dass bis zu der Stelle noch keine weiteren Ausgaben erfolgt sind. Diese Anweisungen zur Ausgabe liegen alle noch auf dem Stack. Und erst jetzt werden die auf dem Stack abgelegten Funktionsaufrufe nach dem FILO-Prinzip abgearbeitet. Das bedeutet, zuerst wird die Anweisung $print(10)$, dann $print(9)$ bis hin zu $print(1)$ ausgeführt (Abb. 9.2).

Nun waren die beiden letzten Beispiele etwas „künstlich“. Aber mit der Berechnung der Fakultät haben wir eine reale mathematische Fragestellung. Bei der Berechnung der Fakultät in der folgenden Funktion $fac()$ sollten wir genauer hinsehen. Die Funktion besitzt einen Übergabeparameter und einen Rückgabewert, und da werden sich bei rekursiven Aufrufen spannende Dinge abspielen. Das ist das – vielleicht überraschend – knappe Listing zur rekursiven Berechnung der Fakultät:

Abb. 9.2 Rekursion

```

10
10
9
8
7
6
5
4
3
2
1
>>> |
```

```

def fac(n):
    if n == 1:
        return 1
    else:
        return n * fac(n-1)
print(fac(5))
```

Wenn die Funktion erstmals aufgerufen wird (*fac(5)*), hat der Parameter *n* den Wert 5. Nun wird im Inneren der Funktion geprüft, ob der Wert von *n* 0 ist. Das ist nicht der Fall, und der *else*-Zweig wird „ausgeführt“, wobei „ausgeführt“ hier eben bedeutet, dass der Aufruf der Funktion auf den Stack gelegt wird, mit dem Wert 5 für den Parameter. Und der *else*-Zweig sorgt dafür, dass die nächste „Karte“ auf dem Stapel der Aufruf von *fac(4)* ist (*fac(n-1)*). Dann kommen *fac(3)*, *fac(2)*, *fac(1)* und zuletzt *fac(0)* auf den Stapel.

Und dann kommt es wieder zum Leeren des Stacks und die Rückgabewerte kommen zum Tragen. Zuerst wird *fac(0)* aufgerufen und vom Stapel genommen. Diese Funktion liefert den Rückgabewert 1, was im Moment uninteressant ist. Aber beim nächsten Aufruf (*fac(1)*) gelangen wir in den *else*-Zweig. Und dort wird der Wert von *n* (1) mit dem Rückgabewert von *fac(0)* (1) multipliziert. Das Ergebnis (1) ist der Rückgabewert von *fac(1)*.

Nun liegt *fac(2)* oben im Stapel. Und dort wird der Wert von *n* (2) mit dem Rückgabewert von *fac(1)* (1) multipliziert. Das Ergebnis (2) ist der Rückgabewert von *fac(2)*.

Jetzt ist *fac(3)* oben im Stapel und hier wird der Wert von *n* (3) mit dem Rückgabewert von *fac(2)* (2) multipliziert. Das Ergebnis (6) ist der Rückgabewert von *fac(3)*.

So geht es weiter. Bei *fac(4)* wird der Wert von *n* (4) mit dem Rückgabewert von *fac(3)* (6) multipliziert. Das Ergebnis (24) ist der Rückgabewert von *fac(4)*.

Der letzte Funktionsaufruf auf dem Stapel nimmt den Wert 5 für *n* und multipliziert ihn mit 24. Und diese Rückgabe von 120 ist letztendlich das, was der Aufrufer von *fac(5)* bekommt und verarbeiten (in unserem Fall ausgeben) kann.

Der Stack ist jetzt auch wieder leer.

9.5 Innere Funktionen – Closures

Python erlaubt auch den Aufbau verschachtelter Funktionen. Dabei wird eine Funktionsdeklaration direkt in eine andere Funktion hinein notiert! In dem Fall gelten einige spezielle Regeln:

- Die innere verschachtelte Funktion kann nur über die äußere aufgerufen werden und ist für direkte Zugriffe unzugänglich!
- Die innere Funktion kann die Variablen oder andere innere Funktionen der äußeren Funktion verwenden.
- Die äußere Funktion hat keinen Zugang zu den lokalen Variablen der inneren Funktion. Bei einem versuchten Zugriff kommt es zu einem *NameError*.

Zur Laufzeit erzeugt der Python-Interpreter beim Aufruf der äußeren Funktion den Code der inneren Funktion. Dabei entsteht ein sogenanntes **Closure** (Einschluss) der inneren Funktion – das ist der Code der Funktion und eine Referenz auf alle Variablen, die von der inneren Funktion benötigt werden. Ein Closure kombiniert den Programmcode mit der lexikalischen Umgebung – das heißt, das Closure „merkt“ sich die Umgebung, in der es erzeugt wurde. Beispiel (*Closure.py*):

```
def aussen():
    i = 42
    def innen():
        text = "Die Antwort ist"
        print(text)
        print(i)
    innen()
aussen()
```

- Zuerst wird eine Funktion *aussen()* definiert, die direkt hinter der Deklaration aufgerufen wird.
- In der Funktion *aussen()* gibt es eine lokale Variable *i*, die mit dem Wert 42 initialisiert wird. Dazu finden Sie die innere Funktion *innen()*.
- Die innere Funktion *innen()* deklariert eine lokale Variable, greift aber auch auf die lokale Variable *i* aus dem Geltungsbereich der äußeren Funktion zu.
- Der Aufruf der inneren Funktion *innen()* erfolgt im Geltungsbereich der äußeren Funktion.

Mit Closures kann man in Python auch rein funktional das Konzept der Datenkapselung unterstützen, das im Rahmen der objektorientierten Programmierung zu den Grundprinzipien zählt.

9.6 Lambda-Ausdrücke und anonyme Funktionen

In vielen Sprachen kennt man sogenannte **anonyme Funktionen**. Das sind Funktionen, bei deren Deklaration man keinen Bezeichner angibt, die aber eine **Funktionsreferenz** auf sich selbst zurückliefern. In Python werden diese über sogenannte **Lambda-Ausdrücke** umgesetzt. Dazu gibt es den *lambda*-Operator, der einer Funktionsdeklaration vorangestellt wird.

Lambda-Funktionen können eine beliebige Anzahl von Parametern haben, führen einen Ausdruck aus und liefern den Wert dieses Ausdrucks als Rückgabewert. Die allgemeine Syntax einer Lambda-Funktion sieht so aus:

Beispiel

Lambda Argumentenliste: Ausdruck ◀

Die Argumentenliste besteht aus einer durch Kommata getrennten Liste von Argumenten, und nach dem Doppelpunkt folgt ein Ausdruck, der diese Argumente benutzt. Schauen wir uns ein einfaches Beispiel der reinen Deklaration einer Lambda-Funktion an (*Lambda.py*).

```
lambda x: x + 21
```

Bei dem obigen Beispiel handelt es sich um eine Funktion mit einem Argument *x*. Der Wert 21 wird dem Wert von *x* aufaddiert. Doch wie können wir die obige Funktion benutzen? Sie hat ja explizit keinen Namen!

9.6.1 Lambda-Funktionen verwenden

Der „Trick“ beruht darauf, dass man entweder den Lambda-Ausdruck direkt in einem anderen Ausdruck (etwa einer Funktion oder dem Iterator für eine sequenzielle Datenstruktur) verwendet oder aber einer Variablen zuweist, um darüber die Funktion zu verwenden. Denn Lambda-Ausdrücke liefern ja eine Funktionsreferenz!

Nutzen wir also nun die Deklaration einer Lambda-Funktion (*Lambda.py*).

```
lbd = lambda x: x + 21
print(lbd(21))
def lambdaAnwenden(x, liste):
    for l in liste:
        print(x(l))

lambdaAnwenden(lbd, [2, 3, 5, 7, 11])
lambdaAnwenden(lambda x: x * 2, [2, 3, 5, 7, 11])
```

Sie sehen, dass der Lambda-Ausdruck einer Variablen *lbd* zugewiesen wird. Damit kann man die Funktion über *lbd()* benutzen, wobei man natürlich die geforderten Parameter anzugeben hat.

Besonders spannend sind Lambda-Ausdrücke, wenn diese als Parameter an andere Funktionen notiert werden, entweder über eine benannte Funktionsreferenz (*lbd*) oder anonym (*lambda x: x * 2*). In der Funktion *lambdaAnwenden()* in dem Beispiel nimmt die Funktion zwei Parameter entgegen. Der erste ist eine Funktionsreferenz und der zweite eine Struktur wie etwa eine Liste (diese wird noch genauer bei sequenziellen Datentypen erläutert). Über diese wird iteriert und für jedes der Elemente der Lambda-Ausdruck angewendet. Das ist eine der häufigsten Anwendungen von Lambda-Ausdrücken.

- ▶ Lambda-Ausdrücke findet man auch oft in der objektorientierten Programmierung, um Methoden anonym zu deklarieren. Damit kann man die Datenkapselung auf einfache Weise umsetzen.
-

9.7 Besondere Situationen bei Funktionen in Python

Der Umgang mit Funktionen ist in Python also – vielleicht mit Ausnahme der rekursiven Aufrufe und der Closures – ziemlich trivial, wie Sie gerade gesehen haben. Aber es gibt ein paar Details, um die man sich Gedanken machen muss.

9.7.1 Lokale Variablen in Funktionen

Es wurde schon beschrieben, dass Parameter in der Funktion natürlich auch in Python lokale Variablen sind. Aber was gilt für Variablen, die in Funktionen neu (und ohne besondere Vorkehrungen) deklariert werden? Sind sie lokal wie etwa in PHP oder global wie zum Beispiel in JavaScript?

Die Antwort: Sie sind in Python **lokal** und stehen damit nicht außerhalb der Funktion zur Verfügung.

Ein Zugriff von außen führt damit zu einem Fehler (*NameError* Abb. 9.3).

```
> print(a) # a ist global nicht vorhanden
NameError: name 'a' is not defined
>>> |
```

Abb. 9.3 Die Variable ist lokal und von außen nicht zugänglich

```
def rechnen():
    a = 3
    print(a * 2)
rechnen()
print(a) # a ist global nicht vorhanden
```

Das Folgende funktioniert hingegen:

```
a = 42
def rechnen():
    a = 3
    print(a * 2)
rechnen()
print(a) # a ist global vorhanden
```

Die Ausgabe ist das:

```
6
42
```

- ▶ Sie sehen am Ergebnis, dass die globale Variable *a* in der Funktion temporär **verdeckt** wird, da sie den gleichen Namen hat. Änderungen an lokalen Variablen mit dem gleichen Namen wie die verdeckten Variablen wirken sich nicht auf globale Variablen aus.

9.7.2 Die Anzahl der Parameter passt nicht

Was passiert nun in Python, wenn die Anzahl der Parameter einer Funktion bei der Deklaration nicht mit der Anzahl übereinstimmt, die beim Aufruf angegeben wird? Schauen wir uns die Details an.

Dabei müssen wir zwei Situationen unterscheiden.

9.7.2.1 Mehr Parameter beim Aufruf

Wenn zu viele Parameter beim Aufruf einer Funktion angegeben werden, kommt es zu einem *TypeError*. Betrachten Sie das Beispiel (Abb. 9.4):

```
rechnen(1)
TypeError: rechnen() takes 0 positional arguments but 1 was given
>>> |
```

Abb. 9.4 Der Parameter beim Aufruf ist zu viel

```
def rechnen():
    print(1)
rechnen() # Ein Parameter wird beim Aufruf angegeben,
          # obwohl kein Parameter erlaubt ist
```

9.7.2.2 Weniger Parameter beim Aufruf – optionale Parameter und variable Anzahl

Auch wenn zu wenige Parameter beim Aufruf einer Funktion angegeben werden, kommt es zu einem *TypeError* – Abb. 9.5.

```
def rechnen(a):
    print(1)
rechnen() # Ein Parameter fehlt beim Aufruf, obwohl ein Parameter nötig ist
```

Aber hier gibt es seitens Python zwei Möglichkeiten, so ein Verhalten zu erlauben:

- Funktionen können einmal **optionale Parameter** haben. Man nennt sie auch Default-Parameter. Dies sind Parameter, die beim Aufruf nicht angegeben werden müssen. In diesem Fall werden dann Vorgabewerte für diese Parameter eingesetzt, die bei der Deklaration durch ein Gleichheitszeichen zugewiesen werden.
- Eine **variable Anzahl von Parametern** lässt sich durch die Markierung eines Parameters mit einem Stern deklarieren. Das muss dann aber zwingend der letzte Parameter einer Funktion sein. Dieser Parameter ist dann als Liste zu verstehen.

Hier ist ein Listing mit beiden Varianten der Parameter (*BesondereParameter.py*):

```
def rechnen(a,b = 8): # optionaler Parameter b
    print(a*b)
def multiplizieren(a, *b): # b ist eine Liste variabler Länge
    erg = a
    for i in b:
        erg *=i
    print(erg)
rechnen(6)
```

```
rechnen() # Ein Parameter fehlt beim Aufruf, obwohl ein Parameter nötig ist
TypeError: rechnen() missing 1 required positional argument: 'a'
>>>
```

Abb. 9.5 Beim Aufruf fehlt ein Parameter

```
rechnen(6, 7)
multiplizieren(6)
multiplizieren(6, 1)
multiplizieren(6, 1, 1)
```

- Beim ersten Aufruf der Funktion *rechnen()* wird für *b* der Wert 8 als Vorgabewert genommen, aber beim zweiten Aufruf der Wert 7.
- Beim ersten Aufruf von *multiplizieren()* ist die Liste mit den variablen Parametern leer, dann enthält sie ein Element und zum Schluss zwei Elemente.

Machen wir uns noch einmal klar, warum bei variablen Parametern diese zwingend am Ende der Parameterliste stehen müssen. Dazu wandeln wir das letzte Beispiel etwas ab, das einen Fehler produzieren wird (Abb. 9.6):

```
def multiplizieren(*b, a): # b ist eine Liste variabler Länge
    erg = a
    for i in b:
        erg *=i
    print(erg)
multiplizieren(6)
multiplizieren(6, 1)
multiplizieren(6, 1, 1)
```

Bei dem Aufruf der Funktion stellt sich die Frage, ob der Wert 6 nun Teil der Liste *b* ist oder der Wert für den Parameter *a*. Denn die Liste *b* ist ja optional und kann leer sein. Es gibt zum Aufrufzeitpunkt nirgendwo eine Möglichkeit, zu erkennen, was genau gemeint ist. Deshalb können optionale Parameter nur als letzter Eintrag in einer Parameterliste auftauchen.

9.7.2.3 Deklaration nach dem Funktionsaufruf

Der nachfolgende Code deklariert die Variable hinter dem Aufruf der Funktion, und das führt wie gesagt zu einer *NameError*.

So weit ist die Sache klar, wenn man die sequenzielle Abarbeitung des Quellcodes im Hinterkopf behält. Leider gibt es ein tückisches Verhalten in Python. Der folgende Code funktioniert auch nicht (*UnboundLocalError* – Abb. 9.7):

Abb. 9.6 Nach der Liste folgt ein weiterer Parameter

```
print(a)
NameError: name 'a' is not defined
>>>
```

```

        multiplizieren(6)
TypeError: multiplizieren() missing 1 required keyword-only argument: 'a'
>>> |

```

Abb. 9.7 Ein UnboundLocalError

```

a = 42
def ausgabe():
    print(a)
    a = 3
    print(a)
ausgabe()
print(a)

```

In diesem Fall wird die Anweisung `print(a)` in der Funktion von Python so betrachtet, dass man eine lokale Variable `a` verwenden will, und die wird erst in der folgenden Zeile deklariert (gebunden – deshalb **UnboundLocalError**). Das Tückische ist, dass erst die **nachfolgende** Zeile diese Interpretation durch den Python-Interpreter bewirkt. Das Verhalten ist umso tückischer, denn Folgendes funktioniert wieder (*GlobaleVariablenKeinUnboundLocalError.py*):

```

a = 42
def ausgabe():
    a = 3
    print(a)
    print(a)
ausgabe()
print(a)

```

In dem Fall wird aber die globale Variable `a` durch die Zuweisung in der Funktion nicht verändert, denn `a` ist lokal. Die Ausgabe ist:

```

3
3
42

```

9.7.3 Unerreichbarer Code

Es ist leider in manchen Sprachen durch unglückliche Codierung des Quelltexts möglich, Codezeilen zu schreiben, die nie erreicht werden können. Dies nennt man **unerreichbaren Code** („unreachable code“). Solcher Code ist zwar eigentlich nicht schädlich

in dem Sinne, dass er etwas Falsches tut, aber wenn man sich darauf verlässt, dass bestimmte Codezeilen ausgeführt werden, und sie werden einfach nicht erreicht, kann der Schaden mindestens genauso groß sein. So etwas würde erstellt, wenn Sie nach einer `return`-Anweisung in einer Funktion weitere Anweisungen notieren. Mächtige Sprachen wie Java verhindern durch ihr Sicherheitskonzept die Erstellung von unerreichbarem Code oder warnen zumindest, wenn man diesen Fehler macht (etwa C#). Wie verhält sich Python?

Leider ist es möglich, dass man in Python unerreichbaren Code erstellt. Das liegt nicht zuletzt daran, dass der Interpreter gar nicht erkennen kann, dass nach einer Sprungweisung noch weiterer Code steht – er wird ja nicht erreicht. Beispiel (*Unreachable-Code.py*):

```
def ausgabe():
    return 1
    print("Wird nie erreicht")
print(ausgabe())
```



Sequenzielle Datenstrukturen – Mehrere Informationen gemeinsam verwalten

10

10.1 Was behandeln wir in diesem Kapitel?

Ein Highlight von Python ist dessen Umgang mit verschiedenen Arten von Datenstrukturen, in denen mehrere Informationen gemeinsam bereitgestellt werden. Für diese gibt es sowohl diverse Arten der Verwaltung als auch eine ganze Reihe an nützlichen und bequemen Techniken für den Umgang damit. Diesem Thema widmen wir uns jetzt.

10.2 Was sind sequenzielle Datenstrukturen?

Allgemein bestehen in Python¹ sogenannte sequenzielle Datentypen aus einer Aneinanderreihung von einzelnen Werten. Damit kann man Variablen anlegen, die mehrere Werte zur Verfügung stellen, oder auch direkt mit diesen komplexen Datenstrukturen arbeiten. In diesem Kapitel sollen zwei sequenzielle Strukturen genauer vorgestellt werden, die wir entweder schon kennen oder die unter dem Begriff so in Python nicht verwendet werden.

¹ Wie auch in den meisten anderen Programmiersprachen.

10.2.1 Zeichenketten als sequenzielle Ansammlung von Zeichenliteralen

Den vermutlich gebräuchlichsten sequenziellen Datentyp haben Sie bereits kennen gelernt, denn wir haben damit schon mehr oder weniger ab dem ersten Python-Beispiel gearbeitet. Das war die **Zeichenkette**. Denn es handelt sich dabei im Hintergrund um eine Sequenz von einzelnen Zeichenliteralen.

10.2.2 Arrays

Nun kennt man eigentlich aus allen anderen Programmiersprachen sogenannte **Arrays**. Ein Array ist allgemein erst einmal nur eine Sammlung von Werten, die alle über

- einen gemeinsamen Bezeichner und
- einen Index

angesprochen werden können. Arrays sind immer dann von Nutzen, wenn eine Reihe von gleichartigen oder zusammengehörenden Informationen gespeichert werden soll. Wenn Sie für jede einzelne Information eine eigene Variable definieren, müssen Sie viele sinnvolle Namen vergeben und haben viel Schreibarbeit. Außerdem steigt die Fehlerwahrscheinlichkeit. Ein Name und ein Index sind effektiver. Der Hauptvorteil ist aber, dass der Zugriff auf die einzelnen Einträge im Array über den Index erfolgen kann. Das kann man in Programmkontrollflussanweisungen und sonstigen automatisierten Vorgängen nutzen.

Unglücklicherweise wird der Begriff „Array“ in Python jedoch etwas anders als üblich verwendet. Es gibt zwar ein paar Funktionen, die in dieser Sprache mit dem Begriff assoziiert sind, aber das ist eher ein Randthema. Und in Python gibt es etwa **Byte-Arrays**, was die Arbeit mit Binäroperatoren berührt. Diese Form eines sequenziellen Datentyps ist für die Speicherung oder für die Übertragung von Binärdaten besonders gut geeignet.

Beispiel:

```
zahlen = bytes([0x13, 0x00, 0x00, 0x00, 0x08, 0x00])
print(zahlen)
```

Allerdings ist die Bedeutung von Byte-Arrays in Python eher gering, und wir werden das Thema hier nicht vertiefen.² Überhaupt werden wir den Begriff „Array“ verlassen. Denn statt Arrays im Sinn anderer Programmiersprachen benennt man in Python diese

²Beim Lesen und Schreiben werden wir allerdings im Zusammenhang mit binären Daten darauf zurückkommen.

indizierten Datenstrukturen mit **Tupeln** und **Listen**. Aber auch **Dictionaries** decken in Python einen Teil dessen ab, was man üblicherweise unter Arrays versteht.

10.3 Tupel

Tupel sind in Python wie Strings eine Aneinanderreihung von beliebigen Werten. Tupel entstehen in Python ganz einfach, wenn man in **runden Klammern** Daten komma-separiert hintereinander schreibt.

Beispiel:

```
primzahlen = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31)
```

Beachten Sie, dass Tupel auf mehrere Zeilen verteilt werden können und dabei Leerzeichen und vor allen Dingen auch die Einrückung keine Rolle spielen, da die Elemente eines Tupels keine neuen Python-Anweisungen darstellen. Dies wäre also eine korrekte Schreibweise, wenngleich aus meiner Sicht nicht gut lesbar und nicht zu empfehlen:

```
primzahlen = (2,  
              3,  
              5, 7, 11, 13, 17,  
              19, 23, 29, 31)
```

Es können in Python auch beliebige Datentypen in einem Tupel auftauchen.

Beispiel:

```
adresse = ("Monthly", "Python", "Milchstrasse", 42, 12345, "Universum")
```

10.3.1 Verschachtelte Tupel

Da in Python eben beliebige Datentypen in einem Tupel auftauchen können, können Tupel in Python auch **verschachtelt** werden. Dazu wird einfach als Wert eines Tupels ein anderes Tupel notiert.

Beispiel:

```
verschachteltestupel = ((1,2), (3,4))
```

Diese und die vorherige Tatsache gestatten es, dass man auch so etwas machen kann (*Tupell.py*):

```
primzahlen = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31)  
print(primzahlen)
```

```

adresse = ("Monthly", "Python", "Milchstrasse", 42, 12345, "Universum")
print(adresse)
verschachteltestupel = ((1,2), (3,4))
print(verschachteltestupel)
tupel1 = (primzahlen, adresse)
print(tupel1)
tupel2 = ((2,3,5,7), (11,13,17))
print(tupel2)

```

Schauen wir uns den Quelltext schrittweise an:

- Erst wird am Anfang ein einfaches Tupel mit Zahlen angelegt und ausgegeben.
- Dann folgt ein Tupel mit verschiedenen Datentypen, dessen Inhalt ebenso ausgegeben wird.
- Das dritte Tupel ist verschachtelt, und an der Ausgabe sehen Sie, dass die print-Funktion einfach sequenziell das erste innere und dann das zweite innere Tupel durchläuft.
- Das vierte Tupel nimmt zwei bereits angelegte Tupel und macht daraus ein verschachteltes Tupel (also ein Tupel mit Tupel als Inhalt).
- Ebenso ist das fünfte Tupel ein verschachteltes Tupel, wobei aber die inneren Tupel wieder direkt angelegt werden. Die Besonderheit ist hier, dass die inneren Tupel eine unterschiedliche Anzahl an Elementen enthalten.

Das ist die Ausgabe (Abb. 10.1):

10.3.2 Tupel und der Membership-Operator

Nutzen wir nun einmal den **Membership-Operator** *in*, um den Inhalt eines Tupels zu untersuchen. Dabei können wir gleich auch noch die *if*-Anweisung und die *while*-Schleife in einer praktischen Anwendung sehen.

```

(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31)
('Monthly', 'Python', 'Milchstrasse', 42, 12345, 'Universum')
((1, 2), (3, 4))
((2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31), ('Monthly', 'Python', 'Milchstrasse', 42, 12345, 'Universum'))
((2, 3, 5, 7), (11, 13, 17))
>>>

```

Abb. 10.1 Tupel in verschiedenen Varianten

Die streng vereinfachte Situation soll wie folgt aussehen (*PruefeGeheimzahl.py*):

- Ein Anwender möchte am Geldautomaten mit einer EC- oder Kreditkarte Geld abheben und muss dazu seine Geheimzahl eingeben. Die Karte soll bereits in den Automaten eingeführt und ein Betrag X für die Auszahlung ausgewählt sein.
- Bei dem Beispiel ist ein Tupel mit gültigen Geheimzahlen vorhanden. Die eingegebene Geheimzahl muss aus diesem Bereich stammen.
- Ist die eingegebene Geheimzahl in dem Tupel vorhanden, dann bekommt der Anwender Geld ausgezahlt (Abb. 10.4). Andernfalls kann der Anwender die Geheimzahl erneut eingeben (Abb. 10.2, 10.3 und 10.4).
- Maximal drei Versuche hat der Anwender, um die richtige Geheimzahl einzugeben (Abb. 10.3).
- Der Anwender kann nach einer fehlerhaften Eingabe der Geheimzahl den Vorgang abbrechen (Abb. 10.2), zumindest so lange, wie die Anzahl der Versuche nicht zu groß wird. Nach dem dritten Fehlversuch ist Schluss.

Abb. 10.2 Der Vorgang wurde vom Anwender abgebrochen

```
Geben Sie Ihre Geheimzahl ein:  
12345  
Die Geheimzahl war leider nicht korrekt.  
  
Abbruch (A) ?  
N  
Geben Sie Ihre Geheimzahl ein:  
12345  
Die Geheimzahl war leider nicht korrekt.  
  
Abbruch (A) ?  
A  
>>>
```

Abb. 10.3 Zu viele Versuche

```
Geben Sie Ihre Geheimzahl ein:  
123456  
Die Geheimzahl war leider nicht korrekt.  
  
Abbruch (A) ?  
W  
Geben Sie Ihre Geheimzahl ein:  
231123  
Die Geheimzahl war leider nicht korrekt.  
  
Abbruch (A) ?  
W  
Geben Sie Ihre Geheimzahl ein:  
232323  
Die Geheimzahl war leider nicht korrekt.  
  
Sie haben zu viele Versuche benötigt.  
Die Karte wird eingezogen.  
>>> |
```

```

Geben Sie Ihre Geheimzahl ein:
56578
Die Geheimzahl war leider nicht korrekt.

Abbruch (A)?
N
Geben Sie Ihre Geheimzahl ein:
1234
Die Geheimzahl ist korrekt. Geld auszahlen
>>> |

```

Abb. 10.4 Erfolgreiche Überprüfung

So könnten die Anforderungen umgesetzt werden:

```

gz = (1234, 5678, 87654, 12333, 4711)
anzahlVersuche = 0
while anzahlVersuche < 3:
    e = int(input("Geben Sie Ihre Geheimzahl ein:\n"))
    if e in gz:
        print("Die Geheimzahl ist korrekt. Geld auszahlen")
        break
    print("Die Geheimzahl war leider nicht korrekt.\n")
    anzahlVersuche += 1
    if anzahlVersuche < 3:
        w = input("Abbruch (A) ?\n")
        if w == "A": break
if anzahlVersuche >= 3:
    print("Sie haben zu viele Versuche benötigt.\nDie Karte wird eingezogen.")

```

Sie sehen in dem Listing den Membership-Operator, mit dem überprüft wird, ob die Eingabe mit den Elementen in dem Tupel für die Geheimzahlen übereinstimmt. In dem Beispiel wird die Anzahl der Eingabeversuche mitgezählt. Der Wert der Variable *anzahlVersuche* wird zur Steuerung des Programmflusses verwendet. Damit wird sowohl die Schleife gesteuert als auch die Entscheidung getroffen, ob die Anzahl der Versuche so hoch ist, dass die Karte eingezogen wird.

Und da ein eventueller Abbruch des Vorgangs nur dann erfolgen kann, wenn der Wert der Variable noch kleiner als 3 ist, kann dieses Einziehen der Karte auch nicht erfolgen, wenn der Anwender den Vorgang rechtzeitig abbricht.

- Beachten Sie, dass in dem Beispiel die Situation wie gesagt streng vereinfacht ist.
Es werden vor allen Dingen keine Fehleingaben (keine Zahlen etc.) abgefangen.

10.3.3 Einzelne Einträge in Tupel ansprechen

Nun stellt sich die Frage, wie man einzelne Elemente in einem Tupel ansprechen kann. Es geht über den Index. Dieser muss in **eckigen Klammern** angegeben werden.

Beispiel:

```
primzahlen[2]
```

- ▶ Tupel sind **nullindiziert**. Der erste Wert in einem Tupel hat also den Index 0. Das geht mit so gut wie allen modernen Sprachen einher, die Datenstrukturen immer nullindizieren.

Die Anzahl der Elemente in einem Tupel als auch jeder anderen abzählbaren sequenziellen Struktur erhalten Sie mit der Build-in Function *len()* (Abschn. 10.3.4).

Doch wie spricht man bei verschachtelten Tupeln die einzelnen enthaltenen Werte an?

Die Indizes müssen in aufeinanderfolgenden eckigen Klammern notiert werden, ganz so wie es in den meisten anderen Programmiersprachen bei Arrays üblich ist.

Beispiel:

```
tupel2[1][1]
```

Schauen wir uns das Verfahren in einem vollständigen Beispiel an (*Tupel2.py*).

```
primzahlen = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31)
print(primzahlen[0])
adresse = ("Monthly", "Python", "Milchstrasse", 42, 12345, "Universum")
print(adresse[1])
verschachteltestupel = ((1,2), (3,4))
print(verschachteltestupel[1])
print(verschachteltestupel[1][1])
tupel1 = (primzahlen, adresse)
print(tupel1[0])
print(tupel1[1][1])
tupel2 = ((2,3,5,7), (11,13,17))
print(tupel2[1][2])
```

Wir verwenden in dem Beispiel die gleichen Tupel wie im Listing *Tupel1.py*. Nur greifen wir jetzt gezielt auf die einzelnen Elemente zu. Verifizieren Sie die Schritte mit der Ausgabe des Beispiels (Abb. 10.5).

```

2
Python
(3, 4)
4
(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31)
Python
17
>>> |

```

Abb. 10.5 Zugriff auf einzelne Elemente des Tupels

- Der erste Zugriff mit `print(primzahlen[0])` sollte klar sein. Da wir eine nullindizierte Sequenz haben, ist das das erste Element des Tupels, und das ist der Wert 2.
 - Der zweite Zugriff mit `print(adresse[1])` greift auf den zweiten Wert des Tupels `adresse` zu, und da steht der String "Python".
 - Der dritte Zugriff über `print(verschachteltestupel[1])` greift auf das zweite Element des Tupels `verschachteltestupel` zu. Das ist aber selbst wieder ein Tupel, deshalb sehen Sie in der Ausgabe (3, 4) – also beide Elemente des „inneren“ Tupels.
 - Mit `print(verschachteltestupel[1][1])` greifen wir jedoch dann gezielt auf das zweite Element im „inneren“ Tupel zu, das ist der Wert 4.
 - Bei dem Zugriff Nummer 5 greifen wir mit `print(tupel1[0])` auf das erste Element des Tupels `tupel1` zu, das ist das innere Tupel `primzahlen`. Deshalb wird der Inhalt des gesamten Tupels `primzahlen` ausgegeben.
 - Mit `print(tupel1[1][1])` nehmen wir das zweite Element von dem inneren Tupel `adresse`, das ist wieder der String "Python".
 - Die letzte Zugriffsvariante `print(tupel2[1][2])` nimmt von dem Tupel `tupel2` das zweite Element (selbst wieder ein Tupel) und dessen drittes Element. Das ist der Wert 17.
- Wenn Sie beim Zugriff auf ein Element einer sequenziellen Datenstruktur einen Index verwenden, den es nicht gibt, erhalten Sie einen *IndexError* (Abb. 10.6 und Abb. 10.7).
- Negative Indizes sind möglich! Dann zählt der Index von hinten (Abschn. 10.3.3.1).

```

        print(person[7])
IndexError: string index out of range
>>>

```

Abb. 10.6 Über das Ende des Strings hinausgegriffen – *IndexError: string index out of range*

```
print(primzahlen[14])
IndexError: tuple index out of range
...
```

Abb. 10.7 Das Tupel-Element ist nicht vorhanden – IndexError: tuple index out of range

10.3.3.1 Angabe eines Indexbereichs (Range)

Wenn Sie bei dem Zugriff auf Elemente eines Tupels beim Index einen Bereich angeben, können Sie damit gezielt Elemente aus dem sequenziellen Datentypen extrahieren. Sie geben dazu in den eckigen Klammern den Anfangs- und den Ende-Index an und trennen diese Indizes mit einem Doppelpunkt.

- ▶ Das Verfahren funktioniert auch bei allen abzählbaren sequenziellen Datentypen – nicht nur Tupel.

Bei Indexbereichen gibt es nun in Python das etwas gewöhnungsbedürftige Verhalten, dass ein zu großer Endeindex bei einem Range **keinen** *IndexError* auslöst, wie es bei einem Zugriff auf einen einzelnen fehlerhaften Index der Fall ist und man eigentlich erwarten sollte.³ Stattdessen wird einfach der Bereich bis zum letzten Element verwendet.

Das nachfolgende Beispiel verwendet Indexbereiche bei einem String und einem Tupel und zeigt die verschiedenen Verhaltensweisen (*Tupel3.py*).

```
person = "Brain"
print(person[1:4])
print(person[1:len(person)])
print(person[1:14])
print(person[-4:-1])
primzahlen = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31)
print(primzahlen[1:4])
print(primzahlen[5:len(primzahlen)])
print(primzahlen[-5:-1])
```

Verifizieren Sie die Schritte mit der Ausgabe des Beispiels (Abb. 10.8).

- Mit *print(person[1:4])* wird der Teil des Strings "Brain" ausgegeben, der mit dem zweiten Zeichen beginnt und dem vierten Zeichen endet. Das ist "rai".
- Bei *print(person[1:len(person)])* nutzen wir die Funktion *len()*. Diese liefert für den String "Brain" den Wert 5, deshalb ist die Ausgabe "rain".

³Bei den meisten anderen Programmiersprachen würde so etwas passieren.

Abb. 10.8 Indexbereiche

```

rai
rain
rain
rai
(3, 5, 7)
(13, 17, 19, 23, 29, 31)
(17, 19, 23, 29)
>>>

```

- Die Länge des Strings ist wie gesagt 5 und der Endeindex der Bereichsangabe bei `print(person[1:14])` deutlich größer. Dennoch kommt es nicht zu einem Fehler, sondern die Ausgabe "rain" zeigt, dass einfach am Ende der Sequenz abgeschnitten wird.
- Mit `print(person[-4:-1])` nutzen wir negative Indizes sowohl für den Beginn als auch den Endeindex. Der Anfangsindex wird vom Wert 5 (der Länge des Strings) abgezogen und $5 - 4$ ergibt 1. Das ist also das Zeichen, das ist "r". Der Index für das letzte Zeichen ergibt sich aus $4 - 1$, und das ist das Zeichen "i". Also ist die Ausgabe "rai".
- Die nachfolgenden Ausgaben zeigen die gleichen Schritte für ein Tupel.

10.3.4 Die Anzahl der Elemente in einem Tupel bestimmen

Wenn Sie die Anzahl der Elemente in einem Tupel bestimmen wollen, steht Ihnen die Funktion `len()` zur Verfügung. Sie brauchen ihr als Parameter bloß das Tupel zu übergeben, dessen Anzahl von Elementen gezählt werden soll. Der Rückgabewert ist die Anzahl der Elemente im Tupel. Die Funktion kann man auch für Listen oder Dictionaries einsetzen.

10.4 Dynamische Listen

Nun gibt es in Python noch **Listen** als sequenziellen Datentypen. Auf den ersten Blick sind Listen und Tupels fast identisch, nur werden Listen in **eckige Klammern** gesetzt. Im einfachsten Fall erzeugt man eine leere Liste durch eckige Klammern ohne Inhalt. Beispiel:

```
primzahlen = []
```

Die Liste müsste man nun nach und nach dynamisch füllen. Aber man kann auch Vorgabewerte direkt bei der Deklaration angeben. So würde das leicht modifizierte Beispiel von oben mit Listen aussehen (*Liste1.py*):

```
primzahlen = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
print(primzahlen)
adresse = ["Monthly", "Python", "Milchstrasse", 42, 12345, "Universum"]
print(adresse)
verschachtelteliste = [[1,2], [3,4]]
print(verschachtelteliste)
listelupel = [(1,2), (3,4)]
print(listelupel)
liste1 = [primzahlen, adresse]
print(liste1)
liste2 = [[2,3,5,7], (11,13,17)]
print(liste2)
```

Das wäre die Ausgabe:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
['Monthly', 'Python', 'Milchstrasse', 42, 12345, 'Universum']
[[1, 2], [3, 4]]
[(1, 2), (3, 4)]
[[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31], ['Monthly', 'Python', 'Milchstrasse', 42, 12345,
'Universum']]
[[2, 3, 5, 7], (11, 13, 17)]
```

Beachten Sie, dass in Listen auch Tupel notiert werden können und umgekehrt. Es kann jeder Datentyp als Inhalt auftauchen. Das wird in dem Beispiel auch gemacht.

10.4.1 Warum Listen und Tupel?

Doch wozu gibt es noch Listen, wenn es schon Tupel gibt? Das wurde schon angedeutet – Tupel können nach der Erzeugung nicht mehr geändert werden („**immutable**“), während Listen nachträglich veränderbar („**mutable**“) sind.

- Tupel sind performanter, Listen dynamisch.

Machen wir uns den Unterschied praktisch deutlich (*Liste2.py*).

```
primzahlen = [2, 3, 5, 7, 11]
print(primzahlen)
primzahlen[0] = 6
print(primzahlen)
```

Sie werden folgende Ausgabe bekommen:

```
[2, 3, 5, 7, 11]
[6, 3, 5, 7, 11]
```

Versuchen wir, ein Tupel nachträglich zu ändern (*TupelDynFehler.py*).

```
# Fehlerhafter Versuch zur Änderung eines Tupels
primzahlen = (2, 3, 5, 7, 11)
primzahlen[0] = 6
print(primzahlen)
```

Sie werden die Ausgabe mit einem **Traceback** (einer Fehlermeldung) bekommen:

primzahlen[0] = 6
TypeError: 'tuple' object does not support item assignment

10.5 Methoden für Listen

Wir werden nun einen kleinen Vorgriff vornehmen müssen, aber für den Einsatz von Listen wie auch Tupeln und andere sequenzielle Strukturen sind die **Listenmethoden**⁴ elementar. Der Vorgriff ist insofern notwendig, weil wir mit „Methoden“ die bisher noch nicht explizit behandelte OOP (objektorientierte Programmierung) berühren und Python erlaubt ja explizit Programmierung nach dem OO-Paradigma, auch gemischt mit anderen Programmierstilen. Im Detail wird das noch vertieft, aber hier folgt zunächst ein kurzer Exkurs zu den elementaren Grundlagen.

► Definition

- **Objekte** dienen in der OOP als Abstraktion eines realen Gegenstands oder Konzepts und verfügen über einen spezifischen Zustand und ein spezifisches Verhalten. Man bezeichnet sie auch als **Instanzen** von Klassen.
- **Klassen** dienen als Baupläne für Objekte. Sie repräsentieren die Objekttypen. Auf der anderen Seite fassen sie alle relevanten Eigenschaften und Verhaltensweisen der repräsentierten Objekttypen zusammen (sie klassifizieren diese).
- **Attribute** versteht man als Beschreibung objekt- und klassenbezogener Daten. Attribute sind die einzelnen Dinge, durch die sich ein Objekt von einem anderen unterscheidet. Man nennt Attribute meist auch die **Eigenschaften** eines Objekts. Man ruft Eigenschaften in den meisten OO-Sprachen über das Voranstellen des Objekts oder der Klasse auf. Das vorangestellte Objekt oder die Klasse werden in vielen⁵ Sprachen mit einem Punkt von dem Attribut abgetrennt (Punkt- oder Dot-Notation).

⁴Oder vielleicht besser „Methoden für sequenzielle Strukturen“. Aber nicht alle sequenzielle Datenstrukturen haben die gleichen Methoden, und hier stehen erst einmal die Methoden für Listen im Fokus.

⁵Nicht allen – etwa in PHP wird keine Punktnotation verwendet.

- **Methoden** versteht man als Beschreibung der objekt- und klassenbezogenen Funktionalität. Sie repräsentieren das, was Objekte tun. Sie sind das OO-Analogon zu Funktionen und Prozeduren in der funktionalen Programmierung. Wie bei Attributen ruft man in der Regel Methoden über das Voranstellen des Objekts oder der Klasse auf, wobei in vielen Sprachen die **Punkt-** oder **Dot-Notation** zum Einsatz kommt.

Für Listen stehen in Python einige spannende Methoden zur Verfügung.

Erstellen wir zum Test der Methoden einige Beispiele.

10.5.1 Verschiedene Listenmethoden in einem Beispiel

Zuerst sehen wir in der Datei unter dem Namen *Liste3.py* ein paar der Methoden nacheinander im Einsatz.

```
liste = [66.25, 333, 333, 1, 1234.5]
print(liste.count(333), liste.count(66.25), liste.count('x'))
liste.insert(2, -1)
liste.append(333)
print(liste)
print(liste.index(333))
print(liste.remove(333))
print(liste)
liste.reverse()
print(liste)
liste.sort()
print(liste)
print(liste.pop())
print(liste)
```

Das ist die Ausgabe:

```
2 1 0
[66.25, 333, -1, 333, 1, 1234.5, 333]
1
None
[66.25, -1, 333, 1, 1234.5, 333]
[333, 1234.5, 1, 333, -1, 66.25]
[-1, 1, 66.25, 333, 333, 1234.5]
1234.5
[-1, 1, 66.25, 333, 333]
```

Die Ausgaben werden durch Beschreibungen in Tab. 10.1 verdeutlicht. Versuchen Sie sie nachzuvollziehen.

Tab. 10.1 Listenmethoden

Methode	Beschreibung
<i>append(x)</i>	Fügt am Ende der Liste ein Element hinzu.
<i>extend(L)</i>	Erweitert die Liste, indem alle Elemente in der angegebenen Liste angehängt werden.
<i>insert(i, x)</i>	Fügt einen Wert an einer bestimmten Stelle ein. Das erste Argument ist der Index des Elements, vor dem man einfügen soll.
<i>remove(x)</i>	Entfernt das erste Element aus der Liste, dessen Wert <i>x</i> ist.
<i>pop([i])</i>	Entfernt das Element an der angegebenen Position in der Liste. Es wird zurückgegeben. Wenn kein Index angegeben ist, entfernt <i>pop()</i> das letzte Element in der Liste und gibt es zurück.
<i>index(x)</i>	Rückgabe des Index vom ersten Element, dessen Wert <i>x</i> ist.
<i>count(x)</i>	Anzahl des Vorkommens von <i>x</i> in der Liste.
<i>sort(cmp=None, key=None, reverse=False)</i>	Sortiert die Elemente der Liste (die Argumente können für Sortierung verwendet werden, müssen aber nicht).
<i>reverse()</i>	Dreht die Elemente der Liste um.
Listenmethoden	

10.5.2 Einen Stack erzeugen

Ein **Stack** ist ein Stapel mit Werten. Was zuerst auf den Stapel gelegt wurde, wird zuletzt wieder vom Stapel genommen. Das nennt man das **FILO**-Prinzip („first in, last out“). Die Listenmethoden machen es in Python sehr einfach, eine Liste als Stapel zu verwenden, wobei das letzte hinzugefügte Element wieder als erstes Element abgerufen wird. Das macht man so lange weiter, bis der Stapel leer ist.

Um ein Element zum Anfang des Stapels hinzuzufügen, verwenden Sie die Methode *append()*. Um ein Element von der Oberseite des Stapels abzurufen, verwenden Sie *pop()* ohne einen expliziten Index.

In dem folgenden Listing *Stack.py* erzeugen wir einen Stapel mit fünf Elementen und bauen ihn Element für Element von oben wieder ab.

```
stack = []
stack.append(2)
stack.append(3)
stack.append(5)
stack.append(7)
stack.append(11)
print("Aktueller Stack nach dem Hinzufügen von 5 Elementen: ", stack)
```

```
print("Oberstes Element vom Stack geholt: ", stack.pop())
print("Aktueller Stack: ", stack)
print("Oberstes Element vom Stack geholt: ", stack.pop())
print("Aktueller Stack: ", stack)
print("Oberstes Element vom Stack geholt: ", stack.pop())
print("Aktueller Stack: ", stack)
print("Oberstes Element vom Stack geholt: ", stack.pop())
print("Aktueller Stack: ", stack)
print("Oberstes Element vom Stack geholt: ", stack.pop())
print("Aktueller Stack: ", stack)
```

Das ist die Ausgabe des Programms:

```
Aktueller Stack nach dem Hinzufügen von 5 Elementen: [2, 3, 5, 7, 11]
Oberstes Element vom Stack geholt: 11
Aktueller Stack: [2, 3, 5, 7]
Oberstes Element vom Stack geholt: 7
Aktueller Stack: [2, 3, 5]
Oberstes Element vom Stack geholt: 5
Aktueller Stack: [2, 3]
Oberstes Element vom Stack geholt: 3
Aktueller Stack: [2]
Oberstes Element vom Stack geholt: 2
Aktueller Stack: []
```

10.5.3 Eine Queue mit einer Liste erzeugen

Zwar sind Listen für diesen Zweck nicht effizient, aber es ist auch möglich, mit Listenmethoden eine Warteschlange (**Queue**) zu erzeugen. Dabei wird das erste Element hinzugefügt und auch wieder als erstes Element abgerufen. Neue Elemente werden hinten an die Schlange eingesortiert (**FIFO-Prinzip** – „first in, first out“ [englisch für: der Reihe nach]).

Wir erzeugen in dem Beispiel eine Liste mit fünf Elementen und bauen diese Element für Element von unten/vorne wieder ab. Sie können die Funktion *len()* nutzen, um die Anzahl der Elemente in der Schlange zu überwachen und mit *pop(0)* ein Element von unten zu entfernen, solange es Elemente in der Liste gibt. Achten Sie auf die Nullindizierung.

```
queue = []
queue.append(2)
queue.append(3)
queue.append(5)
queue.append(7)
queue.append(11)
print("Queue nach Fertigstellung: ", queue)
```

```
while len(queue) > 0:
    print(queue.pop(0))
    print("Queue: ", queue)
```

Das ist die Ausgabe:

```
Queue nach Fertigstellung: [2, 3, 5, 7, 11]
2
Queue: [3, 5, 7, 11]
3
Queue: [5, 7, 11]
5
Queue: [7, 11]
7
Queue: [11]
11
Queue: []
```

- ▶ Wie erwähnt, sind Listen für Warteschlangen (Queue) ineffizient, weil nach dem Entfernen eines Elements von unten alle anderen Elemente um eins verschoben werden müssen. Um eine Warteschlange in der Praxis zu implementieren, stellt Python über das *collections*-Modul *collections.deque* bereit. Allerdings wollen wir in dieser Phase des Buchs die Verwendung solcher Zusatzbibliotheken noch nicht nutzen.

10.6 Dictionaries

Neben den sequenziellen Datentypen Listen, Strings und Tupel gibt es in Python eine weitere Kategorie von Datentypen, die allgemein „Mapping“ genannt wird und derzeit nur einen implementierten Typ besitzt – das **Dictionary**. Das ist ein assoziatives Feld, also so etwas wie ein Hash, Map oder assoziatives Array, wie es in anderen Sprachen genannt wird. Ein Dictionary besteht aus Schlüssel-Objekt-Paaren (Key-Value-Paaren). Zu einem bestimmten Schlüssel gehört immer ein Objekt beziehungsweise Wert. Es handelt sich um eine in Python sehr wichtige Struktur, die aber ziemlich einfach ist.

Wie Listen können Dictionaries dynamisch zur Laufzeit verändert werden. Sie haben aber ein paar interessante zusätzliche Methoden zur Verfügung, die sich aus dem assoziierten Schlüssel ergeben.

Anlegen kann man im einfachsten Fall ein leeres Dictionary durch **geschweifte Klammern** ohne Inhalt. Beispiel:

```
map = {}
```

Wenn bereits Inhalt vorhanden sein soll, gibt man kommatasepariert die gewünschten Vorgabewerte an. Die Inhalte können wie bei Listen von verschiedenen Werten sein. Bei den Schlüsseln gilt jedoch die Einschränkung, dass nur Instanzen unveränderlicher („immutable“) Datentypen verwendet werden können. Damit fallen Listen und Dictionaries als Schlüssel heraus. Beachten Sie nur, dass Sie immer ein Key-Value-System angeben, das durch den Doppelpunkt getrennt wird. Das kennt man so ja von diversen ähnlichen Konzepten wie etwa JSON (JavaScript Object Notation).

Beispiel:

```
person = {"vorname" : "Otto", "nachname" : "Meier", "alter" : 61, 1 : True }
```

Der Zugriff auf einzelne Werte erfolgt wie bei anderen sequenziellen Datentypen in Python über den Index in eckigen Klammern.

10.6.1 Spezielle Methoden für Dictionaries

Dictionaries sind deshalb interessant, weil neben den üblichen Operationen, die Sie von Listen kennen, noch ein paar zusätzliche Dinge bereitstehen.

- Mit *del d[k]* kann man gezielt einen Schlüssel *k* zusammen mit seinem Wert löschen.
- Mit *pop()* kann man auch auf Dictionaries arbeiten, aber ein wenig anders als bei Listen. Man muss den assoziierten Index angeben. Falls *D* ein Dictionary bezeichnet, dann entfernt *D.pop(k)* den Index *k* zusammen mit seinem Wert aus dem Dictionary. Außerdem bekommen Sie den Wert von *D[k]* als Rückgabewert. Falls bei *pop()* der Schlüssel nicht gefunden wird, wird ein *KeyError* generiert. Um solche Fehler zu vermeiden, gibt es einen eleganten Weg in Python. Die Methode *pop()* hat dazu einen zweiten optionalen Parameter, mit dem man einen Default-Wert für diesen Fall mitgeben kann.
- Die Methode *popitem()* benötigt keine Parameter und liefert ein beliebiges Schlüssel-Wert-Paar als Tupel zurück. Wendet man *popitem()* auf ein leeres Dictionary an, wird auch hier der Ausnahmefehler *KeyError* generiert.
- Über die Methode *get()* bekommt man mit dem Schlüssel als Parameter den zugehörigen Wert. Auch ihr kann als zweiter Parameter ein Default-Wert mitgegeben werden.
- Mit der Methode *copy()* kann man ein Dictionary kopieren, wobei man hierbei in Python von einer „flachen (shallow) Kopie“ spricht. Üblicher in der IT ist die Bezeichnung „Referenz“, die damit erzeugt wird. Wenn sich ein Wert im Original ändert, wirkt sich das auch auf die Kopie aus und umgekehrt.
- Mit der Methode *values()* bekommt man alle Werte, die in dem Dictionary gespeichert sind.

- Mit der Methode `keys()` erhalten Sie ein Element vom Typ `dict_keys`. Das sind alle Schlüssel, die in dem Dictionary vorhanden sind.
- Der Inhalt eines Dictionary kann mittels der Methode `clear()` geleert werden. Das Dictionary wird dabei nicht gelöscht, sondern wirklich nur geleert.
- Mit der Methode `update()` kann man ein zweites Dictionary in ein Dictionary einhängen. Enthält das zweite Dictionary Schlüssel, die auch im ersten vorkommen, so werden diese mit den Werten des zweiten überschrieben.

- **Tipp** Der Zugriff auf nicht existierende Schlüssel ist bei einigen Funktionen/Methoden im Zusammenhang mit Dictionaries ein gewisses Problem. Aber mit der folgenden Überprüfung kann man diesem Fehler vorbeugen:
- ```
if "key" in dictionary.keys():...
```

### 10.6.2 Ein Beispiel zum allgemeinen Umgang mit Dictionaries

Betrachten wir ein Beispiel (*Dic1.py*), das einige Techniken im Umgang mit Dictionaries als auch anderen sequenziellen Datentypen durch etwas „extreme“ Zugriffe zeigt:

```
person = ("Meier", "Otto")
print(person[0])
person2 = {"vorname": "Otto", "nachname": "Meier"}
print(type(person2))
print(person2)
print(person2["vorname"])
for v in person2.values():
 print(v)
print(person2.keys())
numindex = {1: 0, 2: True}
boolindex={True: False, False: True}
print(boolindex[True])
tupelindex={(1, 2): True, (1, 2, 3): False}
print(tupelindex[(1, 2, 3)])
#warumgehtdasnicht = {[1, 2]: True, [1, 2, 3]: False}
```

Das ist die Ausgabe:

```
Meier
<class 'dict'>
{'vorname': 'Otto', 'nachname': 'Meier'}
Otto
Otto
Meier
```

```
dict_keys(['vorname', 'nachname'])
```

```
False
```

```
False
```

Diese ergibt sich wie folgt:

- Zuerst wird in dem Beispiel ein Tupel *person* angelegt und danach auf den ersten Eintrag darin zugegriffen.
- Dann sehen Sie ein Dictionary *person2*. Die Ausgabe mit *type()* zeigt, dass es vom Typ *dict* ist.
- Die *print()*-Funktion kann direkt auf alle sequenziellen Datentypen angewendet werden und gibt dann deren vollständigen Inhalt aus. Das geht auch bei Dictionaries.
- Im Folgeschritt wird über den Textindex auf genau einen Wert im Dictionary zugegriffen.
- Mit dem Membership-Operator *in* kann man auch hervorragend in einer *for*-Schleife über ein Dictionary iterieren. Die Variable der Schleife ist der Wert, der aus der Wertemenge geholt wird, die mit *values()* ermittelt wird.
- Mit der Methode *keys()* erhalten Sie wie erwähnt ein Element vom Typ *dict\_keys*. Das sind alle Schlüssel, die in dem Dictionary vorhanden sind.
- Mit dem Dictionary *daten* sehen Sie, dass man auch Zahlen als Schlüssel verwenden kann.
- Sogar boolesche Schlüssel sind möglich, was das Dictionary *boolindex* demonstriert.
- Selbst Tupel kann man verwenden, was Sie mit dem Dictionary *tupelindex* erkennen können.
- Aber veränderliche Datentypen lassen sich nicht als Schlüssel verwenden. Die auskommentierte Codezeile würde das versuchen, und das würde einen Fehler bewirken.

### 10.6.3 Ein Dictionary aktualisieren oder erweitern

Das folgende Listing wäre ein Beispiel (*Dic2.py*), in dem ein leeres Dictionary nachträglich aktualisiert beziehungsweise erweitert wird. Ein Anwender kann in einer Schleife den Namen einer Person und deren Firma eingeben. Der Name wird der Schlüssel und die Firma der Wert. Jedes neue Key-Value-Paar wird dem Dictionary hinzugefügt. Sollte ein Schlüssel bereits vorhanden sein, wird in dem Wertepaar der vorhandene Wert ersetzt (Abb. 10.9).

```
teilnehmer = {}
op = ""
while op != "A":
 name = input("Name\n")
 firma = input("Firma\n")
 teilnehmer.update({name:firma})
 print(teilnehmer)
 op = input("(A)bbruch oder weiteren Wert aktualisieren/hinzufügen?\n")
```

```

Name
Ralph
Firma
RJS EDV-KnowHow
{'Ralph': 'RJS EDV-KnowHow'}
(A)bbruch oder weiteren Wert aktualisieren/hinzufügen?
W
Name
Felix
Firma
Weingut St-Urban
({'Ralph': 'RJS EDV-KnowHow', 'Felix': 'Weingut St-Urban'})
(A)bbruch oder weiteren Wert aktualisieren/hinzufügen?
W
Name
Florian
Firma
Weingut St-Urban
({'Ralph': 'RJS EDV-KnowHow', 'Felix': 'Weingut St-Urban', 'Florian': 'Weingut St-Urban'})
(A)bbruch oder weiteren Wert aktualisieren/hinzufügen?
W
Name
Felix
Firma
RJS EDV-KnowHow
({'Ralph': 'RJS EDV-KnowHow', 'Felix': 'RJS EDV-KnowHow', 'Florian': 'Weingut St-Urban'})
(A)bbruch oder weiteren Wert aktualisieren/hinzufügen?
A
>>>

```

**Abb. 10.1.9** Das Dictionary wurde erweitert und Einträge wurden modifiziert

#### 10.6.4 Iteration über ein Dictionary

Mit der *for*-Schleife kann man auf einfache Weise über die Schlüssel eines Dictionaries iterieren, was oben bereits angedeutet wurde:

```
d = {"a":123, "b":34, "c":304, "d":99}
for key in d:
 print(key)
```

Man kann aber auch die Methode *keys()* benutzen, die einem speziell die Schlüssel liefert:

```
for key in d.keys():
 print(key)
```

Mit der Methode *values()* iteriert man hingegen direkt über die Werte:

```
for value in d.values():
 print(value)
```

Auch das geht, um die Werte zu erhalten:

```
for key in d:
 print(d[key])
```

---

## 10.7 Mengen

**Mengen** bedeuten eine **ungeordnete** Zusammenfassung von bestimmten wohlunterschiedenen Dingen. Der Datentyp *set* dient in Python zum Erstellen solch einer ungeordneten Sammlung von einmaligen und unveränderlichen Elementen. In anderen Worten: Ein Element kann in einem *set*-Objekt nicht mehrmals vorkommen, was bei Listen und Tupel jedoch möglich ist.

Das Erzeugen einer Menge geht mit der Built-in Function *set()*. Als Parameter können beliebige Daten angegeben werden.

Beispiel:

```
x = set("Lovely spam, wonderful spam.")
```

Sets sind wie gesagt so implementiert, dass sie keine veränderlichen („mutable“) Objekte erlauben. Damit sind beispielsweise keine Listen als Elemente erlaubt.

Auch wenn Sets keine veränderlichen Elemente enthalten können, sind sie jedoch selbst veränderlich. Wir können zum Beispiel neue Elemente einfügen. Dazu gibt es die Methode *add()*.

Beispiel:

```
staedte = set(["Mainz", "Eppstein"])
staedte.add("Bodenheim")
```

Man kann Mengen aber auch unveränderlich („immutable“) machen, und zwar mittels sogenannter **Frozensests**.

Beispiel:

```
staedte = frozenset(["Frankfurt", "Eppstein"])
```

### 10.7.1 Vereinfachte Notation

Mengen kann man auch ohne Benutzung der Built-in Function *set()* definieren. Dazu benutzt man einfach die geschweiften Klammern.

Beispiel:

```
standorte = {"Bodenheim", "Eppstein"}
```

Das Verfahren ist ähnlich wie bei Dictionaries, nur ohne Schlüssel.

### 10.7.2 Operationen auf set-Objekten

Menge bieten ein paar interessante Operationen, etwa folgende:

- Mit der Methode *add()* fügt man ein Objekt in eine Menge als neues Element ein, falls es noch nicht vorhanden ist. Sie können aber nicht angeben, wo es genau in der Menge eingefügt wird.
- Mit *clear()* werden alle Elemente einer Menge entfernt.
- Mit *copy()* wird eine flache Kopie einer Menge erzeugt.
- Über *difference()* erhalten Sie die Differenz von zwei oder mehr Mengen.
- Die Methode *difference\_update()* entfernt alle Element einer anderen Menge aus einer Menge.
- Mit *discard()* wird ein Element aus einer Menge entfernt, falls es enthalten ist. Falls es nicht in der Menge enthalten ist, passiert nichts.
- Die Methode *remove()* funktioniert wie *discard()*, aber falls das Element nicht in der Menge enthalten ist, wird ein Fehler generiert, das heißt ein *KeyError*.
- Mit *intersection()* bekommt man die Schnittmenge von zwei Mengen.
- Die Methode *isdisjoint()* liefert *True*, wenn zwei Mengen eine leere Schnittmenge haben.
- Die Methode *issubset()* liefert *True* zurück, falls der Parameter eine Untermenge ist. Analog liefert *issuperset()* *True* zurück, falls der Parameter eine Obermenge ist.
- Die Methode *pop()* liefert das erste Element der Menge zurück. Dieses Element wird dabei aus der Menge entfernt. Die Methode erzeugt einen *KeyError*, falls die Menge leer ist. Beachten Sie, dass eine Menge ungeordnet ist. Deshalb ist es nicht immer eindeutig, welches Element das erste Element in der Menge ist.

Beispiel (*Menge.py*):

```
menge1 = set()
i = 1
while i < 10:
 menge1.add(i * 3)
 i += 1
print(menge1)
```

```

menge2 = {"a", "b", "c", "d"}
print(menge2)
i = 0
while i < 5:
 menge2.add(menge1.pop())
 i += 1
print(menge2)
print(menge1)

```

In dem Beispiel werden zwei Mengen erzeugt. Die erste Menge besteht aus zehn Zahlen, die ein Vielfaches der Zahl 3 sind. Die zweite Menge enthält die Strings (Buchstaben) "a", "b", "c" und "d". Nun zeigt die generierte Ausgabe bei der zweiten Menge bei mehrfachem Ausführen eine unterschiedliche Reihenfolge, wie diese Strings in der Menge auftauchen (Abb. 10.10). Und auch wenn man mit *pop()* aus der ersten Menge jeweils das erste Element entfernt und dann mit *add()* in die zweite Menge einführt, wird die Ausgabe der zweiten Menge die Elemente in unterschiedlicher Reihenfolge zeigen. Das ist aber eben ein Wesen von Mengen – sie sind ausdrücklich nicht geordnet.

**Abb. 10.10** Mengen sind ungeordnet

```

>>>
=====
RESTART: F:/PythonQuell
{3, 6, 9, 12, 15, 18, 21, 24, 27}
{'c', 'd', 'a', 'b'}
{'c', 3, 'a', 6, 9, 12, 15, 'b', 'd'}
{18, 21, 24, 27}

>>>
=====
RESTART: F:/PythonQuell
{3, 6, 9, 12, 15, 18, 21, 24, 27}
{'d', 'a', 'b', 'c'}
{3, 6, 9, 'b', 'd', 12, 15, 'a', 'c'}
{18, 21, 24, 27}

>>>
=====
RESTART: F:/PythonQuell
{3, 6, 9, 12, 15, 18, 21, 24, 27}
{'d', 'c', 'a', 'b'}
{3, 6, 'd', 9, 12, 'b', 15, 'c', 'a'}
{18, 21, 24, 27}

>>>
=====
RESTART: F:/PythonQuell
{3, 6, 9, 12, 15, 18, 21, 24, 27}
{'d', 'b', 'a', 'c'}
{'d', 3, 6, 9, 12, 15, 'b', 'c', 'a'}
{18, 21, 27}
>>> |

```

## 10.8 Operatoren bei sequenziellen Datentypen

Mit dem String-Verkettungsoperator kennen Sie bereits einen Operator, der im Grunde ein allgemeiner Verkettungsoperator bei sequenziellen Datentypen ist. Er hat eben nur bei Zeichenketten die recht triviale Wirkung der **String-Verkettung**.

Aber man kann auch andere Operatoren auf sequenzielle Datentypen anwenden. Das führt zu interessanten Ergebnissen.

### 10.8.1 Der Plusoperator

Der Plusoperator führt bei etwa bei Tupeln und Listen dazu, dass einfach die Elemente des zweiten Operanden an den ersten angehängt werden.

- ▶ Auf Mengen kann man den Plusoperator **nicht** anwenden. Dort nimmt man die Methode `add()`. Auch bei Dictionaries geht es nicht.

Beispiel (*TupelListeVerkettung.py*):

```
tupelzahlen1 = (2, 3, 5, 7)
tupelzahlen2 = (11, 13, 17, 19)
primtupelzahlen = tupelzahlen1 + tupelzahlen2
print(primtupelzahlen)
listenzahlen1 = [2, 3, 5, 7]
listenzahlen2 = [11, 13, 17, 19]
primlistenzahlen = listenzahlen1 + listenzahlen2
print(primlistenzahlen)
```

Das führt dazu:

```
(2, 3, 5, 7, 11, 13, 17, 19)
[2, 3, 5, 7, 11, 13, 17, 19]
```

Der Effekt ist offensichtlich und sicher kaum überraschend.

### 10.8.2 Multiplikationen mit sequenziellen Datentypen

Aber man kann auch Multiplikationen mit sequenziellen Datentypen durchführen, etwa so (*TupelListeMultiplikation.py*):

```
tupelzahlen1 = (2, 3, 5, 7)
zieltupelzahlen = tupelzahlen1 * 4
print(zieltupelzahlen)
```

```
listenzahlen1 = [2, 3, 5, 7]
ziellistenzahlen = listenzahlen1 * 2
print(ziellistenzahlen)
```

Das führt dazu:

```
(2, 3, 5, 7, 2, 3, 5, 7, 2, 3, 5, 7, 2, 3, 5, 7)
[2, 3, 5, 7, 2, 3, 5, 7]
```

Der Effekt ist auch offensichtlich. Die Zahl der Elemente wurde verdoppelt, und jedes Tupel beziehungsweise jede Liste wird so oft wiederholt, wie es der Multiplikator angibt.

- ▶ Auf Mengen und Dictionaries kann man den Multiplikationsoperator **nicht** anwenden.

### 10.8.3 Inhalt überprüfen

Bei allen sequenziellen Datentypen funktionieren die Membership-Operatoren *in* oder *not in*,<sup>6</sup> um Inhalte zu überprüfen, wobei man festhalten muss, dass bei Dictionaries die Anwendung etwas trickreich ist.

- ▶ Man wird bei Dictionaries in der Regel nicht direkt über das Dictionary suchen, sondern entweder über die Schlüssel (erhält man mit *keys()*) oder die Werte (erhält man mit *values()*). Dabei sollte man beachten, dass bei der Suche direkt in der Struktur auf eine exakte Gleichheit geprüft wird – sowohl bei den Schlüsseln als auch den Werten! Sehr oft ist es daher sinnvoll, dass Sie gezielt über einen Schlüssel ein Wertepaar auswählen und dann im Inhalt suchen.

```
tupelzahlen = (2, 3, 5, 7)
print(tupelzahlen)
print("8 in tupelzahlen:\t", 8 in tupelzahlen)
print("8 not in tupelzahlen:\t", 8 not in tupelzahlen)
tupelperson = ("Hans", "Wurst", True)
print(tupelperson)
print("\\"Hans\\" in tupelperson[1]:\t", "Hans" in tupelperson)
print("\\"Hansi\\" in tupelperson[1]:\t", "Hansi" in tupelperson)
print("\\"Ha\\" in tupelperson:\t", "Ha" in tupelperson)
print("\\"Ha\\" in tupelperson[0]:\t", "Ha" in tupelperson[0])
print("\\"Ha\\" in tupelperson[1]:\t", "Ha" in tupelperson[1])
print("True not in tupelperson:\t", True not in tupelperson)
listenzahlen = [11, 13, 17, 19]
```

---

<sup>6</sup>Was streng genommen nur die Negierung des *in*-Operators darstellt.

```

print(listenzahlen)
print("19 in listenzahlen:\t", 19 in listenzahlen)
mengezahlen = {23,29}
print(mengezahlen)
print("29 in listenzahlen:\t", 29 in mengezahlen)
dicperson = {"vorname" : "Otto", "nachname" : "Meier", "alter" : 61,
 1 : True }
print(dicperson)
print("\\"Ot\\" in dicperson:\t", "Ot" in dicperson)
print("\\"Ot\\" in dicperson.values():\t", "Ot" in dicperson.values())
print("\\"Otto\\" in dicperson.values():\t", "Otto" in dicperson.va-
lues())
print("\\"Ot\\" in dicperson[\"vorname\"]:\t", "Ot" in dicperson["vor-
name"])
print("\\"vor\\" in dicperson.keys():\t", "vor" in dicperson.keys())
print("\\"vorname\\" in dicperson.keys():\t", "vorname" in dicperson.
keys())

```

Das Beispiel führt zu einigen sehr interessanten Ergebnissen, und es ist sinnvoll, sich jeden Schritt mit dem resultierenden Ergebnis genau anzusehen (Abb. 10.11).

```

(2, 3, 5, 7)
8 in tupelzahlen: False
8 not in tupelzahlen: True
('Hans', 'Wurst', True)
"Hans" in tupelperson[1]: True
"Hansi" in tupelperson[1]: False
"Ha" in tupelperson: False
"Ha" in tupelperson[0]: True
"Ha" in tupelperson[1]: False
True not in tupelperson: False
[11, 13, 17, 19]
19 in listenzahlen: True
{29, 23}
29 in listenzahlen: True
{'vorname': 'Otto', 'nachname': 'Meier', 'alter': 61, 1: True}
"Ot" in dicperson: False
"Ot" in dicperson.values(): False
"Otto" in dicperson.values(): True
"Ot" in dicperson["vorname"] : True
"vor" in dicperson.keys(): False
"vorname" in dicperson.keys(): True
>>>

```

**Abb. 10.11** Inhaltsüberprüfungen bei sequenziellen Datentypen

- Mit `tupelzahlen = (2, 3, 5, 7)` wird ein einfaches Tupel angelegt und die Anweisungen `8 in tupelzahlen` und `8 not in tupelzahlen` sollten keine Überraschungen liefern.
  - Mit `tupelperson = ("Hans", "Wurst", True)` wird ein weiteres Tupel angelegt.
    - Die Anweisungen `"Hans" in tupelperson` und `"Hansi" in tupelperson` sowie `"Ha" in tupelperson` suchen einen Teststring in dem Tupel. Entweder ist dieser ganz oder teilweise enthalten. Dann wird als Ergebnis `True` geliefert. Oder er ist nicht enthalten und dann ist das Ergebnis `False`. Das ist der eher triviale Part.
    - Aber mit `"Ha" in tupelperson[0]` und `"Ha" in tupelperson[1]` suchen wir nicht in dem Tupel direkt, sondern in einem Element darin. Und das ist dann die Suche in einem String!
    - Mit `True not in tupelperson` suchen wir jedoch wieder unmittelbar in dem Tupel als Struktur. Aber dieses Mal nach einem booleschen Literal (`True` oder `False`), und wir negieren den Wert (wir drehen ihn also um).
  - Die nächsten Schritte suchen in einer Liste und einer Menge, dazu ist nur bemerkenswert, dass es hier keine Besonderheiten zu beachten gibt.
  - Diese Besonderheiten gibt es aber beim Dictionary:
    - Mit `"Ot" in dicperson` suchen wir einen String im Dictionary. Dieser ist zwar Bestandteil eines der Werte in dem Dictionary ("Otto"), aber es gibt keine vollständige Übereinstimmung. Und deshalb ist das Ergebnis des Tests `False`!
    - Auch in der Menge der Werte, die mit `dicperson.values()` ermittelt wird, wird `False` das Ergebnis sein (`"Ot" in dicperson.values()`).
    - Wir erhalten aber `True` bei `"Otto" in dicperson.values()`. Warum? Hier haben wir die vollständige Übereinstimmung.
    - Und auch `"Ot" in dicperson["vorname"]` liefert `True`. Warum? Hier suchen wir in dem Wert des einzelnen Wertepaares.
    - Bei `"vor" in dicperson.keys()` erhalten wir wieder `False`, denn auch bei Schlüsseln brauchen wir die exakte Übereinstimmung, und die liegt hier nicht vor.
    - Aber bei `"vorname" in dicperson.keys()` liegt eine solche exakte Übereinstimmung vor, deshalb liefert das `True`.
- Es gibt noch eine Reihe an weiteren Operationen oder Funktionen, die überwiegend selbsterklärend sind. So bekommt man mit den Funktionen `min()` und `max()` die kleinsten und größten Werte in einem Tupel mit Zahlen, wobei das auch für Texte geht – dann wird der Zeichencode der Zeichen herangezogen. Für mehr Informationen sei auf die Dokumentation von Python verwiesen, die Sie in IDLE über die Hilfe erreichen.

---

## 10.9 Über sequenzielle Strukturen iterieren

Oft möchte man bei einer sequenziellen Datenstruktur über alle Elemente iterieren. Dazu gibt einige Möglichkeiten:

- Die Größe einer Struktur bestimmen und eine „normale“ Schleife verwenden.
- Einen Iterator verwenden.

Wenn Sie über eine sequenziellen Datenstruktur iterieren wollen, können Sie zum Beispiel bei Strukturen mit einer Sortierung und einem numerischen Index mit `len()` deren Größe bestimmen und dann mit dem Index auf die einzelnen Elemente zugreifen.

Sie können aber auch einen sogenannten **Iterator** verwenden. Ein Iterator bezeichnet einen Zeiger, der es erlaubt, über die Elemente einer abzählbaren Struktur zu iterieren. Der Vorteil eines Iterators ist, dass man keine Kenntnis der Datenstruktur oder der Implementierung haben muss. Obwohl man selbst einen Iterator erzeugen kann, ist das in Python eigentlich unnötig. Denn es arbeitet die `for`-Schleife implizit mit einem Iterator. Die sequenziellen Basistypen sowie ein Großteil der Klassen der Standardbibliothek von Python unterstützen Iterationen. Auch der Datentyp Dictionary (`dict`) unterstützt die Iteration. In diesem Fall läuft die Iteration über die Schlüssel des Dictionaries. Beispiel (`Iteration1.py`):

```
primzahlen = (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31)
i = 0;
while i < len(primzahlen):
 print(primzahlen[i]);
 i+=1;
for x in primzahlen:
 print(x);
```

---

## 10.10 Pattern-Matching

Bereits bei der Behandlung von Kontrollstrukturen wurde das Thema Pattern-Matching kurz angesprochen, als die mit Python 3.10 neu eingeführte *match-case*-Anweisung vorgestellt wurde. Deren hauptsächliche Leistungsfähigkeit zeigt sich aber erst bei besagtem Pattern-Matching, das wiederum meist sequenzielle Datenstrukturen als Basis benötigt. Von daher passt das Thema wunderbar in dieses Kapitel.

Wie wir schon gesehen haben, können Testfälle bei der *match-case*-Anweisung auch beliebige Strings sein. Das sind aber – wie wir uns in diesem Kapitel klargemacht haben – sequenzielle Datenstrukturen, die sehr viele Gemeinsamkeiten mit Tupeln oder Listen haben.

In Python bezieht sich **Pattern-Matching** darauf, eine spezifische Form oder Struktur in Daten zu erkennen und mit diesem Muster übereinstimmende Teile zu identifizieren. Pattern-Matching ist wie das Finden von Puzzleteilen, die in ein bestimmtes Muster (also an eine gewisse Stelle im Puzzle) passen. Das ermöglicht, komplexe Muster in Daten effizient zu suchen und darauf zu reagieren.

- Wer mit dem Thema vertraut ist – Pattern-Matching ist in gewisser Weise mit dem Umgang mit regulären Ausdrücken zu vergleichen, nur signifikant einfacher.

In Python werden folgende grundlegende Konzepte des Pattern-Matching verfolgt:

- Pattern sind spezielle Ausdrücke, die verwendet werden, um nach bestimmten Strukturen in Daten zu suchen. Diese Strukturen können Werte, Datentypen oder auch Kombinationen davon sein.
- Beim Pattern-Matching kann man Wildcards, oft als Unterstrich (`_`), verwenden, um Teile der Daten zu ignorieren.
- Es gibt beim Pattern-Matching das Konzept der Variablenbindung. Damit werden Werte aus den übereinstimmenden Mustern in Variablen aus den Mustern extrahiert.

Hier ist nun ein erstes einfaches Beispiel für explizites Pattern-Matching in Python (*Patternmatch1.py*):

Beispiel

```
def treffer(data):
 match data:
 case (1, 2):
 print("Das Muster ist (1, 2)")
 case (x, _):
 print(f"Das Muster enthält {x} als erstes Element")
 case _:
 print("Kein Muster passt")
treffer((1,2))
treffer((7,5))
treffer((3,"Beliebiger Text"))
treffer((2,3,5,7,11))
treffer((1,3,5,7,11))
treffer((1,(3,5,7,11)))
```

Beachten Sie, dass der zu testende Ausdruck kein primitiver Wert ist, sondern in dem Fall ein Tupel, also eine sequenzielle Datenstruktur. Wobei das aus Sicht von Python gar nicht so ungewöhnlich erscheinen sollte, denn Strings sind ja wie gesagt auch sequenzielle Datenstrukturen, und man geht diese Philosophie mit *match-case* und Pattern-Matching einfach konsequent weiter.

In diesem Beispiel wird eine Funktion mit dem Übergabeparameter *data* deklariert. Bei Aufruf werden verschiedene Muster (Tupel) für die Variable *data* überprüft. Je nachdem, welches Muster übereinstimmt, wird der entsprechende Codeblock ausgeführt.

- Im ersten *case*-Block wird überprüft, ob der Wert von *data* genau dem Muster (1, 2) entspricht. Wenn es übereinstimmt, wird der Text „Das Muster ist (1, 2)“ ausgegeben.

- Im zweiten *case*-Block wird überprüft, ob der Wert von *data* dem Muster (x, \_) entspricht. Das Zeichen "\_" ist eine Wildcard und bedeutet, dass das zweite Element des Tupels ignoriert wird. Es wird also ein beliebiges Tupel mit zwei Elementen gesucht. Wenn das Muster übereinstimmt, wird der Wert von "x" ausgegeben. Das ist der Wert des ersten Elements in dem Tupel. Dann wird etwa in dem Fall (7, 5) der Text „Das Muster enthält 7 als erstes Element“ ausgegeben.
- Im dritten *case*-Block wird die Wildcard "\_" verwendet, um jedes mögliche Muster zu akzeptieren, das nicht zu den beiden vorherigen *case*-Blöcken passt. In diesem Beispiel passen die Muster (2,3,5,7,11) und (1,3,5,7,11) nicht (Tupel hat zu viele Elemente), also wird dieser Block ausgeführt und „Kein Muster passt“ wird ausgegeben.

Beachten Sie, dass ein String in dem zu testenden Tupel als ein Element gesehen wird. Ebenso ein weiteres Tupel, weshalb *treffer((9,(3,5,7,11)))* zur Ausgabe „Das Muster enthält 9 als erstes Element“ führt.

Das Suchen in den Mustern kann auch über verschiedene Datentypen gehen. Angenommen, es liegen Daten vor, die aus Zahlen und Buchstaben bestehen, und nur die Zahlen sollen extrahiert werden. Das geht mit Pattern-Matching und zeigt nebenher die Ähnlichkeit zu regulären Ausdrücken. Nur geht das hier viel einfacher (*Patternmatch2.py*):

Beispiel

```
data = [1, 'A', 2, 'B']
for i in data:
 match i:
 case int(x):
 print(f'Gefundene Zahl: {x}')
 case _:
 pass # Wir ignorieren alles andere
```

In diesem Beispiel wird über die Daten iteriert, und für jede Komponente wird überprüft, ob sie eine Zahl ist. Wenn ja, wird sie ausgegeben. Wenn es sich um etwas anderes handelt, wird es ignoriert.

- Beachten Sie, dass **keine** Ausnahme ausgelöst wird, wenn die Typumwandlung mit *int()* keine Zahl liefert. Das ist ein bemerkenswertes Verhalten und umgeht die Notwendigkeit, mit Ausnahmebehandlung zu arbeiten.

Betrachten wir nun ein Beispiel mit Pattern-Matching, wo der Umgang mit Wildcards genauer analysiert werden soll. Angenommen, wir haben eine Liste von Wörtern, und wir möchten alle Wörter ausdrucken, die mit dem Buchstaben "A" beginnen. Wir werden das Pattern-Matching verwenden, um nur die Wörter, die zu unserem Muster passen, auszuwählen (*Patternmatch3.py*).

```

woerter = ["Apfel", "Banane", "Erdbeere", "Ananas", "Birne",
"Pflaume"]
for w in woerter:
 match w:
 case "A":
 print(f"Das Wort beginnt mit 'A': {w[1:]}")
 case "Ba":
 print(f"Das Wort beginnt mit 'Ba': {w[2:]}")
 case _:
 print(f"Das Wort beginnt weder mit 'A' noch 'Ba': {w}")

```

- In der ersten *case*-Anweisung erwarten wir den Buchstaben "A" gefolgt von allem anderen, was im Wort kommt. Mit *w[1:]* haben wir eine Variable, in der der Rest des Worts gespeichert wird.
- In der zweiten *case*-Anweisung verwenden wir das Muster "Ba" und setzen eine Übereinstimmung der ersten beiden Buchstaben voraus.
- In der dritten *case*-Anweisung fangen mit einem Unterstrich ("\_") alle anderen Wörter auf, die nicht mit "A" oder "Ba" beginnen. Beachten Sie, dass wir auch hier den Rest von dem Testwert selbst erhalten.

Kommen wir nun zur Angabe von Alternativen bei den Testfällen im Pattern-Matching (*Patternmatch4.py*).

```

bestellung = input("Geben Sie Ihre Bestellung auf\n")
Pattern Matching
match bestellung.split():
 case ['Bier']:
 print('Prost.')
 case ['Wein' | "Apfelwein"]:
 print('Wohl bekomm\'s.')
 case ['Limo' | "Saft" | "Wasser"]:
 print('Sie müssen wohl noch fahren?')
 case _:
 print('Leider haben wir', bestellung, 'nicht vor-'
rätig.')

```

Dieser Python-Code ist ein einfaches Programm, das eine Benutzereingabe erwartet und dann mithilfe von Pattern-Matching entscheidet, welche Art von Getränk bestellt wurde und entsprechend reagiert.

- Zuerst fragt das Programm den Benutzer nach seiner Bestellung und speichert die Eingabe in der Variablen *bestellung*.
- Dann kommt der Teil des Pattern-Matchings. Hier wird die Eingabe, die in *bestellung* gespeichert ist, in eine Liste von Wörtern aufgeteilt. Die *split*-Methode trennt die Eingabe am Leerzeichen, was dazu führt, dass Wörter getrennt werden.
- Nun erfolgt das Pattern-Matching selbst:
- Mit *case ['Bier']*: wird überprüft, ob die Eingabe genau "Bier" ist. Wenn ja, wird "Prost." ausgegeben. Dies ist der Fall, wenn der Benutzer nur "Bier" bestellt hat.
- Bei *case ['Wein'|'Apfelwein']*: wird überprüft, ob die Eingabe entweder "Wein" oder "Apfelwein" ist. Wenn ja, wird "Wohl bekomm's." ausgegeben. Das ist der Fall, wenn der Benutzer entweder "Wein" oder "Apfelwein" bestellt.
- Bei *case ['Limo'|'Saft'|'Wasser']*: gibt es drei Alternativen. Es wird überprüft, ob die Eingabe "Limo", "Saft" oder "Wasser" ist. Wenn ja, wird „Sie müssen wohl noch fahren?“ angezeigt. Dies ist der Fall, wenn der Benutzer eines dieser Getränke bestellt.
- Der Default-Fall *case \_*: nutzt einen Wildcard-Fall, der alle anderen Bestellungen abfängt, die nicht zu den oben genannten Mustern passen. Hier wird die ursprüngliche Bestellung (der gesamte Text) in der Ausgabe verwendet, um dem Benutzer mitzuteilen, dass das bestellte Getränk nicht vorrätig ist.

Der Code ermöglicht es also, die Benutzereingabe auf einfache Weise in Kategorien (Getränke) zu unterteilen und basierend auf dieser Kategorie eine entsprechende Antwort auszugeben. Das Pattern-Matching erleichtert die Verarbeitung verschiedener Bestellungen, ohne viele komplizierte *if-elif-else*-Anweisungen verwenden zu müssen.

Kommen wir nun zu einem Beispiel, das einen Testwert aufsplittet und dann die resultierenden Segmente mittels Pattern-Matching prüft (*Patternmatch5.py*).

#### Beispiel

```
zugangsdaten = input("Geben Sie Ihre Zugangskennung ein (Userid Password)\n")
Pattern Matching
match zugangsdaten.split():
 case ["root", "geheim"]:
 print("Hallo root.")
 case ["admin", "123"]:
 print("Hallo", zugangsdaten.split()[0], ".")
 case _:
 print("Zugangsdaten falsch.")
```

Dieser Python-Code erfasst Zugangsdaten in Form von Benutzer-ID (Userid) und Passwort von einem Benutzer und verwendet Pattern-Matching, um auf diese Eingaben zu reagieren. Mit *match zugangsdaten.split():* sehen Sie den Beginn der Pattern-Matching-Block. Es wendet *split()* auf die Eingabe an, um sie in eine Liste von Wörtern oder

Token aufzuteilen. Dies ist nützlich, um Benutzer-ID und Passwort voneinander zu trennen.

Mit den folgenden *case*-Anweisungen werden dann die resultierenden Listen mit der Userid und dem Passwort auf Übereinstimmung geprüft.

Zusammengefasst: Dieser Code erwartet Benutzer-Zugangsdaten in der Form „Benutzer-ID Passwort“, trennt sie auf und verwendet Pattern-Matching, um auf verschiedene Benutzerkombinationen zu reagieren. Wenn die Eingabe mit den vordefinierten Benutzern und Passwörtern übereinstimmt, werden spezifische Begrüßungsnachrichten ausgegeben. Andernfalls wird eine allgemeine Fehlermeldung angezeigt.

Das letzte Beispiel zu Pattern-Matching baut auf dem vorherigen auf, kombiniert aber mehrere mögliche Trefferfälle (*Patternmatch6.py*).

#### Beispiel

```
eingabe = input("Geben Sie Ihre Zugangskennung ein (Userid Password)\n")
zugangsdaten=eingabe.split()
Pattern Matching
match zugangsdaten:
 case (['root','geheim'] | ['admin','123']): # Kombination von Oder
 und sequenzieller Datenstruktur - Datentypen beachten
 print('Hallo Meister.')
 case _:
 print('Zugangsdaten falsch.')
```

Die entscheidende Erweiterung gegenüber dem vorherigen Beispiel ist, dass in dem Muster *case* (*['root', 'geheim'] | ['admin', '123']*): zwei Kombinationen von Benutzerkennung und Passwort überprüft werden. Die vertikale Trennlinie (^|) fungiert als logisches Oder.

---

## 10.11 Comprehensions

Im Zusammenhang mit sequenziellen Datenstrukturen und dem Iterator gibt es in Python eine sehr interessante Konstruktion, die ähnlich wie Lambda-Ausdrücke zu sehen ist, diese aber noch einmal vereinfacht und auch schon vor der Einführung von Lambda-Ausdrücken vorhanden war. Betrachten Sie diesen Beispielcode (*Comprehension1.py*):

#### Beispiel

```
print([chr(i) for i in range(256)])
text = "Hallo"
print([chr(ord(i)+3) for i in text])
```

In dem Beispiel werden zwei Listen generiert unter Zuhilfenahme sogenannter **List Comprehensions**. In Python sind Comprehensions eine praktische Methode, um Listen, Mengen und Dictionaries auf eine kompakte Weise zu erstellen. Dazu werden entweder geschweifte Klammern oder eckige Klammern notiert und im Inneren ein Iterator mit vorangestelltem Ergebnis notiert. Es gibt also verschiedene Arten von Comprehensions:

- List Comprehensions
- Set Comprehensions
- Dictionary Comprehensions

Bei dem ersten Beispiel wird mit `print([chr(i) for i in range(255)])` eine Liste erstellt, welche die Zeichen enthält, die den Unicode-Werten von 0 bis 256 entsprechen. Der Ausdruck `[chr(i) for i in range(256)]` ist wie gesagt eine List Comprehension, die durch eine Schleife geht und für jeden Wert *i* im Bereich von 0 bis 255 das entsprechende Zeichen mithilfe von `chr(i)` in die Liste einfügt. Das Ergebnis ist eine Liste von Zeichen.

Die Anweisung `chr(i)` gibt das Zeichen zurück, das dem Unicode-Wert *i* entspricht. Daher erhält man eine Liste von Zeichen, die die ersten 256 Unicode-Zeichen repräsentieren.

Teil 2 des Beispiels setzt mit nur einer Codezeile eine Verschlüsselung eines Texts mithilfe von Caesar-Verschlüsselung (auch Caesar-Chiffre) um. Hier wird der Text "Hallo" verschlüsselt, indem für jeden Buchstaben in "Hallo" der Unicode-Wert um 3 erhöht wird. Dies ist ein einfaches Beispiel für eine Caesar-Verschlüsselung, bei der jeder Buchstabe im Text um eine feste Anzahl von Positionen im Alphabet verschoben wird. Dabei gibt `ord(i)` den Unicode-Wert des Buchstabens *i* zurück und `chr(ord(i) + 3)` nimmt den Unicode-Wert des Buchstabens, erhöht ihn um 3 und gibt das entsprechende Zeichen zurück. Der Ausdruck `[chr(ord(i) + 3) for i in text]` erstellt eine Liste der verschlüsselten Buchstaben des Texts "Hallo". Das Ergebnis ist eine Liste von Zeichen, die den verschlüsselten Text repräsentieren. In diesem Fall wird "Hallo" zu "Kdoor". Die Entschlüsselung geht analog mit -3.

Hier ist noch ein weiteres Beispiel für eine List Comprehension (deutsch Listen-Comprehension) zur Erzeugung einer Liste von Quadratzahlen (*Comprehension2.py*).

Beispiel

```
print([x**2 for x in range(1, 6)])
```

Hier wird eine Liste erstellt und ausgegeben, die die Quadrate der Zahlen von 1 bis 5 enthält. Die Schleife `for x in range(1, 6)` durchläuft die Zahlen von 1 bis 5, und der Ausdruck `x**2` berechnet das Quadrat jeder Zahl aus dem Range.

Auch für **Set Comprehensions** (deutsch Mengen-Comprehensions) wollen wir ein Beispiel ansehen. Sie funktionieren ähnlich wie Listen-Comprehensions, aber sie erzeugen Mengen anstelle von Listen. Sie werden in geschweiften Klammern geschrieben. Das wäre ein Beispiel für die Erzeugung einer Menge von Quadratzahlen (*Comprehension3.py*):

### Beispiel

```
print({x**2 for x in range(1, 6)})
```

Zum Abschluss soll ein Beispiel zu **Dictionary Comprehensions** folgen. Hier haben Sie Schlüssel-Wert-Paare. Sie werden in geschweiften Klammern geschrieben, wobei ein Doppelpunkt den Schlüssel vom Wert trennt. Hier ist ein Beispiel zur Erzeugung eines Dictionaries mit Zahlen und ihren Quadraten.

### Beispiel

```
print({x: x**2 for x in range(1, 6)})
```

Hier wird ein Dictionary erstellt, in dem die Zahlen von 1 bis 5 den zugehörigen Quadraten zugeordnet werden.

- ▶ Beachten Sie, dass wir im Gegensatz zu Set Comprehensions ein Key-Value-System haben. Durch die ähnliche Schreibweise mit den geschweiften Klammern kann dies in Python leicht verwechselt werden.

Comprehensions sind recht offensichtlich sehr nützlich, um in Python Listen, Mengen und Dictionaries auf elegante und effiziente Weise zu erstellen, ohne komplexe Schleifen und temporäre Variablen verwenden zu müssen. Sie tragen damit auch zur Lesbarkeit des Codes bei.

---

## 10.12 Anwendung des Walrus Operator

Der Walross-Operator (`:=`) wurde bereits vorgestellt, aber für wirklich sinnvolle Anwendungen benötigt man Kontrollflussanweisungen. Aber auch sequenzielle Datentypen sind als Voraussetzung sinnvoll, um nicht ganz triviale Beispiele zeigen zu können. Da diese Python-Techniken jetzt erarbeitet wurden, soll der in Python 3.8 eingeführte Operator an dieser Stelle nun in einem Beispiel demonstriert werden. Wie schon beschrieben ermöglicht er es, einen Wert einer Variablen zuzuweisen und diesen Wert **gleichzeitig** in einem Ausdruck zu verwenden. Dies kann dazu beitragen, den Code kompakter und lesbarer zu gestalten. In dem nachfolgenden einfachen Beispiel wird das verdeutlicht (*walross.py*):

### Beispiel

```
zahlenliste = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -1, -2, -3]
Ohne den Walrus Operator:
positive_sum = 0
for num in zahlenliste:
 if num > 0:
 positive_sum += num
print("Summe der positiven Zahlen (ohne Walrus Operator):", positive_sum)
Mit dem Walrus Operator:
positive_sum = 0
for num in zahlenliste:
 if (positive_num := num) > 0:
 positive_sum += positive_num
print("Summe der positiven Zahlen (mit Walrus Operator):", positive_sum)
```

In diesem Beispiel haben wir eine Liste von Zahlen, aus der die Summe der positiven Zahlen berechnet werden soll. Dabei werden zwei Möglichkeiten gegenüber gestellt.

- In der Variante ohne den Walrus Operator wird eine Variable mit 0 initialisiert, und dann wird mit einer *for*-Schleife über die Liste iteriert. Innerhalb der Schleife wird überprüft, ob das jeweilige Element der Liste größer als 0 ist. Falls ja, wird der Wert zur Variable *positive\_sum* hinzu addiert. Dieser Ansatz erfordert die Verwendung einer separaten Variablen *num*, um den Wert zwischenspeichern.
- In der Variante mit dem Walrus Operator wird in der *for*-Schleife der Walrus Operator bei *positive\_num := num* direkt in der Bedingung verwendet. Das bedeutet, dass *num* unmittelbar der Variablen *positive\_num* zugewiesen und gleichzeitig überprüft wird, ob *positive\_num* größer als 0 ist. Wenn die Bedingung erfüllt ist, wird *positive\_num* zum Wert der Variable *positive\_sum* hinzugefügt. Dieser Ansatz ist kompakter und eliminiert die Notwendigkeit für eine separate Variable, um den Wert zwischenspeichern.
- Am Ende des Skripts werden die Summen der positiven Zahlen sowohl mit als auch ohne den Walrus Operator ausgegeben.

Beide Ansätze liefern das gleiche Ergebnis, aber der Ansatz mit dem Walrus Operator ist in diesem Fall etwas kürzer und möglicherweise leichter zu lesen.



# Objektorientierte Programmierung in Python – Klassen, Objekte, Eigenschaften und Methoden

11

## 11.1 Was behandeln wir in diesem Kapitel?

Python unterstützt bewusst mehrere Programmierparadigma und steht damit in Gesellschaft einiger älterer Sprachen wie PHP oder C/C++, aber auch moderner Sprachen wie F#. So können Sie auch objektorientierte Programmierung nutzen, was zwar schon erwähnt, aber bisher noch nicht explizit angewendet wurde. Python sieht sowieso im Grunde alles als Objekt – Klassen, Typen, Methoden, Module etc., auch primitive Datentypen, wobei der Datentyp jeweils an das Objekt (den Wert) gebunden ist und nicht an eine Variable. Das bedeutet, dass Datentypen in Python dynamisch vergeben werden, so wie bei Smalltalk oder Lisp, und nicht etwa wie bei Java. Von daher wurde also bisher schon immer in unseren Python-Beispielen mit Objekten gearbeitet, auch wenn die Zugriffswege das nicht erkennen ließen. Das lag nicht zuletzt daran, dass ausdrücklich nicht nach dem OO-Paradigma programmiert wurde, sondern dem funktionalen Paradigma. In diesem Kapitel werden wir uns nun auch ausdrücklich mit den OO-Konzepten in Python beschäftigen, bei deren Umsetzung Python teils sehr „eigenwillige“ Wege geht.

## 11.2 Hintergründe der OOP

Objektorientierte Programmierung ist kein wirklich neuer Ansatz. Im Gegenteil – die Idee der objektorientierten Programmierung reicht sogar bis Anfang der 1970er-Jahre und teilweise noch weiter zurück. Sehr frühe Vertreter von Programmiersprachen mit objektorientiertem Denkansatz sind **Lisp** oder **Logo**. In den 1970er-Jahren entstand als wichtigster Vertreter erster objektorientierter Sprachen **Smalltalk**, das sich in mehreren

Zyklen entwickelte und als einer der geistigen Väter von Java gilt. Aber erst mit **Java** in der Mitte der 1990er-Jahre hat sich die OOP richtig durchgesetzt.<sup>1</sup> Seit der Zeit entstand eine Reihe moderner, eigenständiger Programmiersprachen, die sich ausdrücklich als rein objektorientierte Programmiersprachen verstehen und mit prozeduralen Erb-lasten vollkommen brechen. Java zählt wie gesagt explizit dazu, oder von Microsoft die Sprache C#. Diese Programmiersprachen erzwingen einen streng objektorientierten Ansatz und unterstützen im Allgemeinen die wichtigsten Konzepte der objektorientierten Programmierung – allerdings in unterschiedlichem Maße. Andere Sprachen, die in dieser Zeit oder auch später entstanden, verfolgen einen hybriden Ansatz und erzwingen keine strenge objektorientierte Programmierung – wie eben auch Python oder JavaScript, PHP oder F#. Dabei kann man durchaus geteilter Meinung sein, ob man den sicheren und klaren OO-Ansatz wirklich aufweichen soll und diese freie Wahl des Programmierparadigmas wirklich den Königsweg darstellt.

### 11.2.1 Ziele der OOP – Wiederverwendbarkeit und bessere Softwarequalität

Das (!) Argument für die Verwendung der objektorientierten Programmierung und der Grund dafür, dass sie sich durchgesetzt hat, ist die **Wiederverwendbarkeit**. In der althergebrachten Programmierung<sup>2</sup> programmiert man prozedural beziehungsweise funktional. Dies bedeutet die Umsetzung eines Problems in ein Programm durch eine Folge von Anweisungen, die in einer festgelegten Reihenfolge auszuführen sind. Einzelne zusammengehörende Anweisungen werden dabei maximal in kleinere Einheiten von Befehlsschritten zusammengefasst (sogenannte Unterprogramme – Funktionen oder Prozeduren). Vor allen Dingen gilt, dass die Datenebene und die Anweisungsebene aufgetrennt werden können. Diesen Ansatz haben Sie bisher auch in allen unseren Python-Beispielen gesehen.

Das Problem prozeduraler/funktionaler Programmierung ist, dass bei einer solchen Aufteilung Änderungen in der Datenebene Auswirkungen auf die unterschiedlichsten Programmsegmente haben können und damit die Wiederverwendbarkeit von Unterprogrammen sehr eingeschränkt ist, da oft Abhängigkeiten gegenüber anderen Bestandteilen eines Programms existieren (beispielsweise globalen Variablen).

---

<sup>1</sup>C/C++ war zwar zu der Zeit auch schon sehr populär, aber eine hybride Sprache, in der man sowohl objektorientiert als auch funktional programmieren konnte. Und oft wurde in C/C++-Projekten nicht wirklich objektorientiert programmiert – was ich aus eigener Erfahrung weiß.

<sup>2</sup>Die historisch erste Generation der Programmierung mit Maschinensprache und die zweite Generation mit Assembler sollen hier nicht beachtet werden, sondern nur die höheren Programmiersprachen.

Objektorientierte Programmierung lässt sich im Vergleich zu diesem Paradigma der Programmierung darüber abgrenzen beziehungsweise definieren, dass zusammengehörende Anweisungen und Daten eine zusammengehörende, abgeschlossene und eigenständige Einheit bilden – Objekte! Und diese sind ohne Abhängigkeiten zu verwenden und ohne Randwirkungen intern zu ändern.

► Objektorientierte Programmierung hebt die Trennung von Daten- und Anweisungsebene auf.

Aber die objektorientierte Programmierung verfolgt noch ein Ziel. Bei der Entwicklung komplexer Softwaresysteme ist die wichtigste Aufgabe, eine möglichst hohe Softwarequalität zu erreichen. Allgemein betrachtet man dabei sowohl die innere als auch die äußere Softwarequalität.

- Die innere Softwarequalität bezeichnet die Sicht des Entwicklers. Objektorientierte Programmierung bietet durch die Möglichkeiten der Abstraktion, Hierarchiebildung, Kapselung von Interna, Wiederverwendbarkeit, Schnittstellenbildung und einige weitere Techniken hervorragende Ansätze zur Gewährleistung einer hohen inneren Softwarequalität.
- Die äußere Softwarequalität spiegelt die Sicht des Kunden wider. Dieser erwartet Dinge wie Korrektheit, Stabilität, Anwendungsfreundlichkeit oder Erweiterbarkeit einer Software.

Ein Paradigma der objektorientierten Programmierung ist, dass eine hohe innere Softwarequalität zu einer hohen äußeren Softwarequalität führt.

### 11.2.2 Kernkonzepte der Objektorientierung

Die Kernkonzepte in der objektorientierten Softwareentwicklung sind folgende (zum Teil wurden sie schon angedeutet):

- **Objekte** dienen als Abstraktion eines realen Gegenstands oder Konzepts mit einem spezifischen Zustand und einem spezifischen Verhalten. Man nennt sie **Instanzen** einer Klasse.
- Alle Bestandteile, die direkt zu einem Objekt gehören, nennt man **Instanzelemente**. Der Name resultiert daraus, dass wie gesagt „Instanz“ oft als Synonym für „Objekt“ genommen wird. Man bezeichnet diese Bestandteile oft auch als „nichtstatisch“ oder „dynamisch“, da sie für jede Instanz einer Klasse dynamisch erstellt werden.
- **Klassen** dienen als Baupläne für Objekte. In der umgekehrten Sichtweise „klassifizieren“ sie alle Gemeinsamkeiten, die verwandte Objekte auszeichnen. Man nennt sie auch **Objekttypen** oder kurz **Typen**.

- Alle Bestandteile, die nicht direkt zu einem spezifischen Objekt, sondern einer Klasse gehören, nennt man **Klassenelemente** oder auch statische Elemente, wobei man hier in Python aufpassen muss.<sup>3</sup>
- **Attribute** sind eine Beschreibung objekt- und klassenbezogener Daten. Oft wird auch der Bezeichner „**Eigenschaft**“ für „Attribut“ verwendet, wobei auch hier einige Sprachen Besonderheiten aufweisen (Abschn. 11.6.3.2). Ebenso ist „**Feld**“ ein oft genutztes Analogon.
- **Methoden** sind die Beschreibung objekt- und klassenbezogener Funktionalität.
- **Assoziationen** und **Aggregationen/Kompositionen** sind Bezeichnungen von Mechanismen, um Objektbeziehungen auszudrücken.
- **Vererbung** ist eine besondere Form der Assoziation und als Mechanismus zum Generalisieren und Spezialisieren von Objekttypen besonders wichtig in der objektorientierten Programmierung.
- Der Begriff **Polymorphie** ist leider nicht ganz eindeutig und wird in verschiedenen Quellen unterschiedlich streng ausgelegt. Generell bedeutet er Vielgestaltigkeit. Die Polymorphie dient hauptsächlich der flexiblen Auswahl geeigneter Methoden identischen Namens anhand des Objekttyps und der Argumentenliste. Aber der Begriff kann auch weiter gefasst werden und dann weitere Mechanismen zum vielgestaltigen Verhalten umfassen. Auch der Begriff des späten oder dynamischen Bindens wird dafür verwendet. Allgemein bezeichnet man damit eine Auswahl einer Operation, die nicht nur vom Bezeichner der Operation abhängt. Ebenfalls hängt die Auswahl vom Kontext ab, bei Methoden etwa wie gesagt von Anzahl und Typ der Parameter, wenn das eine Sprache berücksichtigt. Oder bei Operatoren von den Typen der Operanden.
- **Schnittstellen** implementieren einen Mechanismus zur Strukturierung von Klassenbeziehungen.
- **Abstrakte Klassen** implementieren einen Mechanismus zum Erzwingen bestimmter Operationen in spezialisierten Klassen. In der Regel kann man damit vorgeben, **dass** etwas zu tun ist, aber noch nicht festlegen, **wie** es zu tun ist.
- **Generische Klassen** dienen der Darstellung von Klassenfamilien.

Auch Sprachen, die explizit als objektorientiert eingestuft werden, implementieren nicht unbedingt alle diese Charakteristika. Wir werden in dem Buch auch nicht sämtliche Techniken sehen, zumal Python nicht alle theoretischen OO-Konzepte bereitstellt.

Gehen wir nun systematisch die Art an, wie Python objektorientierte Programmierung unterstützt. Denn wie schon angedeutet, geht man bei Python an einigen Stellen einen sehr eigenwilligen Weg, der vor allen Dingen Umsteigern aus anderen Sprachen häufig Schwierigkeiten macht.

---

<sup>3</sup>Darauf gehen wir natürlich noch ein.

## 11.3 Klassen

In der objektorientierten Programmierung arbeitet man mit Objekten, entweder mit bereits vom System bereitgestellten Standardobjekten oder selbst erstellten Objekten. Aber wenn nun Objekte die Basis der OOP sind – wie entstehen Objekte und was muss man als Programmierer konkret tun?

### 11.3.1 Klassen als Baupläne, Konstruktoren und Destruktoren

Die Lösung heißt **Klassen** (auch Objekttyp oder abstrakter Datentyp genannt) und darin enthaltene **Konstruktoren**, womit in der Regel aus den Klassen konkrete Objekte (Instanzen) erzeugt werden. Es gibt in einigen Sprachen aber auch die sogenannte **literale Erzeugung** für Instanzen – so auch in Python.

Eine Klasse kann man sich einmal als eine Gruppierung von ähnlichen Objekten vorstellen, die deren Klassifizierung ermöglicht. Eine Klasse ist also die Definition der gemeinsamen Attribute und Methoden sowie der Semantik für eine Menge von gleichartigen Objekten. Alle erzeugten Objekte einer Klasse werden dieser Definition entsprechen.

Die Eigenschaften und Funktionalität der Objekte werden also in der Gruppierung gesammelt und für eine spätere Erzeugung von realen Objekten verwendet. Mit anderen Worten: Klassen sind so etwas wie Baupläne oder Rezepte, um mit deren Anleitung ein konkretes Objekt zu erzeugen. Ein aus einer bestimmten Klasse erzeugtes Objekt nennt man deren Instanz.

Der **Destruktor** zerstört ein Objekt und gibt den Speicherplatz auf. Auf die Details dazu gehen wir in Abschn. 11.5 ein.

### 11.3.2 Der konkrete Klassenaufbau in Python

Zuerst einmal brauchen wir Klassen, um überhaupt Objekte erzeugen zu können. Klassen werden in Python durch das Schlüsselwort *class*, einen Bezeichner, den Doppelpunkt und zwingend eine Anweisung oder einen Anweisungsblock definiert.

Das wäre ein übliches Schema:

---

#### Beispiel

```
class Bezeichner:
 ... Anweisungen ◀
```

Im Inneren der Klasse definiert man die Eigenschaften und Methoden vollkommen analog, wie Sie außerhalb von Klassen Variablen und Funktionen definieren. Hier sehen Sie die Deklaration einer Klasse *Person* mit zwei Eigenschaften und einer Methode:

```
class Person:
 name = "Florian"
 alter = 17
 def reden(self):
 print("Ihr seid alle Individuen!")
```

Die Deklaration der Attribute erfolgt im Grunde erst einmal wie bei „normalen“ Variablen und die Deklaration der Methoden wie bei „normalen“ Funktionen – nur eben im Inneren der Klasse deklariert.

#### 11.3.2.1 Die Objektreferenz `self`

Bei dem letzten Beispiel zur Deklaration einer Klasse soll eine bemerkenswerte Stelle hervorgehoben werden – das ist der Parameter `self`. Dieser muss in Python in jeder Deklaration einer Instanzmethode als erster Parameter notiert werden und verweist auf das aktuelle Objekt. Beim Aufruf wird dieser Parameter dann aber automatisch gefüllt. Dafür darf kein Wert angegeben werden. Das bedeutet, dass beim Aufruf einer Instanzmethode der erste übergebene Wert immer für den zweiten (!) Parameter steht.

In vielen anderen Sprachen kennt man diese `self`-Referenz unter `this`. Nur ist `self` in Python **kein** Schlüsselwort – im Gegensatz zu `this` in vielen anderen Sprachen. Was wiederum bedeutet, dass man im Prinzip hier einen beliebigen Bezeichner notieren kann. In der Praxis wird man jedoch so gut wie immer `self` nehmen.

- ▶ Für eine leere Klasse notieren Sie als einzige Anweisung `pass` – es wird ja zwingend mindestens eine Anweisung benötigt.

#### 11.3.3 Die konkrete Instanziierung

Wie in anderen Programmiersprachen definieren Klassen auch in Python nur einen „Bauplan“, aus dem in der Regel<sup>4</sup> beliebig viele Instanzen erzeugt werden können. Jedes Objekt einer Klasse bildet einen eigenen Namensraum (abgesehen von den statischen Elementen).

- ▶ Ein **Namensraum** ist ein Bereich, in dem ein Bezeichner (etwa der Name einer Methode oder einer Eigenschaft, aber auch der Name einer Datei, wenn man an ein Verzeichnissystem denkt) eindeutig sein muss.

---

<sup>4</sup>Sogenannte Entwurfsmuster können die Anzahl der möglichen Instanzen reglementieren, aber das soll hier außer Acht bleiben.

Attribute und Methoden können von außen zugänglich (*public*) oder nicht zugänglich (*private*) sein (Abschn. 11.5.2).

### 11.3.3.1 Der Default-Konstruktor

Jede (normale) Klasse in Python stellt einen Default-Konstruktor zur Verfügung. Der Aufruf des Konstruktors zur Instanziierung erfolgt in Python einfach, indem der Klassename mit runden Klammern aufgerufen wird. Damit erzeugt dann der Konstruktor ein Objekt und der Konstruktor hat den gleichen Namen wie die Klasse.

Schematisch würde der Aufruf von einem Konstruktor für die Klasse *Person* so aussehen:

#### Beispiel

`Person()` ◀

### 11.3.3.2 Der parametrisierte Konstruktor

Es gibt auch die Möglichkeit, einen parametrisierten Konstruktor zu verwenden. Diesem werden wie einer „normalen“ Funktion beziehungsweise Methode Werte für Parameter übergeben, etwa so:

`Person(42)`

Natürlich muss man diese Werte dann im Konstruktor verwenden, wenn ein parametrisierter Konstruktor einen Sinn haben soll.

#### ► Vorsicht

In den meisten anderen Programmiersprachen verwendet man beim Aufruf des Konstruktors das vorangestellte Schlüsselwort *new*. Das gibt es in Python so nicht. Das ist eine gefährliche Stolperfalle für Umsteiger.

Stattdessen wird die Syntax von Python automatisch durch entsprechende Aufrufe der sogenannten magischen Methoden `__new__` und `__init__` ersetzt (siehe Abschn. 11.4.2).

### 11.3.3.3 Magische Methoden statt parametrisierte Konstrukturen

Es ist in der Regel nicht sinnvoll, dass man ein Objekt ohne Initialisierung erzeugt. Auch ist es kaum sinnvoll, dass es immer mit den gleichen Werten initialisiert wird. In der Regel initialisiert man das Objekt beim Erzeugen bereits mit individuellen, sinnvollen Werten, und das macht man üblicherweise mit einem **parametrisierten** Konstruktor. Nur stellt Python für so etwas nicht direkt eine der vielleicht aus vielen anderen Sprachen gewohnte Notation bereit, in der einfach der aufgerufene Konstruktor neu deklariert wird, sondern man nutzt in Python eine der schon angesprochenen „magischen Methoden“.

- Diese magischen Methoden erkennt man in Python grundsätzlich an zwei vorangestellten und zwei nachgestellten Unterstrichen. Das bedeutet auch, dass „normale“ Methoden und Funktionen so nicht ausgezeichnet werden dürfen.

Beispiele:

```
__new__
__init__
__add__
__sub__
__and__
__str__
__del__
```

#### Hintergrundinformation

Magische Methoden sind in Python einmal die Grundlage für das polymorphe Verhalten von Operatoren. Denn diese Operatoren sind zum Teil überladen. Man kann beispielsweise das Plus-Zeichen sowohl für die Addition von verschiedenen numerischen Werten als auch für die String-Verkettung benutzen. Der Token ist zwar gleich, bedeutet aber je nach Kontext (Art der Parameter) etwas ganz anderes – das nennt man Überladen.

Die Möglichkeit des Überladens eines Operators in Python erlaubt es auch, dass man das mit eigenen Klassen selbst machen kann. Für jedes Operatorzeichen in Python gibt es eine spezielle Methode.

Für „+“ lautet der Name der Methode beispielsweise `__add__`, und für „–“ lautet der Name `__sub__`.

Das Thema werden wir in den Unterlagen aber nicht weiter vertiefen.

Beim Erzeugen von Instanzen wird zuerst von Python eine Methode `__new__` im Hintergrund aufgerufen, die man als Programmierer nicht weiter modifiziert. Aber die Initialisierung erfolgt mit der `__init__`-Methode, die danach automatisch aufgerufen wird und die man redefinieren kann.

Beachten Sie, dass diese Redefinition das vorangestellte Schlüsselwort `def` notwendig macht. Genau genommen wird diese magische Methode damit im Sinne der objektorientierten Programmierung überschrieben, wie dies oben erläutert wurde.

#### 11.3.3.4 Das Top-Level-Script-Environment

Das Top-Level-Script-Environment (auch als globales Umgebungs- oder Modul-Umgebung bezeichnet) in Python bezieht sich auf den höchsten Ausführungskontext, in dem ein Python-Skript ausgeführt wird. Dieses Umfeld besteht aus den folgenden Komponenten:

**Globaler Gültigkeitsbereich** (Global Scope): Hier werden Variablen, Funktionen und Klassen definiert, die im gesamten Skript oder Modul verfügbar sind. Alles, was auf dieser Ebene definiert ist, wird *global* genannt und es kann von überall im Skript aus zugegriffen werden (*globalscope.py*).

**Beispiel**

```
global_variable = 42
def global_function():
 return "global scope"
class GlobalClass:
 pass ◀
```

In diesem Beispiel sind *global\_variable*, *global\_function* und *GlobalClass* im globalen Gültigkeitsbereich definiert und können von überall im Skript aus verwendet werden.

**Hauptprogramm** (Main Program): Das Hauptprogramm in Python ist der Teil des Codes, der direkt im globalen Gültigkeitsbereich ausgeführt wird. Normalerweise ist dies der Code, der außerhalb von Funktionen und Klassen steht und den Einstiegspunkt für das Skript bildet. Aber es gibt auch spezielle Syntaxkonstruktionen, um dieses Hauptprogramm anzugeben.

In Python ist *\_\_main\_\_* ein spezielles Attribut bzw. eine magische Methode, die als ein Konzept verwendet wird, um den Einstiegspunkt eines Python-Skripts oder Moduls zu kennzeichnen. Das *\_\_main\_\_*-Attribut hat zwei Hauptverwendungszwecke:

Es ist einmal der Einstiegspunkt für das Hauptprogramm. Wenn ein Python-Skript direkt ausgeführt wird (nicht importiert), wird das Skript als das Hauptprogramm betrachtet. Der Python-Interpreter setzt dann das spezielle Attribut *\_\_name\_\_* auf den Wert *\_\_main\_\_* für das Hauptprogramm. Das kann man im Code ausnutzen (*hauptprogramm.py*).

**Beispiel**

```
if __name__ == "__main__":
 # Dieser Code wird nur ausgeführt, wenn das Skript als Hauptprogramm läuft
 print("main program") ◀
```

Der obige Code stellt sicher, dass der darin enthaltene Code nur ausgeführt wird, wenn das Skript direkt ausgeführt wird. Wenn das Skript jedoch von einem anderen Skript oder Modul importiert wird, wird der Code in der Bedingung nicht ausgeführt.

Was zur zweiten Situation führt – der Modul-Verwendung. Wenn ein Python-Skript als Modul in ein anderes Skript importiert wird, wird das *\_\_name\_\_*-Attribut auf den Namen des Moduls (dem Dateinamen ohne Erweiterung) gesetzt. Darauf kommen wir bei Modulen zurück.

In der **Interaktive Shell** ist die interaktive Shell selbst das Top-Level-Script-Environment. Sie können in der Shell direkt Python-Ausdrücke und Anweisungen eingeben und sehen die sofortigen Ergebnisse.

Das Top-Level-Script-Environment ist der äußerste Kontext, in dem Ihr Python-Code ausgeführt wird. Alle Variablen und Funktionen, die in diesem Bereich definiert sind,

sind im gesamten Skript oder Modul verfügbar. Es ist wichtig zu beachten, dass Python-Dateien (Module) ein eigenes Top-Level-Script-Environment haben, was bedeutet, dass sie getrennt voneinander arbeiten und ihre eigenen globalen Gültigkeitsbereiche haben.

Wenn Sie Python-Dateien (Module) erstellen, können Sie den globalen Gültigkeitsbereich nutzen, um Variablen, Funktionen und Klassen zu definieren, die von anderen Teilen des Skripts oder von anderen Modulen aus wiederverwendet werden können. Dies erleichtert die Organisation und Wiederverwendbarkeit von Code in Python-Anwendungen.

---

## 11.4 Details zu Objekten

In der strengen objektorientierten Programmierung wird alles, was Sie jemals im Quelltext notieren, als Objekt beziehungsweise Klasse oder als ein Objekt-/Klassenbestandteil zu verstehen sein. Das gilt nicht zwingend für sogenannte hybride Sprachen, mit denen man zwar objektorientiert programmieren kann, aber auch nach anderen Paradigma. Aber versuchen wir jetzt genauer zu klären, was Objekte sind.

**Objekte** sind im objektorientierten Denkansatz alles, was sich eigenständig erfassen und ansprechen lässt. Sie besitzen

- Eigenschaften,
- Methoden und
- einen Zustand.

Ein Objekt ist eine konkret vorhandene, agierende Einheit mit Identität (die relevanten Eigenschaften) und definierten Grenzen, die den Zustand und das Verhalten des Objektes kapseln. Ein Objekt ist durch seine Identität, seinen Zustand und sein Verhalten vollständig charakterisiert und die Eigenschaften und Methoden sind immer an ein spezifisches Objekt gekoppelt. In der objektorientierten Programmierung sind Methoden und Eigenschaften immer nur über ein spezifisches Objekt zugänglich.

Es gibt in der streng objektorientierten Programmierung grundsätzlich keine „freien“ Funktionalitäten und Variablen. Ein Objekt ist nach außen nur durch seine offengelegten Methoden und Attribute definiert. Es ist gekapselt, versteckt seine innere Struktur – bis auf die offengelegten Elemente – vollständig vor anderen Objekten. Man nennt dies **Information Hiding**, **Datenkapselung** oder einfach **Kapselung**. Kapselung ermöglicht einen differenzierten Zugriffsschutz und kann sowohl für eine Klasse als auch jedes Klassenelement separat festgelegt werden. Damit kann die sogenannte **Sichtbarkeit** geregelt werden.

Die Verwendung von Objekten erfolgt in der objektorientierten Programmierung über einen Namen für das Objekt oder zumindest einen Stellvertreterbegriff, der ein Objekt repräsentiert (der Identifikator). Meist<sup>5</sup> wird die **Punktnotation** für den Zugriff verwendet – so auch in Python.

### 11.4.1 OO-Philosophie als Abstraktion

Objektorientierte Denkweise ist jetzt ganz und gar nicht künstlich. Ganz im Gegenteil. Die gesamte objektorientierte Philosophie entspricht vielmehr der realen Natur als der prozedurale Denkansatz, der von der Struktur des Computers definiert wird. Ein Objekt ist im Sinne der objektorientierten Philosophie eine Abstraktion eines in sich geschlossenen Elements der realen Welt. Dabei spricht man von Abstraktion, weil zur Lösung eines Problems normalerweise weder sämtliche Aspekte eines realen Elements benötigt werden noch überhaupt darstellbar sind. Ein Mensch bedient sich eines Objekts, um eine Aufgabe zu erledigen. Man weiß in der Regel sogar nicht genau, wie das Objekt im Inneren funktioniert, aber man kann es bedienen (weiß also um die im Moment interessanten Methoden, um es verwenden zu können) und weiß um die derzeit interessanten Eigenschaften des Objekts (seine Attribute) und wann was zur Verfügung steht (sein Zustand).

Aus programmiertechnischer Sicht ist es so, dass nicht nur der Programmierer weiß, was ein Objekt leistet. Auch das Objekt weiß es selbst und kann es nach außen dokumentieren. Es ist sich quasi seiner Existenz bewusst. Das wird in geeigneten Entwicklungsumgebungen genutzt, indem Ihnen ein Editor bereits Hilfe anbietet, indem er Ihnen bei einem Objekt alles anzeigt, was das Objekt leisten kann. Was da dann nicht auftaucht, kann auch nicht von einem Objekt gefordert werden.

### 11.4.2 Instanzelemente versus Klassenelemente

Wie in anderen Programmiersprachen wird auch in Python zwischen **Instanzeigenschaften** und **Klasseneigenschaften** (auch Instanzattribute beziehungsweise Klassenattribute oder statische Attribute genannt) unterschieden, wobei letztere eher in Ausnahmefällen verwendet werden. Diese gleiche Differenzierung nimmt man bei Methoden vor. Aber es ist notwendig, dass man explizit zwischen Eigenschaften und Methoden unterscheidet. Denn während sich Python bei Eigenschaften an die sonst meist übliche OO-Philosophie hält,<sup>6</sup> geht man bei Methoden einen eigenen Weg.

---

<sup>5</sup>Eine andere Notation finden Sie zum Beispiel in PHP.

<sup>6</sup>Wennleich auch hier mit einer speziellen Syntax.

### 11.4.2.1 Instanzattribute versus Klassenattribute in Python

- Klassenattribute sind in Python weitgehend „normale“ Variablen. Nur wird deren Deklaration in Python **direkt im class-Block** unter Zuweisung eines Initialwerts vorgenommen (das heißt nicht innerhalb des Konstruktors oder einer anderen Methode).
- Instanzattribute hingegen werden in Python **im Konstruktor** definiert und mit Initialwerten versehen.

Es gibt in Python jetzt aber einige Besonderheiten, die wir ausarbeiten müssen.

### 11.4.3 Der Aufbau von Objekten in Python

Wenn ein Objekt erzeugt wurde, werden in Python die Attribute wie auch Methoden eines Objektes in einem Dictionary `__dict__` gespeichert, also jeweils über ein Wertepaar. Insbesondere sollte man beachten, dass Methoden damit über **Referenzen** bereitgestellt werden. Wir werden diese Tatsache in Abschn. 11.8.2 genauer untersuchen und damit einige interessante Dinge anstellen, die aber im Moment zu früh kommen.

### 11.4.4 Zugriff auf Objektbestandteile

Die Eigenschaften und Methoden eines Objekts werden dann ganz klassisch über die Punktnotation angesprochen. Der schon angesprochene Parameter `self` dient in Python auch dazu, dass man in Methoden auf die anderen Instanzelemente zugreifen kann. Man notiert in der Punktnotation einfach `self` und dann die gewünschte Eigenschaft oder Methode. Der Parameter `self` steht also für ein anonymes Objekt.

#### 11.4.4.1 Alles ist ein Objekt

Bereits in der Einleitung wurde angedeutet, dass in Python im Grunde alles ein Objekt ist. Auch wenn man beispielsweise nach dem funktionalen Paradigma programmiert und dort etwa primitive Datentypen einsetzt, arbeitet Python im Hintergrund mit Objekten. Ein kleines Beispiel soll das zeigen (*Objekt1.py*):

```
x = 42.0
print(x.is_integer())
```

Das Ergebnis wird *True* sein, aber wichtiger ist in unserem Zusammenhang, dass wir in der ersten Zeile des Quellcodes literal **ein Objekt** erzeugt haben, obwohl wir mit

`x = 42.0` vordergründig eine Variable mit einem **primitiven Datentyp** anlegen. Der Zugriff über die Punktnotation auf eine Methode beweist jedoch, dass Python auch hier im Hintergrund ein Objekt erzeugt hat.

- ▶ Wichtig ist, dass man sich in Python sehr oft eben gar keine Gedanken darum machen muss, ob hinter den Kulissen ein Objekt verwendet wird oder wie Python überhaupt im Hintergrund arbeitet. Das ist ja explizit eines der Ziele von Python.

#### 11.4.4.2 Ein Beispiel für einen parametrisierten Konstruktor

Schauen wir uns jetzt die Notation einer Klasse mit einem parametrisierten Konstruktor zur Initialisierung und die Instanziierung in einem Beispiel an (*Klasse1.py*):

```
class Tier:
 def __init__(self, name, alter, typ):
 self.typ = typ
 self.name = name
 self.alter = alter
 def lautgeben(self):
 print("Mein Name ist ", self.name, ", bin ein ", self.typ,
 " und ich bin ", self.alter, " Jahre alt.", sep="")
 print("Miau")
obj = Tier("Herby", 5, "Kater")
print(obj.name, obj.alter, obj.typ)
obj.lautgeben()
```

Sie sehen, dass die magische Methode `__init__` mit vorangestelltem `def` überschrieben wird und wir damit einen parametrisierten Konstruktor anlegen. Durch das vorangestellte `self` werden damit im Inneren auch gleich die Instanzvariablen `typ`, `name` und `alter` angelegt und mit den übergebenen Werten initialisiert.

Weiter finden Sie die Deklaration einer Methode `lautgeben()`.

Nach der Erzeugung eines Objekts wird ganz normal über die Punktnotation auf die Eigenschaften und die Methode zugegriffen. In Abb. 11.1 sehen Sie die Ausgabe des Beispiels.

```
Herby 5 Kater
Mein Name ist Herby, bin ein Kater und ich bin 5 Jahre alt.
Miau
>>>
```

**Abb. 11.1** Ausgabe der Methode und die Werte der Eigenschaften

### 11.4.4.3 Beispiele mit Klassenattributen

Werfen wir einen Blick auf eine kleine Abwandlung des letzten Beispiels (*Klasse2.py*):

```
class Tier:
 eigentuemer = "Papa"
 def __init__(self, name, alter, typ):
 self.typ = typ
 self.name = name
 self.alter = alter
 def lautgeben(self):
 print("Mein Name ist ", self.name, ", bin ein ", self.typ,
 " und ich bin ", self.alter, " Jahre alt.", sep="")
 print("Miau")
obj = Tier("Herby", 5, "Kater")
print(obj.name, obj.alter, obj.typ)
obj.lautgeben()
print("Zugriff über das Objekt:", obj.eigentuemer)
print("Zugriff über die Klasse:", Tier.eigentuemer)
Tier.eigentuemer = "Felix"
print("Verkauft an", obj.eigentuemer)
```

Sie sehen in dem Beispiel, dass Attribute der Klasse **außerhalb** der magischen Methode deklariert und dort neu mit Werten belegt werden können. Das sind in Python dann wie erwähnt **Klasseneigenschaften** oder statische Attribute.

Es ist in der Regel der Sinn der Verwendung von Klassenelementen, dass man sich einmal die Instanziierung spart. Aber noch wichtiger ist die Tatsache, dass Informationen über Klassenelemente in allen Instanzen verfügbar sind und sich auch Änderungen in allen Instanzen auswirken.

Doch was passiert bei Änderungen des Wertes der Klasseneigenschaft?

In dem konkreten Listing gibt es eine Klasseneigenschaft *eigentuemer*. Das Besondere an Python ist, dass der folgende Zugriff auf diese Klasseneigenschaft sowohl über eine Instanz (*obj.eigentuemer*) als auch über die Klasse (*Tier.eigentuemer*) erfolgen kann, und zwar einerseits lesend, was im Prinzip unkritisch ist.

Aber es geht andererseits auch schreibend, um den Wert der Klasseneigenschaft zu ändern (Abb. 11.2). Diese Möglichkeit, auf Klassenelemente über die Instanz zuzugreifen, kennt man zwar beispielsweise auch aus Java,<sup>7</sup> aber das eng verwandte C# unterbindet diese Art des Zugriffs kategorisch. Und im Grunde ist eine Warnung beziehungsweise das grundsätzliche Unterbinden äußerst sinnvoll, denn der Zugriff über

---

<sup>7</sup> Wenngleich es da explizit nicht empfohlen wird und der Compiler eine Warnung anzeigt.

**Abb. 11.2** Klassenelemente

```

Herby 5 Kater
Mein Name ist Herby, bin ein Kater und ich bin 5 Jahre alt.
Miau
Zugriff über das Objekt: Papa
Zugriff über die Klasse: Papa
Verkauft an Felix
>>>

```

ein Objekt ist im Grunde unlogisch und auch gefährlich. Gerade Python verhält sich aus meiner Sicht in dem Kontext etwas tückisch.

Die Ausgabe des Beispiels *Klasse2.py*, macht das tückische Verhalten noch nicht wirklich deutlich. Denn mit *Tier.eigentuemer = "Felix"* wird die Klasseneigenschaft über die Klasse geändert, und der folgende Zugriff über die Instanz (*obj.eigentuemer*) zeigt dann den geänderten Wert (Abb. 11.2).

Aber auch das folgende Beispiel *Klasse3.py* ist gültiger Python-Code:

```

class Tier:
 eigentuemer = "Papa"
 def __init__(self, name, alter, typ):
 self.typ = typ
 self.name = name
 self.alter = alter
obj1 = Tier("Herby", 5, "Kater")
obj2 = Tier("Hägar", 2, "Kanarienvogel")
obj3 = Tier("Helga", 4, "Kanarienvogel")

print(obj1.name, "gehört", obj1.eigentuemer)
print(obj2.name, "gehört", obj2.eigentuemer)
print(obj3.name, "gehört", obj3.eigentuemer)
print("Alle Tiere gehören", Tier.eigentuemer, "\n")

obj1.eigentuemer = "Florian"

print(obj1.name, "gehört jetzt", obj1.eigentuemer)
print(obj2.name, "gehört jetzt", obj2.eigentuemer)
print(obj3.name, "gehört jetzt", obj3.eigentuemer)
print("Alle Tiere gehören jetzt", Tier.eigentuemer, "\n")

Tier.eigentuemer = "Felix"
print(obj1.name, "gehört jetzt", obj1.eigentuemer)
print(obj2.name, "gehört jetzt", obj2.eigentuemer)

```

```
print(obj3.name, "gehört jetzt", obj3.eigentuemer)
print("Alle Tiere gehören jetzt", Tier.eigentuemer)
```

Das Ergebnis muss genauer betrachtet werden. Die ersten Ausgaben sollten noch klar sein. Es gibt eine Klassenvariable, und deren Wert wird in den ersten vier Ausgaben verwendet. Der Zugriff erfolgt über die drei Instanzen und die Klasse. Das ist ja wie gesagt in Python nicht verboten, deshalb bekommt man diese Ausgabe:

*Herby gehört Papa  
Hägar gehört Papa  
Helga gehört Papa  
Alle Tiere gehören Papa*

Nun wird mit der folgenden Anweisung *obj1.eigentuemer = "Florian"* versucht, den Wert der Klasseneigenschaft über die Instanz zu ändern. Denn man kann ja auf die Klasseneigenschaft lesend zugreifen, wie wir oben gesehen haben. Und damit sollte man auch schreibend darauf zugreifen können, wie es etwa in Java der Fall wäre. Python wird auch bei so einem Versuch keinen Fehler oder Problem melden. Aber das ist die nachfolgende Ausgabe:

*Herby gehört jetzt Florian  
Hägar gehört jetzt Papa  
Helga gehört jetzt Papa  
Alle Tiere gehören jetzt Papa*

Der schreibende Zugriff hat den Wert der Klasseneigenschaft **nicht** (!) geändert, sondern verdeckt nur deren Wert in der konkreten Instanz *obj1*. Dort gibt es somit eine implizite Instanzeigenschaft.

Und die Sache wird noch heimtückischer, denn eine nachfolgende Änderung der Klasseneigenschaft über die Klasse mit *Tier.eigentuemer = "Felix"* führt zu der Ausgabe:

*Herby gehört jetzt Florian  
Hägar gehört jetzt Felix  
Helga gehört jetzt Felix  
Alle Tiere gehören jetzt Felix*

Da die Klasseneigenschaft durch die vorherige Anweisung *obj1.eigentuemer = "Florian"* in *obj1* verdeckt ist, wird sich auch die nachfolgende Änderung der Klasseneigenschaft über die Klasse bei dem folgenden Zugriff über die Instanz nicht auswirken.

- ▶ Trotz der gezeigten Möglichkeit, auf Klassenelemente über eine Instanz zuzugreifen, sollte man auch in Python meines Erachtens immer den Zugriff über die Klasse wählen. Das ist dann schon vom Quellcode her viel besser lesbar,

wartbar und klarer. Aber insbesondere das letzte Beispiel hat gezeigt, dass man auch mit recht tückischem Verhalten rechnen muss, wenn man schreibend über die Instanz auf die Klasseneigenschaft zugreift (auch wenn man in Python das Verhalten natürlich auch bewusst und sinnvoll einsetzen kann).

- **Tipp** Wenn Sie aus einer Instanzmethode oder dem Konstruktor auf eine Klasseneigenschaft zugreifen wollen, können Sie den Klassennamen voranstellen. Allerdings kann man auch die Instanz verwenden und über das Built-in `type()` auf die Klassen zugreifen. Das hat den wesentlichen Vorteil, dass man den Namen der Klasse nicht explizit angeben muss und damit der Code universeller ist. Das geht dann so:

```
type(self)
```

### 11.4.5 Von Grund auf objektorientiert

Nicht zuletzt durch das funktionale Paradigma werden die meisten Python-Applikationen aus einer sequenziellen Abarbeitung von Anweisungen in einem .py-Dokument bestehen. Sie finden darin Anweisungen und Deklarationen, aber es existiert damit immer erst einmal eine funktionale Basis – auch wenn Sie objektorientiert arbeiten wollen.

Aber man kann in Python auch ein wirklich objektorientiertes Fundament als Basis einer Anwendung nehmen. Dazu deklariert man einfach eine Klasse und lässt den gesamten folgenden Code aus dem Konstruktor des Klasse „entspringen“. Man muss dann nur noch die Klasse instanziieren. Sofern der restliche Code dann konsequent mit dem objektorientierten Ansatz arbeitet, ist das dann eine wirklich objektorientierte Python-Applikation. Hier sehen Sie die Schablone für so eine Minimalstruktur, bei der die Programmklasse *Application* lauten soll:

---

#### Beispiel

```
class Application():
 def __init__(self):
 ...
 Application() ◀
```

Man programmiert in Python eigentlich selten nach diesem Ansatz. Eine Ausnahme ist die Erstellung von grafischen Oberflächen. Da wird man oft so ein Design finden. Man hat einen Konstruktor mit einem Frame und kann da andere GUI-Elemente wie Label, Button oder Eingabefelder erstellen.

## 11.5 Klassenmethoden und statische Methoden

Instanzmethoden gehören zur konkreten Instanz. So weit haben wir das schon besprochen. Aber es gibt analog der Situation bei Eigenschaften auch Methoden, die nicht zur konkreten Instanz gehören, sondern zu der Klasse. In der objektorientierten Programmierung nennt man solche **Klassenmethoden** normalerweise **statische Methoden**, und auch die meisten Programmiersprachen unterscheiden hier nicht. Nicht so Python! In Python gibt es Unterschiede zwischen Klassenmethoden und statischen Methoden, das sorgt bei Umsteigern meist für Verwirrung.

### 11.5.1 Klassenmethoden

Klassenmethoden sind in Python offensichtlich nicht an Instanzen gebunden, aber anders als statische Methoden (Abschn. 11.5.2) sind Klassenmethoden an eine Klasse gebunden (was zu erwarten ist). Sie gehören also explizit zur Klasse, und das erste Argument einer Klassenmethode ist in Python immer eine Referenz auf die Klasse. Diese Referenz wird in Python das **Klassenobjekt** genannt. Aufrufen kann man diese Methode über den vorangestellten Klassennamen oder eine Instanz.<sup>8</sup>

Das bedeutet, dass beim Aufruf einer Klassenmethode der erste übergebene Wert immer über den zweiten (!) Parameter der Deklaration bereitsteht. Das Konzept entspricht damit der Logik beim Aufruf von Instanzmethoden. Auch hier gilt, dass der Name des ersten Parameters **kein** Schlüsselwort ist und frei gewählt werden kann. Üblich ist aber *cls*.

- ▶ Klassenmethoden werden in der Regel mit der Annotation<sup>9</sup> `@classmethod` gekennzeichnet.

Betrachten wir das Beispiel *Klasse4.py*:

```
class Tier:
 anzahl = 0
 def __init__(self):
 Tier.anzahl += 1

 @classmethod
 def AnzahlTiere(cls):
 return cls.anzahl
```

---

<sup>8</sup>Also die analoge Situation wie bei Klasseneigenschaften.

<sup>9</sup>Annotation bedeutet „Anmerkung“, „Beifügung“ oder „Hinzufügung“ und wird in Python als Kommentar verstanden, den der Interpreter ignoriert.

```
print("Anzahl der Tiere: ", Tier.AnzahlTiere())
obj1 = Tier()
print("Anzahl der Tiere: ", Tier.AnzahlTiere())
obj2 = Tier()
obj3 = Tier()
print("Anzahl der Tiere (Abfrage für die Klasse): ", Tier.AnzahlTiere())
print("Anzahl der Tiere (Abfrage für eine Instanz): ", obj2.Anzahl-
Tiere())
```

Die Ausgabe wird sein:

```
Anzahl der Tiere: 0
Anzahl der Tiere: 1
Anzahl der Tiere (Abfrage für die Klasse): 3
Anzahl der Tiere (Abfrage für eine Instanz): 3
```

Es gibt in dem Beispiel eine Klasseneigenschaft *anzahl*, mit der die Anzahl der erzeugten Instanzen mitgezählt werden soll. Im Konstruktor wird der Wert mit *anzahl.Tier* bei jeder Erzeugung einer Instanz um den Wert 1 erhöht.

Die Klassenmethode *AnzahlTiere()* liefert diese Anzahl als Rückgabewert. Über den ersten (und zwingenden) Parameter kann man dabei auf die Klasse und damit die Klasseneigenschaft zugreifen. Wie Sie sehen, können Sie auch hier sowohl über die Klasse als auch die Instanz auf die Klassenmethode zugreifen.

### 11.5.2 Statische Methoden

Python differenziert wie gesagt zwischen Klassenmethoden und statischen Methoden. Statische Methoden in Python sind weder an Instanzen noch an Klassen gebunden. Statische Methoden „wissen“ nichts von der Klasse, in der sie definiert wurden. Es sind einfach Methoden, die in der Klasse definiert sind und deren erster Parameter **nicht** für die Klasse oder eine Instanz steht.

Statische Methoden können von außen oder in anderen statischen Methoden über den vorangestellten Klassennamen, in Klassenmethoden über den ersten Parameter oder den Klassennamen aufgerufen werden.

- ▶ Statische Methoden werden in Python in der Regel mit der Annotation `@staticmethod` gekennzeichnet.

Betrachten wir das Beispiel *Klasse5.py*:

```
class Rechne:
 @staticmethod
```

```
def addiere(a,b):
 return a + b
@staticmethod
def multipliziere(a,b):
 return a * b
@staticmethod
def dividiere(a,b):
 return a / b
@staticmethod
def punktrechnung():
 print(Rechne.multipliziere(21,2))
 print(Rechne.dividiere(21,2))
 obj = Rechne()

 print(obj.addiere(5, 6))
@classmethod
def rechne(cls):
 print(cls.addiere(22,20))
 print(Rechne.multipliziere(22,20))
 cls.punktrechnung()
 obj = Rechne()
 print(obj.addiere(15, 6))
Rechne.multipliziere(22,20)
obj = Rechne()
print(obj.addiere(33,30))
obj.rechne()
```

Das ist die Ausgabe des Beispiels:

63

42

440

42

10.5

11

21

Sie finden in dem Beispiel die Deklaration von vier statischen Methoden. Die ersten drei statischen Methoden berechnen einfach aus den beiden übergebenen Parametern ein Ergebnis und liefern dieses zurück.

Die vierte statische Methode ist interessant. Denn in dieser werden die drei anderen statischen Methoden aufgerufen, einfach über den vorangestellten Klassennamen oder auch eine Instanz der Klasse. Allerdings müssen Sie diesen Identifier auch notieren, denn eine statische Methode „weiß“ wie gesagt rein gar nichts über die Klasse, in der sie deklariert ist. Damit unterscheidet sich der Zugriff aus der statischen Methode nicht von dem Zugriff von außen.

In der Klassenmethode hingegen können Sie entweder den ersten Parameter oder aber den Klassennamen oder auch eine Instanz der Klasse verwenden.

---

## 11.6 Eine Frage der Sichtbarkeit

Die **Sichtbarkeit** von Methoden und Attributen außerhalb einer Klasse kann in der objektorientierten Programmierung in der Regel vom Programmierer vorgegeben werden. So auch in Python. Python unterstützt folgende drei Sichtbarkeitsstufen:

- Attribute und Methoden mit der Sichtbarkeit **public** sind von außerhalb uneingeschränkt und direkt über eine Referenz auf das Objekt der Klasse zugänglich. Ohne spezielle Kennzeichnung (also Voreinstellung) sind in Python alle Attribute und Methoden public und damit von überall aus sichtbar.
- Attribute und Methoden mit der Sichtbarkeit **private** sind von außen nicht sichtbar. Attribute und Methoden werden in Python als private deklariert, indem man ihren Namen mit dem Prefix `_` (doppelter Unterstrich) versieht.
- Als **protected** erklärte Attribute und Methoden werden gekennzeichnet, indem man dem Namen einen einfachen Unterstrich `_` voranstellt. Technisch ist für diese Elemente in Python der Zugriff von außen genauso uneingeschränkt möglich wie im Fall *public*. Das unterscheidet diese Kennzeichnung in Python von der Verwendung in den meisten anderen OO-Sprachen, wo hiermit eine Beschränkung der Zugriffsmöglichkeiten auf Objekte festgelegt wird, die in einer Vererbungsbeziehung stehen! Es handelt sich bei diesem Sichtbarkeitslevel in Python nur um eine Konvention oder Empfehlung.

### 11.6.1 Ein Beispiel für den Zugriff auf ein öffentliches Element

Zur Verdeutlichung der Sichtbarkeiten betrachten wir das Beispiel *Klasse6.py*.

```
class Tier:
 anzahl = 0
 def __init__(self):
```

```

 Tier.anzahl += 1
 @classmethod
 def AnzahlTiere(cls):
 return cls.anzahl
print("Anzahl der Tiere: ", Tier.AnzahlTiere())
obj1 = Tier()
print("Anzahl der Tiere: ", Tier.AnzahlTiere())
obj2 = Tier()
print("Anzahl der Tiere: ", Tier.anzahl)

```

Es gibt in dem Beispiel eine Klasseneigenschaft *anzahl* ohne besondere Kennzeichnung. Damit ist sie in Python public und kann sowohl im Konstruktor und der Klassenmethode im Inneren der Klasse, aber auch von außen über die Klasse verwendet werden.

### 11.6.2 Ein Beispiel für den versuchten Zugriff auf ein privates Element von außen

Machen wir die Klasseneigenschaft nun private. Dazu notieren wir einfach bei der Deklaration zwei Unterstriche vor den Bezeichner (*Klasse7.py*). Natürlich muss dann der Bezeichner auch bei jedem Zugriff angepasst werden.

```

class Tier:
 __anzahl = 0
 def __init__(self):
 Tier.__anzahl += 1
 @classmethod
 def AnzahlTiere(cls):
 return cls.__anzahl
print("Anzahl der Tiere: ", Tier.AnzahlTiere())
obj1 = Tier()
print("Anzahl der Tiere: ", Tier.AnzahlTiere())
obj2 = Tier()
print("Anzahl der Tiere: ", Tier.__anzahl)

```

Auf die private Klasseneigenschaft kann man weiterhin von innerhalb der Klasse über den Konstruktor und die Klassenmethode zugriffen, aber der Zugriff von außerhalb der Klasse ist nicht mehr möglich (Abb. 11.3). Sie erhalten eine entsprechende Fehlermeldung.

Beachten Sie, dass es in Python trotzdem Techniken gibt, diesen privaten Level zu umgehen. Dazu verwenden Sie die magischen Methoden, was wir aber in den Unterrlagen nicht vertiefen. Zumal das Aushebeln von wichtigen und sinnvollen Mechanismen – bis auf Ausnahmefälle – selten eine gute Idee ist.

**Abb. 11.3** Auf die private Eigenschaft kann man von außen nicht zugreifen

```
Anzahl der Tiere: 0
Anzahl der Tiere: 1
Traceback (most recent call last):
 File "F:/PythonQuellcodes/Kap11/Klasse7.py", line 16, in <module>
 print("Anzahl der Tiere: ", Tier.__anzahl)
AttributeError: type object 'Tier' has no attribute '__anzahl'
>>> |
```

### 11.6.3 Getter und Setter

Wenn man private Eigenschaften in einer Klasse anlegt, sind diese damit von außen nicht direkt sichtbar. Aber oft möchte man dennoch darauf zugreifen – nur indirekt über spezielle Methoden. Es gibt verschiedene Gründe, warum viele Details in einem Objekt oft nicht direkt offengelegt werden und stattdessen einen solchen indirekten Zugang über Methoden bereitstellen. Denn das Konzept hat vielfältige Vorteile. Insbesondere kann man damit Werte filtern. Wenn Sie etwa ein Objekt haben, das eine Woche repräsentieren soll, ergibt es keinen Sinn, wenn Sie eine Eigenschaft für einen Wochentag auf den Wert 8 setzen würden. Bei einem direkten Zugriff auf ein Feld wäre so ein Filter nur mühsam oder gar nicht zu realisieren, denn die verfügbaren Datentypen lassen nur eine recht grobe Spezifikation der Inhalte zu, die in der Regel nicht genügt (etwa der Wertebereich eines Datentyps).

Wenn Sie jedoch über eine Methode zum Setzen der Tage auf das Feld zugreifen, kann im Inneren dieser Methode eine Überprüfung auf erlaubte Werte greifen. Das gilt für all die Fälle, in denen nur bestimmte Werte sinnvoll sind. Aber auch bei der Abfrage des Wertes von einem Feld kann so ein indirekter Zugriff sehr sinnvoll sein. Es kann sein, dass – je nach Situation – nur bestimmte Informationen Sinn ergeben. Denken Sie bei unserem Beispiel-API mit den Tieren an die Anzahl der Eier, die ein Tier vom Typ *Geflügel* liefern kann – die hängt vom Geschlecht ab. Ebenso können Sie durch die Unterbindung eines direkten Zugriffs auf ein Feld einrichten, dass ein Feld entweder nur zum Lesen (read-only – **Lesezugriff**) oder zum Setzen (write-only – **Schreibzugriff**) eines Wertes freigegeben ist oder auch beides. Je nach gewünschter Situation stellen Sie die entsprechenden Methoden bereit oder auch nicht. Ein indirekter Zugriff gestattet auch Änderungen der Eigenschaften selbst, sowohl was den Namen als auch den Typ der repräsentierenden Variable angeht. Die Filterfunktion sorgt gegebenenfalls dafür, dass die Schnittstelle nach außen unverändert bleibt.

#### 11.6.3.1 Die tatsächliche Notation von Getter und Setter

Grundsätzlich werden Sie diese Vorgehensweise in vielen objektorientierten APIs sehr oft realisiert vorfinden. Die Methoden, die zum Setzen des Werts einer Eigenschaft verwendet werden, beginnen dort sehr oft mit *set* und die Methoden zum Abfragen des Wertes einer Eigenschaft mit *get* oder *is* (Letzteres beim Zugriff auf boolesche Werte – Wahrheitswerte). Beispiele sind etwa *setText("Text")* zum Setzen der Beschriftung einer

Schaltfläche oder `getText()` zum Abfragen des Wertes. Man nennt solche Methoden Getter-Methoden und Setter-Methoden oder kurz nur **Getter** und **Setter**. Die Eigenschaft selbst wird bei so einem Zugriffskonzept privatisiert, um einen direkten Zugriff zu verhindern. Das ist zwar syntaktisch nicht zwingend, aber vom Design her unbedingt zu empfehlen – sonst braucht man auch keine Getter und Setter.

- Ein indirekter Zugriff auf Eigenschaften hat sich über die Jahre als Königsweg des Designs einer objektorientierten Anwendung etabliert.

Das ist ein Beispiel mit der Anwendung von Gettern und Settern (*GetterSetter.py*):

```
class Katze:
 def __init__(self, alter):
 self.__alter = alter
 def getAlter(self):
 return self.__alter
 def setAlter(self, alter):
 self.__alter = alter
obj = Katze(1)
print(obj.getAlter())
obj.setAlter(5);
print(obj.getAlter())
```

Es sollten Ihnen der identische Parametername bei dem Setter und dem Bezeichner des Attributs auffallen (bis auf die Unterstriche zur Privatisierung des Attributs). Für Anfänger ist die identische Wahl des Parameternamens des Setters und des Bezeichners des Feldes am Anfang oft verwirrend. Aber es hat eine einfache und extrem konsistente Logik, diese Bezeichner gleich zu wählen. Denn damit ist die Zuordnung schon von den Bezeichnern her klar.

### 11.6.3.2 Properties als Ersatz für Getter und Setter

In einigen Programmierkonzepten, wie etwa dem .NET-Framework allgemein, versteht man den Begriff einer „Eigenschaft“ beziehungsweise auf Englisch „property“ so, dass der indirekte Zugriff auf ein Attribut/Feld über eine spezielle set- und get-Syntax anstelle der „normalen“ Getter und Setter geregelt wird. Die eigentlich in der objektorientierten Programmierung äquivalenten Attribute beziehungsweise Felder werden dort als gesonderte Struktur betrachtet. Diese Syntax über Properties macht aber im Grunde genau das Gleiche wie der indirekte Zugriff über Getter und Setter, und auch Python bietet ein Sprachkonstrukt `property()`, das solch einen lesenden und schreibenden Zugriff auf private-Attribute ermöglicht. Es gibt aber einige Leute, die diese Property-Notation für besser lesbar halten.

Um eine Property zu deklarieren, deklariert man eine Getter- und eine Setter-Methode als privat und übergibt Referenzen auf diese Methoden der Anweisung `property()` als Parameter. Diese liefert dann den Zugriff auf die Instanzvariable über eine Property zurück, etwa so (*Properties.py*):

```
class Katze:
 def __init__(self, alter):
 self.__alter = alter
 def __getAlter(self):
 return self.__alter
 def __setAlter(self, alter):
 self.__x = alter
 alter = property(__getAlter, __setAlter)
obj = Katze(1)
print(obj.alter)
obj.alter = 5;
print(obj.alter)
```

Sie erkennen die zusätzliche Privatisierung der Getter- und Setter-Methode und die Angabe der Referenzen als Parameter. Die Privatisierung der Getter- und Setter-Methode ist zwar syntaktisch nicht zwingend, aber vom Design her unbedingt zu empfehlen. Das ist die gleiche Argumentation wie bei dem Zugriff über Getter und Setter auf Felder – sonst braucht man keine Property.

- ▶ Natürlich können Sie auch bei Properties einen read-only- oder write-only-Zugriff bereitstellen. Implementieren Sie mit pass einfach eine leere Anweisung in dem Getter (das ergibt einen write-only-Zugriff) oder in dem Setter (read-only-Zugriff).

---

## 11.7 Ein Objekt löschen

Der Konstruktor erzeugt ein Objekt und sein Gegenspieler ist der **Destruktor**. Dieser zerstört ein Objekt und gibt den Speicherplatz auf. In alten Sprachen wie C/C++ muss man sich selbst um die Speicherbereinigung kümmern. Aber moderne Sprachen – auch Python – haben ein automatisches Speichermanagement und kümmern sich um den automatischen Aufruf des Destruktors im Hintergrund, wenn man nicht selbst eingreifen möchte und ein Objekt nicht benötigt wird (eine sogenannte **Garbage Collection**). Aber wenn Sie wollen, können Sie ein Objekt mit der Anweisung *del* auch gezielt löschen.

- ▶ Sie können auch den Destruktor – die magische Methode `__del__` – redefinieren, wenn da bestimmte Schritte<sup>10</sup> in Verbindung mit dem Löschen des Objekts passieren sollen. Das ist aber in der Regel unnötig, und das eigentliche Löschen wird automatisch erledigt.

---

<sup>10</sup>Etwas Meldungen anzeigen oder Dinge protokollieren.

### 11.7.1 Ein Beispiel für das Redefinieren des Destruktors

In dem folgenden Beispiel redefinieren wir den Destruktor und löschen ein Objekt mit *del* manuell, auch wenn das wie gesagt in der Praxis selten notwendig ist. Es soll einfach nur gezeigt werden, wie das geht (*Klasse8.py*):

```
class Person:
 def __init__(self, name, alter):
 self.name = name
 self.alter = alter
 def __del__(self):
 print("In der Regel nicht notwendig")
 def lautgeben(self):
 print("Mein Name ist", self.name, "und ich bin", self.alter,
 "Jahre alt.")
obj = Person("Herby", 5)
print(obj.name, obj.alter)
obj.lautgeben()
del obj
```

Das ist dann die Ausgabe mit der Meldung des Destruktors:

*Herby 5*  
*Mein Name ist Herby und ich bin 5 Jahre alt.*  
*In der Regel nicht notwendig*

---

## 11.8 Ein paar besondere OO-Techniken

Kommen wir zu ein paar besonderen OO-Techniken. Diese sind nicht unbedingt Teil des „Grundwortschatzes“, können Ihnen bei der Objektorientierung aber ab und zu vielleicht ganz nützlich sein.

### 11.8.1 Eine to-string-Funktionalität bereitstellen – \_\_str\_\_

Man kennt es aus fast allen OO-Sprachen, dass der Inhalt eines Objekts stringifiziert (in Form eines Strings gespeichert) werden kann. Das bedeutet, dass der Inhalt des Objekts in eine lesbare Form gebracht wird. In Python wird die Ausgabe eines Objekts in der Grundeinstellung nur dessen Speicheradresse und den Namen anzeigen (Abb. 11.4). Damit kann man in der Regel nichts anfangen. Deshalb erzeugt man bei Bedarf eine bestimmte vorgegebene Meldung, die für so einen Fall angezeigt werden soll. Um das

```
<__main__.Person object at 0x05050DB0>
Das soll ausgegeben werden
>>>
```

**Abb. 11.4** Eine nichtssagende versus eine individuelle Meldung bei Ausgabe eines Objekts

zu machen, kann man in Python die magische Methode `__str__` nutzen und diese überschreiben.

Beispiel (*Klasse9.py*):

```
class Person:
 pass
class Gebaeude:
 def __str__(self):
 return "Das soll ausgegeben werden"
print(Person())
print(Gebaeude())
```

In der Klasse *Person* wird keine individuelle Meldung erstellt, deshalb sieht man bei `print(Person())` nur die Adresse des Objekts und dessen Objekttyp, bei einem Objekt vom Typ *Gebaeude* hingegen die gewünschte Meldung (Abb. 11.4).

### 11.8.2 Objekte dynamisch erweitern, das Dictionary `__dict__` und Slots

Wenn ein Objekt erzeugt wurde, werden die Attribute als auch Methoden eines Objektes in einem Dictionary `__dict__` gespeichert. Das wurde schon kurz angesprochen. So eine Verankerung von Attributen wie auch Methoden eines Objekts über ein Key-Value-System setzt sich derzeit auf breiter Front durch. Insbesondere JavaScript hat mit JSON (JavaScript Object Notation) einen schlanken und eleganten Referenzstandard geschaffen, an dem sich viele andere Konzepte orientieren. Wie bei jedem anderen Dictionary in Python ist nun die Anzahl der Elemente nicht begrenzt und kann dynamisch erweitert werden.

Und das bedeutet, dass einem bereits existierenden Objekt weitere Attribute und auch Methoden hinzugefügt werden können. Denn es werden ja nur Elemente einem Dictionary hinzugefügt. Hier ist ein Beispiel (*ObjektErweiterung.py*).

```
class Tier():
 def __init__(self):
 self.alter = 5
obj1 = Tier()
print("Das Dictionary __dict__:", obj1.__dict__, "\n")
```

```

obj1.name = "Herby"
print(obj1.alter)
print(obj1.name)
print("Das Dictionary __dict__:", obj1.__dict__, "\n")
def lautgeben():
 print ("Miau")
obj1.lautgeben = lautgeben
obj1.lautgeben()
print("Das Dictionary __dict__:", obj1.__dict__, "\n")
obj1.fressen = lambda x: print (x)
obj1.fressen("Mampf")
print("Das Dictionary __dict__:", obj1.__dict__, "\n")

```

Das Dictionary, welches die Attribute und Methoden des Objekts enthält, kann folgendermaßen über `obj1.__dict__` angesprochen werden. Das ist die gesamte Ausgabe:

```

Das Dictionary __dict__: {'alter': 5}
5
Herby
Das Dictionary __dict__: {'alter': 5, 'name': 'Herby'}
Miau
Das Dictionary __dict__: {'alter': 5, 'name': 'Herby', 'lautgeben': <function lautgeben at 0x0644D618>}
Mampf
Das Dictionary __dict__: {'alter': 5, 'name': 'Herby', 'lautgeben': <function lautgeben at 0x0644D618>, 'fressen': <function <lambda> at 0x0644D660>}
```

Die Klasse `Tier` stellt bloß die Instanzvariable `alter` bereit. Aber mit `obj1.name = "Herby"` wird einem vorher bereits erzeugten Objekt eine neue Instanzeigenschaft hinzugefügt. Die Erweiterung erfolgt also, nachdem es bereits definiert worden ist.

Aber auch Methoden werden wie gesagt nur als Referenzen in dem Dictionary verankert und können deshalb nachträglich hinzugefügt werden. Sie sehen das in dem Beispiel auf zwei Weisen umgesetzt:

- Es gibt die Deklaration einer „normalen“ Funktion, die über eine Referenz nachträglich dem Objekt hinzugefügt wird.
- Über einen `lambda`-Ausdruck wird eine anonyme Funktion direkt in dem Objekt verankert.

### 11.8.2.1 Slots statt dem Standarddictionary

Das Verwalten von Attributen und Methoden über das dynamische Standarddictionary `__dict__` ist bei vielen Instanzen im Grunde ziemlich speicherfressend und auch nicht

performant. Um dem entgegenzuwirken, kommen sogenannte **SLOTS** ins Spiel. Mit Slots wird eine statische Struktur zur Verfügung gestellt, die weiteres Hinzufügen von Attributen verhindert, sobald eine Instanz erstellt wurde. Das ist ressourcenschonender und gut für die Performance.

Um Slots zu benutzen, müssen Sie in der Deklaration der Klasse eine Liste `__slots__` definieren. Die Liste muss alle Attribute beinhalten, die ein Objekt später zur Verfügung haben soll. Beispiel (*Slots.py*):

```
class Tier():
 __slots__ = ['alter']
 def __init__(self):
 self.alter = 5
obj1 = Tier()
print(obj1.alter)
```

Beim Versuch, ein weiteres Attribut dem Objekt *obj1* dynamisch hinzuzufügen, gibt es einen Fehler der Art:

```
obj1.name = "Herb"
AttributeError: 'Tier' object has no attribute 'name'.
```

Seit Python 3.3 nimmt die Bedeutung von Slots ab, denn mit dem sogenannten Key-Sharing-Dictionaries-Konzept für die Speicherung von Objekten können Attribute der verschiedenen Instanzen einer Klasse einen Teil ihres genutzten Speichers mit anderen Instanzen der Klasse teilen.

### 11.8.3 Dynamische Erzeugung von Klassen, Metaklassen und die Klasse `type`

Kommen wir noch zu einem kleinen Ausflug ist die Hintergründe von Klassen, wobei diese nur angedeutet werden. In der objektorientierten Programmierung werden Klassen auch Typen genannt, was schon besprochen wurde. Deshalb liefert aber auch die Built-in Function `type()` immer Informationen zur Klasse einer Instanz, auch bei primitiven Datentypen, was ja deutlich macht, dass diese im Hintergrund Objekte sind.

Beispiel (*Typinformationen.py*):

```
class Tier():
 def __init__(self):
 self.alter = 5
obj1 = Tier()
x = 42
print(type(obj1))
print(type(x))
```

Damit etwa erhalten Sie diese Ausgabe:

```
<class '__main__.Tier'>
<class 'int'>
```

Bei Anwendung auf eine Klasse selbst kommt die folgende Ausgabe:

```
<class 'type'>
```

Dies zeigt, dass alle Klassen selbst Instanzen einer Klasse *type* sind. Doch wo und wie ist diese Klasse *type* deklariert? Sie ist eine sogenannte **Metaklasse**. Metaklassen werden in einigen objektorientierten Konzepten nur als philosophischer Überbau eingeführt, um erklären zu können, dass Klassen selbst wieder als Objekte gesehen werden können. Nur sind es besondere Objekte, die eben über Metaklassen erzeugt werden. Aber einige Sprachen erlauben es, Metaklassen konkret zu notieren beziehungsweise darauf zuzugreifen. So auch Python, denn man kann direkt eine Subklasse der Metaklasse *type* bilden und damit dynamisch Klassen generieren. Statt eines einzigen Arguments kann *type()* dazu mit drei Parametern aufgerufen werden:

- Der erste Parameter ist der String, der den Klassennamen angibt und zu einem *name*-Attribut wird.
- Der zweite Parameter ist eine Liste oder ein Tupel mit der oder den Superklasse(n) der generierten Klasse (Abschn. 11.9). Die Liste oder das Tupel wird zu einem *bases*-Attribut.
- Das dritte Attribut ist ein Dictionary, welches als Namensraum der Klasse fungiert und alle Deklarationen der Klasse enthält. Diese Deklaration baut das *dict*-Attribut auf.

Beispiel (*DynamischeKlassen.py*):

```
Tier = type("Tier",
 (),
 {"alter":0 })
obj1 = Tier()
print(type(obj1))
print(type(Tier))
```

#### 11.8.4 Shallow Copy: flaches und tiefes Kopieren

Python kennt verschiedene Versionen von Kopievorgängen von Objekten. Diese haben Auswirkungen darauf, was kopiert und was nur referenziert wird. Die Details beruhen explizit darauf, dass in Python „alles“ ein Objekt ist. Die Konzepte von flachem Kopieren ("shallow copy") und tiefem Kopieren ("deep copy") sind dabei die üblichen Techniken.

Das **flache Kopieren** erstellt eine neue Kopie eines Objekts, kopiert jedoch nur die Referenzen auf die enthaltenen Objekte, nicht die Objekte selbst. Das ist extrem effizient und ressourcenschonend, bedeutet jedoch, dass Änderungen in den enthaltenen Objekten in der Kopie und im Originalobjekt sichtbar sind. Python bietet dazu explizit die *copy*-Library, welche die *copy*-Methode für das flache Kopieren von Objekten bereitstellt.

### Beispiel

```
import copy
original_list = [1, 2, [3, 4]]
shallow_copy = copy.copy(original_list)
Ändern des eingebetteten Elements in der Kopie
shallow_copy[2][0] = 5
print(original_list, shallow_copy) ◀
```

Im obigen Beispiel kann man an der Ausgabe `[1, 2, [5, 4]] [1, 2, [5, 4]]` erkennen, dass das flache Kopieren dazu führt, dass die enthaltenen Listen in der Kopie und im Original auf dasselbe Objekt verweisen.

Das **tiefe Kopieren** erstellt eine vollständig unabhängige Kopie eines Objekts und aller darin enthaltenen Objekte. Änderungen in der Kopie beeinflussen das Originalobjekt und seine enthaltenen Objekte nicht. In Python können Sie über die *copy*-Library das tiefe Kopieren mit der *deepcopy*-Methode durchführen (*copy2.py*).

### Beispiel

```
import copy
original_list = [1, 2, [3, 4]]
deep_copy = copy.deepcopy(original_list)
Ändern des eingebetteten Elements in der Kopie
deep_copy[2][0] = 5
print(original_list, deep_copy) ◀
```

Im obigen Beispiel sehen wir an der Ausgabe `[1, 2, [3, 4]] [1, 2, [5, 4]]`, dass das tiefe Kopieren dazu führt, dass Änderungen in der Kopie das Originalobjekt nicht beeinflussen.

Neben diesen beiden Varianten stellt Python noch eine weitere Version des Kopierens bereit, die „Klonen“ oder „Kopieren durch Zuweisung“ ("assignment copy") genannt wird und ohne zusätzliche Bibliothek auskommt. Diese dritte Methode ist jedoch weniger gebräuchlich und basiert auf dem Zuweisen eines bestehenden Objekts zu einer neuen Variablen. Dabei wird einem neuen Namen (einer neuen Variablen) eine Referenz auf dasselbe Objekt zugewiesen wie dem Originalnamen. Es ist wichtig zu beachten, dass das Klonen durch Zuweisung keine echte Kopie des Objekts erstellt, sondern lediglich eine zusätzliche Referenz auf das bereits vorhandene Objekt entsteht. Dies führt dazu, dass beide Variablen auf dasselbe Objekt im Speicher zeigen, Änderungen an einem der beiden Namen wirken sich auf das Objekt und somit auf den anderen Namen aus (*copy3.py*).

**Beispiel**

```
original_list = [1, 2, [3, 4]]
assignment_copy = original_list # Kopieren durch Zuweisung
Ändern des eingebetteten Elements in der Kopie
assignment_copy[2][0] = 5
print(original_list, assignment_copy) ◀
```

Im obigen Beispiel sehen wir, dass das Klonen durch Zuweisung dazu führt, dass beide Namen auf dasselbe Objekt verweisen. Daher wirken sich Änderungen an einem Namen auf das Objekt aus, auf das beide Namen verweisen.

### 11.8.5 Reference Counting

Eines der geheimnisvollsten Dinge in Python ist die Referenznummer bei Objekten. Die Details und Auswirkungen sind aber sehr spannend und erklären die manchmal etwas „eigenartigen“ Verhaltensweisen von Python, obwohl sie rein im Hintergrund ablaufen. Reference Counting (Referenzzählung) bezeichnet eine Technik zur Verwaltung des Speichers in Python. Bei dieser wird verfolgt, wie viele Referenzen auf ein Objekt zur Laufzeit gerade vorhanden sind. Jedes Objekt in Python enthält ein Attribut namens "Referenzzähler" (Reference Count), das angibt, wie oft das Objekt referenziert wird. Wenn der Referenzzähler eines Objekts auf null fällt, bedeutet dies, dass es nicht mehr erreichbar ist, und der Speicher, den es belegte, kann freigegeben werden. Dies ist eine grundlegende Technik zur automatischen Speicherverwaltung in Python.

Hier sind einige Schlüsselkonzepte im Zusammenhang mit Reference Counting:

- Referenzzähler erhöhen: Wenn ein neues Objekt erstellt oder eine Variable einem Objekt zugewiesen wird, wird der Referenzzähler des Objekts um den Wert 1 erhöht. Dies bedeutet, dass das Objekt eine weitere Referenz bekommen hat.
- Referenzzähler verringern: Wenn eine Variable ihren Verweis auf ein Objekt verliert, sei es, weil sie außerhalb ihres Gültigkeitsbereichs geht oder neu zugewiesen wird, wird der Referenzzähler des Objekts um den Wert 1 verringert. Wenn der Referenzzähler den Wert 0 bekommt, ist das Objekt nicht mehr erreichbar und wird als nicht mehr benötigt angesehen.
- Automatische Speicherfreigabe: Python verfolgt ständig die Referenzzähler aller Objekte und gibt den Speicher automatisch für nicht mehr benötigte Objekte frei (Garbage Collection).

Reference Counting ist eine schnelle und effiziente Methode zur Speicherverwaltung, aber es kann auch zu Problemen beim Erkennen von Zyklen in Referenzen (sogenannte

„Referenzzyklen“) kommen. Dabei kann es passieren, dass Objekte, die sich gegenseitig referenzieren, nicht ordnungsgemäß freigegeben werden, was zu Speicherlecks führt. Python verwendet zusätzlich zur Referenzzählung den Mechanismus der zyklischen Garbage Collection, um solche Probleme zu lösen.

- ▶ Es ist wichtig zu verstehen, dass man in der Regel nicht direkt mit dem Referenzzähler und der Speicherbereinigung interagieren muss, da Python die Speicherverwaltung größtenteils automatisch durchführt. Dennoch ist ein grundlegendes Verständnis dieses Konzepts hilfreich, um effizienten Python-Code zu schreiben und Speicherprobleme zu vermeiden.
- 

## 11.9 Vererbung

Eine der wichtigsten Beziehungen zwischen Klassen beziehungsweise Objekten ist die Vererbung. Dieser wollen wir uns nun widmen.

### 11.9.1 Grundlagentheorie zur Vererbung

In der objektorientierten Programmierung werden also ähnliche Objekte zu Gruppierungen (Klassen) zusammengefasst, die eine leichtere Klassifizierung der Objekte ermöglichen. Zentrale Bedeutung hat dabei die hierarchische Struktur der Gruppierungen, von allgemein bis fein. Das kann man sich über ein Beispiel aus der realen Welt klarmachen, das als Modell zu verstehen ist:

- Ein Objekt vom Typ *Schaf* hat spezifische Eigenschaften und Methoden, gehört aber ebenso zu einer „übergeordneten“ Klasse *Säugetier*. Objekten dieser Klasse fehlen alle speziellen Eigenschaften und Methoden, die ein Schaf von anderen Säugetieren unterscheiden. Da sind nur diejenigen Eigenschaften und Methoden zu finden, die allen Säugetieren zugehörig sind.
- Diese Klasse *Säugetier* wiederum gehört in unserem Modell zu einer höheren Klasse *Tier* und diese wieder zu der Klasse *Lebewesen*.

So kann man das Verfahren fortsetzen. In jeder dieser übergeordneten Klassen findet eine Reduzierung auf nur solche Eigenschaften und Methoden statt, die wirklich auch für alle denkbaren Instanzen dieser Objekttypen gelten.

Gemeinsame Erscheinungsbilder sollten also in der objektorientierten Philosophie in einer möglichst hohen Klasse zusammengefasst werden. Erst wenn Unterscheidungen möglich beziehungsweise notwendig sind, die nicht für alle Mitglieder einer Klasse gelten, werden Untergruppierungen – untergeordnete Klassen – gebildet.

**Vererbung** bezeichnet nun eine Verbindung zwischen einer Klasse und einer oder mehreren anderen Klassen, in der die abgeleitete Klasse das Verhalten und den Aufbau der Oberklassen übernimmt, gegebenenfalls neues Verhalten und einen erweiterten Aufbau besitzt und gegebenenfalls übernommenes Verhalten modifiziert.

### ► Definition

- Die vererbende Klasse nennt man **Basisklasse**, **Oberklasse**, **Elternklasse** oder **Superklasse**.
- Die Klasse, die erbtl., nennt man **abgeleitete Klasse**, **Unterklasse**, **Kindklasse** oder **Subklasse**.
- Subklassen sind immer als **Spezialisierung** einer Superklasse zu sehen, während die Superklasse die **Verallgemeinerung** ist.

Die ineinander geschachtelten Klassen bilden einen sogenannten **Klassenbaum**. Dieser kann im Prinzip beliebig tief werden. Eben so tief, wie es notwendig ist, um eine Problemstellung detailliert zu beschreiben. Vererbung ist über eine beliebige Anzahl von Ebenen im Klassenbaum hinweg möglich. Die oberste Klasse des Baums heißt **Wurzelklasse** ("root class"). Die Klassen am Ende eines Vererbungsbaumes heißen **Blattklassen** ("leaf classes").

#### 11.9.1.1 Mehrfachvererbung versus Einfachvererbung

In der Theorie der Objektorientierung gibt es sogenannte **Einfachvererbung** sowie **Mehrfachvererbung**. In der Einfachvererbung (engl. "single inheritance") gilt für die Klassenhierarchie in einer baumartigen Struktur die Voraussetzung, dass eine Subklasse immer nur genau eine Superklasse hat und eine Superklasse eine beliebige Anzahl an Subklassen haben kann. Wenn jedoch die Möglichkeit besteht, eine einzige Klasse direkt mit beliebig vielen Superklassen durch die Vererbung zu verknüpfen, nennt man dies Mehrfachvererbung (engl. "multiple inheritance"). Objekte der Subklasse erben direkt Eigenschaften aus verschiedenen Superklassen.

Python verwendet Mehrfachvererbung.

#### 11.9.2 Umsetzung von Vererbung in Python

**Vererbung** wird in Python bei der Deklaration einer Klasse dadurch erreicht, dass die Superklasse in Klammern dem Bezeichner der deklarierten Subklasse nachgestellt wird.

Beispiel (*Klasse10.py*):

```
class Tier:
 def __init__(self, alter):
 self.alter = alter
```

```
class Katze (Tier):
 def __init__(self, alter, name):
 Tier.__init__(self, alter)
 self.name = name
kater = Katze(5, "Herby")
print(kater.name, kater.alter)
```

Ein Objekt vom Typ *Katze* wird also auch über sämtliche Eigenschaften und Methoden der Superklasse *Tier* verfügen. Ebenso sollte Ihnen auffallen, dass in dem Beispiel in einer Python-Quelltextdatei zwei Klassen deklariert werden. Das ist durchaus erlaubt.

#### ► **Wichtig**

Beachten Sie in der magischen Methode `__init__` der Subklasse den Zugriff auf den Konstruktor der Superklasse über `Tier.__init__(self, alter)` als erste Anweisung. Nur mit der Verwendung des Konstruktors der Superklasse wird die Vererbung der Eigenschaften und Methoden von der Superklasse gewährleistet.

Ab Python 3 kann man in einem Konstruktor (also der magischen Methode `__init__`) einfach `super()` notieren, um damit auf den Konstruktor der Superklasse zugreifen. Das geht so:

```
super().__init__(self, alter)
```

Wenn bei der Deklaration einer Klasse keine Superklasse angegeben wird, wird immer die Klasse *object* als Superklasse genommen. Diese ist die oberste Klasse sämtlicher Python-Klassen.

### 11.9.3 Mehrfachvererbung in Python

Python gehört zu den ganz wenigen modernen Programmiersprachen, die **Mehrfachvererbung** unterstützen. Damit kann man mehrere Superklassen angeben, und die Subklasse erbt von mehreren Superklassen.

Beispiel (*Klasse11.py*):

```
class Tier:
 def __init__(self, alter):
 self.alter = alter
class Eigentuemer:
 def __init__(self, wohnort):
 self_wohnort = wohnort
class Katze (Tier, Eigentuemer):
 def __init__(self, alter, name, wohnort):
 Tier.__init__(self, alter)
```

```
Eigentuemer.__init__(self,wohnort)
 self.name = name
kater = Katze(5, „Herby“, "Bodenheim")
print(kater.name,kater.alter,kater_wohnort)
```

Die Klasse *Katze* ist Subklasse von *Tier* und *Eigentuemer*. Die Eigenschaft *alter* stammt von der Superklasse *Tier*, während die Eigenschaft *wohnort* von der Superklasse *Eigentuemer* vererbt wird. Beachten Sie die beiden Zugriffe auf die magischen Methoden der jeweiligen Superklasse.

#### 11.9.3.1 Die Sache mit MRO

Bei Mehrfachvererbung in Python kann es zu Mehrdeutigkeiten kommen, denen mit dem Konzept der „Method Resolution Order“ (MRO) begegnet wird. Dies bestimmt, wie Python Mitglieder einer Klasse auswählt, wenn eine Instanz dieser Klasse ein Mitglied nutzt, das in mehreren vererbten Klassen definiert ist.

Wenn man in Python von mehreren Klassen gleichzeitig erbt, kann eine abgeleitete Klasse identisch benannte Attribute und Methoden von mehr als einer Elternklasse erben. Das MRO-System stellt sicher, dass insbesondere die Methoden in einer hierarchischen Vererbungsstruktur in einer bestimmten Reihenfolge aufgelöst werden.

Die MRO ist die Reihenfolge, in der Python die Elternklassen durchsucht. Python verwendet einen Algorithmus namens C3 Linearization (auch C3 Superklassenalgorithmus genannt), um die MRO für eine Klasse und deren Hierarchie zu bestimmen. Der MRO-Algorithmus achtet darauf, dass die Vererbungshierarchie konsistent und widerspruchsfrei bleibt.

Dabei kommt in Python oft die *super*-Funktion zum Einsatz, um auf die Mitglieder einer Elternklasse zuzugreifen. Auch sie berücksichtigt die MRO und stellt sicher, dass die Mitglieder aus der korrekten Klasse aufgerufen werden.

Beispiel (*mro1.py*):

---

#### Beispiel

```
class A:
 i = 42
class B:
 i = 1
class C(A,B):
 def __init__(self):
 print(super().i)
C() ▶
```

Sowohl in der Klasse *A* als auch in der Klasse *B* gibt es das Attribut *i*. Bei der Initialisierung der Subklasse *C* wird mit *super().i* das Attribut aus der Klasse *A* verwendet. Aber warum nicht aus der Klasse *B*?

Die Antwort ist, dass die Klasse A in der Vererbungsangabe zuerst notiert wird. Schauen wir uns noch ein etwas komplexeres Beispiel an, um das Konzept zu verdeutlichen. Beispiel (*mro2.py*):

---

**Beispiel**

```
class A:
 def reden(self):
 print(„A“)
class B(A):
 def reden(self):
 print(„B“)
class C(A):
 def reden(self):
 print(„C“)
class D(B, C):
 pass
d = D()
d.reden() ◀
```

In diesem Fall ist die MRO für die Klasse `D` `D -> B -> C -> A`. Das bedeutet, wenn wir `d.reden()` aufrufen, wird die Methode aus Klasse `B` aufgerufen, da sie zuerst in der MRO erscheint.

Man kann sich die MRO auch durch die Methode `mro()` ausgeben lassen. Beispiel (*mro3.py*):

---

**Beispiel**

```
class A:
 def process(self):
 print(„Klasse A“)
class B:
 def process(self):
 print(„Klasse B“)
class C(A, B):
 pass
obj = C()
obj.process()
print(C.mro()) ◀
```

Die Ausgabe ist das:

Klasse A  
[<class ,\_\_main\_\_.C>, <class ,\_\_main\_\_.A>, <class ,\_\_main\_\_.B>, <class ,object>]

## 11.9.4 Polymorphie über Überschreiben und Überladen

Wir hatten schon angedeutet, dass der Begriff Polymorphie nicht ganz eindeutig ist. Allgemein bezeichnet man damit eine Auswahl einer Operation, die nicht nur vom Bezeichner der Operation, sondern dem Kontext abhängt. Die zentralen Techniken zur Implementierung von Polymorphie nennen sich **Überschreiben** und **Überladen**.

### 11.9.4.1 Überschreiben von Methoden

Wenn man bei Vererbung Methoden (und im Grunde auch andere Dinge wie Eigenschaften) in Superklasse und Subklasse gleich benennt und diese über Vererbung in Verbindung stehen (also in einer Superklasse-Subklassen-Beziehung), **überschreiben** Sie die Methode in der Subklasse, egal über wie viele Ebenen. Beim Überschreiben (**Overriding**)<sup>11</sup> überlagert eine Methode in einer Subklasse eine Methode einer Superklasse mit der exakt identischen Signatur, soweit sie vom System unterschieden wird. Die überschreibende Methode verfeinert und ersetzt die Methodenimplementierung der Superklasse. Dabei wird die Methode der Superklasse verdeckt, wobei gerade „verdecken“ in dem Zusammenhang oft ein sehr sinnvoller Begriff ist, denn gerade Eigenschaften der Superklasse werden in der Regel verdeckt und nicht in dem Sinn überschrieben. Aber das ist eine etwas spitzfindige Behandlung des Vorgangs, der in Python – im Gegensatz zu anderen Sprachen – nicht wirklich relevant ist.

#### 11.9.4.1.1 Ein Beispiel zum Überschreiben einer vererbten Methode

Hier ist ein einfaches Beispiel, in dem eine vererbte Methode aus einer Superklasse überschrieben wird (*Klasse12.py*).

```
class Tier(object):
 def __init__(self):
 self.alter = 5
 def getAlter(self):
 return self.alter

class Katze(Tier):
 def getAlter(self):
 return self.alter + 1
```

---

<sup>11</sup>Eigentlich besser mit „überdefinieren“ übersetzt.

```
obj1 = Tier()
obj2 = Katze()
print(obj1.getAlter())
print(obj2.getAlter())
```

In der Superklasse *Tier* wird eine Instanzmethode *getAlter()* deklariert, die eine Instanz-eigenschaft *alter* anlegt. Diese wird an die Subklasse *Katze* vererbt.

Beachten Sie, dass *Tier* explizit von *object* ableitet. Wenn diese Notation der Superklasse fehlt, ist das Vorgabe. Aber man notiert das dennoch häufiger zusätzlich, um diese Vererbung deutlich zu machen.

In der Subklasse *Katze* wird die Methode überschrieben. Ein Objekt vom Typ *Tier* ruft also die Methode der Superklasse, ein Objekt von Typ *Katze* jedoch die Methode der Subklasse auf – auch wenn diese auf die vererbte Eigenschaft *alter* der Superklasse *Tier* zugreift.

#### 11.9.4.1.2 Zugriff auf die Superklasse und *super()*

Es kann vorkommen, dass man in einer abgeleiteten Klasse mit einer überschriebenen Methode auf ein verdecktes Element der Superklasse zugreifen muss. Das geht – wie etwas weiter oben schon gesehen – mit *super()*, etwa so (*Klasse13.py*):

```
class Tier(object):
 def __init__(self):
 self.alter = 5
 def getAlter(self):
 return self.alter

class Katze(Tier):
 def getAlter(self):
 return super(Katze, self).getAlter() + 1
obj1 = Tier()
obj2 = Katze()
print(obj1.getAlter())
print(obj2.getAlter())
```

In der überschriebenen Methode gibt man in *super()* als ersten Parameter die aktuelle Klasse und als zweiten *self* an. Das referenziert die Superklasse, und die folgende Punkt-notation erlaubt den Zugriff auf die überschriebene Methode in der Superklasse.

#### 11.9.4.2 Überladen

Nun gibt es in der OO-Theorie noch das genannte Überladen (engl. "overloading") von Methoden oder auch Operatoren. Beim Überladen von Methoden würde zur Auswahl verschiedener Methoden mit gleichem Namen die Parameterliste zur Unterscheidung

herangezogen. So etwas geht etwa in Java oder C#.<sup>12</sup> Python erlaubt jedoch nur das Überladen von Operatoren über einige der „magischen Methoden“, was schon angedeutet wurde und wir aber nicht tiefer verfolgen wollen.<sup>13</sup> Aber der Verzicht auf das Überladen von Methoden in Python ist keine wirkliche Einschränkung, denn durch die mögliche flexible Verwendung von Parametern und Vorgabewerten kann man mit einer geeigneten internen Programmierung ein überladenes Verhalten auf einfache Weise simulieren.

---

## 11.10 Was ist mit Schnittstellen und abstrakten Klassen in Python?

In den meisten objektorientierten Programmiersprachen gehören sogenannte abstrakte Klassen und Schnittstellen (Interfaces) zu den elementaren Mitteln, um Vererbungshierarchien zu organisieren. Insbesondere in Java oder C# kommt man ohne diese Techniken nicht aus. Im Wesentlichen bezeichnet beziehungsweise kennzeichnet eine so ausgezeichnete Struktur (potenziell) unvollständigen Code. Und damit dieser verwendet werden kann, erzwingt das gewisse Verhaltensweisen, die hauptsächlich der Vervollständigung dieses Codes vor einer konkreten Instanzierung dienen.

### 11.10.1 Abstrakte Superklassen

Grundsätzlich kann man in Python abstrakte Klassen deklarieren, die dann als abstrakte Superklassen für die Vererbung dienen. Denn abstrakter Code muss wie gesagt vor einer konkreten Instanzierung vervollständigt werden. Und das macht man bei abstrakten Basisklassen in Subklassen, die diese Klassen erweitern.

Die Verwendung von abstrakten Basisklassen ist in Python jedoch nicht ganz so „natürlich“ möglich, wie man es von Sprachen wie Java, C/C++ oder C# gewohnt ist, und das Konzept ist nicht ganz trivial umgesetzt. Deshalb soll hier nur angedeutet werden, wie man in Python vorgehen kann.

Diese „abc“ (Abstract Base Classes oder auf Deutsch „Abstrakte Basisklassen“) nutzen in Python ein zusätzliches Modul (Abschn. 11.8) namens *abc.py*, das erst hinzugebunden werden muss. Erst dieses Modul stellt überhaupt die Infrastruktur zum

---

<sup>12</sup>Also streng typisierten Sprachen.

<sup>13</sup>Python erlaubt es etwa in eigenen Klassen, dem +-Operator eine eigene Bedeutung zu geben, indem die Methode `__add__` redefiniert wird. Für den Operator - würde etwa `__sub__` redefiniert. Aber meines Erachtens ist so eine Überladung hochgefährlich, schlecht lesbar und schlecht wartbar. Ich rate davon ab.

Definieren abstrakter Superklassen in Python bereit.<sup>14</sup> Doch eigentlich braucht man in Python sowieso nur sehr, sehr selten eine abstrakte Klasse. Denn wie schon mit der Überschrift deutlich gemacht, verwendet man diese hauptsächlich zur Strukturierung eines Klassenbaums. Und das muss man in Python selten mit abstrakten Klassen machen, denn man hat Mehrfachvererbung, und damit erst recht keine Notwendigkeit zur Bereitstellung von Schnittstellen, die es in Python auch gar nicht gibt.

### 11.10.2 Was ist im Allgemeinen eine Schnittstelle?

Eine Schnittstelle (gelegentlich auch Interface-Klasse oder kurz Interface genannt) ist in Sprachen, die so etwas bereitstellen, eine spezielle Form einer abstrakten Klasse. Sie kann aber ausschließlich abstrakte<sup>15</sup> Methoden und Konstanten enthalten. Schnittstellen ermöglichen damit ähnlich wie abstrakte Klassen das Erstellen von Pseudoklassen, die ganz aus abstrakten Methoden und gegebenenfalls Konstanten zusammengesetzt sind. Schnittstellen werden jedoch nicht wie Superklassen zur Vererbung verwendet, sondern werden in Klassen oder anderen Schnittstellen implementiert.

Damit kann man sie in Programmiersprachen mit Einfachvererbung als Alternative zur Mehrfachvererbung benutzen. Sie lassen es zu, eine bestimmte Funktionalität zu definieren, die in mehreren Klassen benutzt werden soll, aber bei der die genaue Umsetzung noch nicht sicher ist und die konkrete Ausprogrammierung erst in der implementierenden Klasse erfolgt. Sofern Sie derartige Methoden in einer Schnittstelle unterbringen, können Sie gemeinsame Verhaltensweisen definieren und die spezifische Implementierung dann den Klassen selbst überlassen. Bei Schnittstellen redet man manchmal auch von „leichter Mehrfachvererbung“, und wenn Python schon die richtige Mehrfachvererbung bereitstellt, wird klar, warum man einfach keine Schnittstellen benötigt.

---

## 11.11 Module und Pakete

In Python können Klassen in verschiedene Module verteilt werden. Das sind erst einmal einfach .py-Dateien mit Klassendefinitionen. Ein Modul in Python ist also erst einmal nur eine einzelne Datei mit Python-Code. Ein Paket in Python ist eine Verzeichnisstruktur, die eine Sammlung von Modulen (als .py-Dateien) und möglicherweise weitere Pakete enthält. Ein Paket muss eine spezielle Datei namens `__init__.py` in seinem Verzeichnis enthalten, um als Paket erkannt zu werden. Der Inhalt der Datei ist irrelevant

---

<sup>14</sup>Das zeigt schon, dass abstrakte Klassen nicht so natürlich in das Kernkonzept von Python eingebaut wurden, wie es in Java oder C# der Fall ist.

<sup>15</sup>Methoden ohne Implementierung.

und diese in der Regel leer. In einem Paket können Sie verwandte Module gruppieren, um den Code besser zu organisieren.

In diesen Moduldateien werden in Python also in der Regel eine oder mehrere Klasse(n) definiert. Diese Datei kann aber auch außerhalb von Klassen definierte Variablen und Funktionen enthalten – da zeigt sich wieder, dass Python kein bestimmtes Programmier-Paradigma erzwingt. Man kann nun in Python vorgefertigte Standardmodule eines APIs verwenden (was wir später im Buch noch mehrfach machen werden), aber auch eigene Module erstellen und verwenden.

Man kann sich das so vorstellen: In Python sind Pakete und Module logische Organisationsstrukturen, die entwickelt wurden, um Code in größeren Projekten zu organisieren und zu strukturieren und damit übersichtlich zu halten, die Wiederverwendbarkeit zu fördern und die Kollisionsgefahr von Namensräumen zu verringern.

### 11.11.1 Die import-Anweisung

Python nennt nun die Gruppierung von Code in getrennte Strukturen **Module**. In Dateien, in denen diese Module verwendet werden sollen, notiert man eine *import*-Anweisung und gibt den Namen des Moduls (der Dateiname ohne Erweiterung) an. Der Zugriff erfolgt dann über den Namen und die Punktnotation. Wir werden gleich noch darauf eingehen, dass Pakete eine objektorientierte Abstraktion von Verzeichnissen sind (Abschn. 11.11.3). Module sind hingegen eine objektorientierte Abstraktion von Dateien. Auf diese Weise kann man direkt auf die Bestandteile dieser „Objekte“ zugreifen – eben die Klassen.

Beispiel:

In der Python-Datei *Tier.py* soll folgender Code stehen:

```
class Tier:
 def __init__(self, alter):
 self.alter = alter
```

Das ist eine einfache Klassendeklaration mit einer Eigenschaft ohne weitere erkläруngsbedürftige Besonderheiten.

Das soll nun der Code einer Subklasse *Katze* sein, die in einer anderen Datei mit Namen *Katze.py* deklariert ist. Sie soll aber das Modul *Tier* verwenden können, deshalb wird das am Anfang importiert:

```
import Tier

class Katze (Tier.Tier):
 def __init__(self, alter, name):
```

```
Tier.Tier.__init__(self, alter)
 self.name = name
```

- ▶ Beachten Sie die Notation *Tier.Tier*. Das erste *Tier* ist das Modul (also die Datei), während das zweite *Tier* für die Klasse in dem Modul steht. Die gleichen Bezeichner sind durchaus möglich. Es gibt ja auch Personen, die etwa „Hans Hans“ heißen.

In der Datei *Klasse7.py* soll nur die Klasse *Katze* instanziert werden. Dazu muss das Modul *Katze* (nicht aber das Modul *Tier*) importiert werden. Der Zugriff auf die Klasse geht wie oben über den Namen des Moduls, einen Punkt und dann den Namen der Klasse.

```
import Katze

obj = Katze.Katze(5, "Herby")
print(obj.name, obj.alter)
```

## 11.11.2 Importieren mit from

Wenn Sie statt der einfachen *import*-Anweisung eine *from*-Anweisung verwenden, können Sie im folgenden Code auf das Voranstellen des Moduls verzichten. Das ist ganz bequem und auch kürzer.

Erzeugen wir kurz eine Abwandlung der *Klasse15.py* mit Namen *Klasse15.py*:

```
from Katze import Katze

obj = Katze(5, "Herby")
print(obj.name, obj.alter)
```

- ▶ Sie können durch Kommata getrennt auch mehrere Klassen aus dem Modul importieren. Wenn Sie den Stern \* statt einem Klassennamen notieren, werden alle Klassen aus dem Modul so importiert.

Wie schon früher erwähnt, wird das *\_name\_*-Attribut auf den Namen des Moduls (dem Dateinamen ohne Erweiterung) gesetzt, wenn ein Modul importiert wird. Dieses Attribut können Sie gegebenenfalls abfragen.

## 11.11.3 Pakete

Gehen wir diese Modularisierung jetzt noch allgemeiner an. Oft ist es wie gesagt notwendig, dass man Codestrukturen zu Modulen zusammenfasst, die auf verschiedene

Weise erzeugt werden können, etwa über einzelne Dateien. Aber diese will man zum Beispiel selbst wiederum in Verzeichnissen gruppieren. Pakete sind in der objektorientierten Programmierung – vereinfacht – die objektorientierte Abbildung von Verzeichnissen, in denen Klassen eingesortiert werden können. Pakete (engl. "packages") sind also Gruppierungen von Klassen etc. eines Verzeichnisses. Sie sind eine Art objektorientierte Bibliotheken vieler anderer Computersprachen und bilden im allgemeinen Sinn Namensräume.

In Python legt man also einfach die zusammengehörenden `.py`-Dateien in ein Verzeichnis. Nur wird man dort – wie schon erwähnt – zusätzlich eine leere Textdatei mit Namen `__init__.py` (wie die magische Methode) erzeugen. Das braucht Python, um ein Verzeichnis als Paket zu erkennen.

- ▶ Es kann sein, dass man in gewissen Situationen auf die Datei `__init__.py` verzichten kann. Aber zur Sicherheit sollte man sie auf jeden Fall in einem Verzeichnis notieren, dass von Python als Paket verstanden werden soll.

Allerdings muss man dann aufpassen, in welchen Beziehungen die Verzeichnisse beziehungsweise Pakete und die enthaltenen Dateien zueinander stehen.

Angenommen, es gibt ein Paket/Verzeichnis, in dem sich die Datei `Klasse16.py` befindet. Das kann durchaus das Hauptverzeichnis sein.

In einem Unterordner/Unterpaket davon mit Namen `rjs` befinden sich neben der Datei `__init__.py` die Dateien `Tier.py` und `Katze.py` (Abb. 11.5).

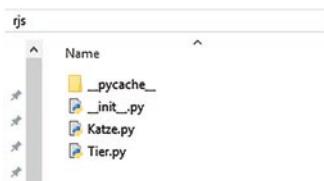
Diese sollen zu einem Modul zusammengefasst sein, wie wir es oben beschrieben haben. Nur muss die Klasse `Katze` ja selbst auf die Datei der Superklasse `Tier` referenzieren. Von wo aus wird nun die Angabe des Pakets gesehen?

Das ist die Datei `Klasse16.py` im Projektverzeichnis:

```
from rjs.Katze import *

obj = Katze(5, "Herby")
print(obj.name, obj.alter)
```

**Abb. 11.5** Das Verzeichnis `rjs` mit der Kennzeichnungsdatei `__init__.py` und den beiden Moduldateien



Der Name des Unterverzeichnisses wird über dessen Objektrepräsentation beim Import angegeben, alle Klassen aus dem Modul werden importiert. Das ist die Datei *Katze.py*:

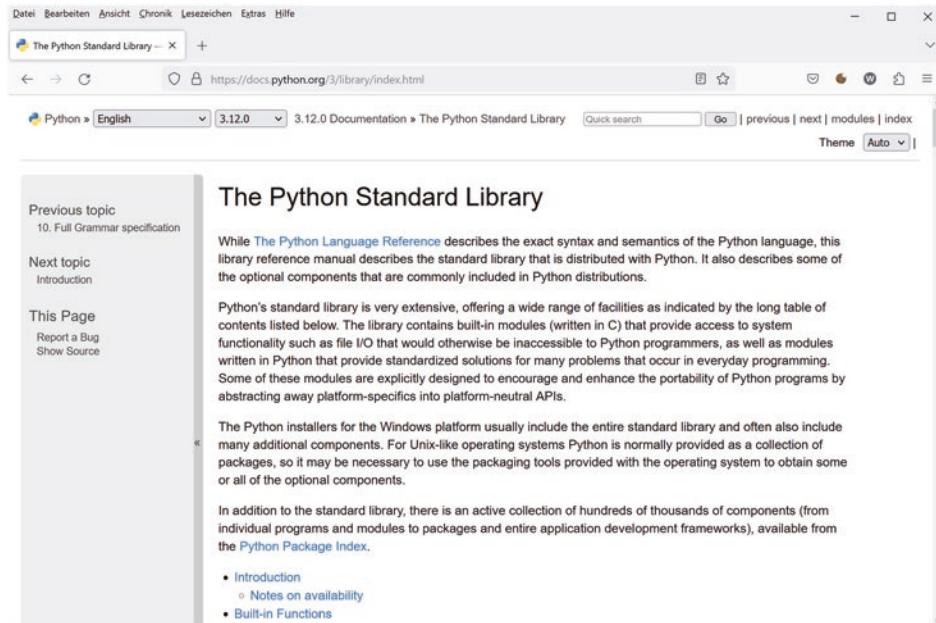
```
from rjs.Tier import *

class Katze (Tier):
 def __init__(self,alter,name):
 Tier.__init__(self,alter)
 self.name = name
```

Auch da wird also der Import aus Sicht des **Projektverzeichnisses** angegeben!

#### 11.11.4 Das Python-API

Python kommt mit einer riesigen Bibliothek von Standardmodulen daher, die in der **Python Library Reference** (<https://docs.python.org/3/library/>) genau beschrieben sind (Abb. 11.6).



**Abb. 11.6** Die Python Library Reference

- ▶ Einige Module sind auf allen Plattformen verfügbar, manche nur in bestimmten Umgebungen. Zum Beispiel ist das *winreg*-Modul nur auf Windows-Systemen verfügbar.

Die Standardbibliothek von Python beinhaltet zahlreiche unterschiedliche Module, um beispielsweise Zeit- und Datumsberechnungen durchzuführen, über Daten zu iterieren, grafische Oberflächen zu erstellen, auf das bestehende Dateisystem zuzugreifen sowie Werte in Datenbanken zu speichern und vorhandene Daten einzulesen. In den folgenden Kapiteln schauen wir uns einige Beispielthemen an. Diese setzen dann aber auch die jeweils verschiedenen notwendigen Importe der Standardmodule voraus.

### 11.11.5 Fremde Pakete nutzen

Das Standard-API von Python stellt eine Vielzahl an Erweiterungen bereit. Oft möchte man allerdings fremde Erweiterungen nutzen. In dem Fall ist der Standardweg, dass man mittels pip diese Fremdressourcen verfügbar macht. Fast alle Python-Ressourcen, die frei zur Verfügung gestellt werden, sind über dieses Paketverwaltungswerkzeug für Python nutzbar. Sie können damit Python-Pakete aus dem Python Package Index (PyPI) und anderen Quellen herunterladen, installieren, aktualisieren und allgemein verwalten.

Die Installation von neuen Python-Paketen erfolgt in der Regel einfach, indem nach dem Token *install* der Namen des Pakets oder Moduls angegeben wird:

*pip install requests*

Hiermit wird das Paket *requests* installiert, das häufig für HTTP-Anfragen in Python verwendet wird.

- ▶ Den konkreten Namen eines Pakets erfahren Sie in der Regel auf der Webseite eines Projekts, wenn Sie diesen nicht kennen.

Eine Aktualisierung auf die neueste Version eines Pakets geschieht mithilfe des Befehls *pip install --upgrade*, gefolgt vom Paketnamen.

Eine Deinstallation von Paketen erfolgt einfach mit dem Befehl *pip uninstall* gefolgt vom Paketnamen.

- ▶ Alternativen zu pip sind etwa Conda (das schon im Zusammenhang mit der Anaconda-Distribution erwähnt wurde), Poetry, Hatch oder Crank. Meist sind diese aber für spezielle Anforderungen gedacht.



# Exception-Handling – Ausnahmsweise

12

## 12.1 Was behandeln wir in diesem Kapitel?

### Übersicht

In jeder Form der Programmierung muss man mit **Fehlersituationen** und ungeplanten Vorgängen umgehen. Dabei muss man verschiedene Situationen unterscheiden:

- typografische Fehler beim Schreiben des Quelltextes,
- syntaktische Fehler beim Schreiben des Quelltextes.
- Programmfehler zur Laufzeit, die auf logische Fehler im Programmaufbau zurückzuführen sind,
- Programmfehler und unklare Situationen zur Laufzeit, die auf äußere Umstände zurückzuführen sind.

Die Bereinigung der ersten drei Punkte ist hauptsächlich das, was man unter **Debugging** versteht. Die Behandlung des vierten Punkts jedoch zählt im Wesentlichen zum sogenannten **Exception-Handling** beziehungsweise der **Ausnahmebehandlung** – sofern eine Programmiersprache dieses Konzept unterstützt, was bei Python der Fall ist. Diesem Thema widmen wir uns in diesem Kapitel.

## 12.2 Was sind Ausnahmen?

Unter einer Ausnahme kann man eine Unterbrechung des normalen Programmablaufs aufgrund einer besonderen Situation verstehen, die eine isolierte und unverzügliche Behandlung notwendig macht. Der normale Programmablauf kann erst fortgesetzt werden, wenn diese Ausnahme behandelt wurde. Andernfalls wird das Programm beendet.

Ein klassisches Beispiel, das eine solche Situation provoziert, ist der Versuch, auf einen Wechseldatenträger zuzugreifen, in dem kein Medium eingelegt oder kein Speicherplatz mehr vorhanden ist. Natürlich kann man vor einem Zugriff auf einen Wechseldatenträger überprüfen, ob ein Medium eingelegt ist oder genügend Speicherplatz zur Verfügung steht.

Aber nicht jede Situation kann man so prophylaktisch abfangen. Denn viele potentielle Probleme lassen sich auch gar nicht so leicht erkennen. Dann ist das „Werfen“ einer Ausnahme im Fall eines Problems sehr sinnvoll.

In einer objektorientierten Sprache wie Python ist eine Ausnahme aber auch ein **Mitteilungsobjekt**.<sup>1</sup> Es gibt Informationen zurück, welcher Fehler genau vorliegt, und aufgrund dieser Information kann der Programmierer Gegenmaßnahmen ergreifen. Wobei diese Reaktion meist sogar erzwungen wird, weil das Auftreten einer Ausnahme ohne Gegenmaßnahmen ein Programm beendet.

Python hat wie gesagt solch ein Ausnahmekonzept integriert.

Auch wenn es sehr oft so behauptet wird – Ausnahmebehandlung ist **nicht** auf reine Fehler im engen Sinn begrenzt, sondern beschäftigt sich mit jeder Form einer kritischen Situation zur Laufzeit eines Programms, die eine unverzügliche Reaktion erfordern und mit Gegenmaßnahmen behandelt werden kann. Obwohl es wenig sinnvoll ist, kann man mit dem gezielten Werfen und Auffangen von Ausnahmen sogar ein Programm steuern. Wir werden dennoch im Zusammenhang mit Ausnahmen sehr oft von Fehlern sprechen, denn das ist im Allgemeinen üblich so.

---

## 12.3 Warum ein Ausnahmekonzept?

Ein Ausnahmekonzept ist nicht unabdingbar. Immerhin hatten ältere Programmiersprachen so ein Konzept ja nicht implementiert. Sie können auch mit selbst programmierten Kontrollmechanismen jeden Fehler in einem Programm abfangen, den Sie als potenzielle Fehlerquelle erkennen. Aber da haben wir den entscheidenden Schwachpunkt – Sie müssen die potenzielle Fehlerquelle erkennen, was oft nahezu unmöglich ist. Gerade in komplexen Programmen können Sie nie alle kritischen Zusammenhänge überblicken. Es werden immer Fehlerquellen da sein, die an kaum beachteten Stellen

---

<sup>1</sup> Deshalb taucht die Ausnahmebehandlung auch erst nach dem Kapitel zur objektorientierten Programmierung auf.

**Abb. 12.1** Eingabe einer Zahl

```
Bitte eine Ganzzahl (integer) eingeben: 4
16
>>>
```

aufreten. Außerdem ist es sehr viel Arbeit, bei jeder Anweisung im Quelltext zu überlegen, ob da eine Gefahr lauert, und diese dann sicher abzufangen. Es ist sogar so, dass Sie in der überwiegenden Anzahl von auffangbaren Fehlern das Rad neu erfinden, denn es gibt sehr oft schon Standardmaßnahmen. Zu guter Letzt setzt diese individuelle Technik der Laufzeitfehlerbehandlung oft voraus, dass der Fehler selbst nicht eintritt, sondern vorher erkannt und umgebogen wird.

Vorsorge ist meist besser als Ausnahmebehandlung! Trotz des sicheren Konzepts der Ausnahmebehandlung ist ein vorheriges Auffangen von potenziellen Problemen vorzuziehen – wenn es geht und man eine Problemsituation entschärfen kann und sich auch der potenziellen Probleme bewusst ist! Das Auftreten einer Ausnahme ist für die Performance, den gradlinigen Programmfluss und den Ressourcenverbrauch negativ.

---

## 12.4 Konkrete Ausnahmebehandlung in Python

Da Python einen Mechanismus zur Erzeugung von Ausnahmen bereitstellt, ist klar, dass es auch einen standardisierten Mechanismus zu deren Auswertung und Behandlung besitzt. Die Realisierung der Ausnahmebehandlung sieht meist so aus, dass im Fall einer Ausnahmesituation eine Behandlungsmaßnahme anstelle der eigentlich vorgesehenen Maßnahme durchgeführt wird. In den meisten Sprachen (auch in Python), werden dazu potenziell kritische Codeteile in einem *try*-Block eingeschlossen. Die Reaktionsschritte beim Auftreten einer Ausnahme werden in einem *except*-Block notiert.<sup>2</sup>

### 12.4.1 Ein erstes Beispiel mit einfacher Ausnahmebehandlung

Schauen wir uns ein einfaches Beispiel an. Ein Benutzer soll in der Konsole eine ganze Zahl eingeben und diese soll mit dem Wert 4 multipliziert werden. Die Eingabe wird allerdings als String erfolgen, und diese Eingabe müssen wir vor der Multiplikation in ein Integer wandeln (*OhneAbsicherung.py* – Abb. 12.1).

Beim Casting kann es jedoch zu einem Fehler kommen, wenn der Eingabestring kein gültiges Integer-Format einhält. Es wird dann die Ausnahme vom Typ *ValueError*

---

<sup>2</sup>In den meisten anderen Programmiersprachen mit Ausnahmebehandlung nutzt man dazu das Schlüsselwort *catch*.

generiert, das Programm beendet, und in der Konsole gibt es eine Traceback-Meldung (Abb. 12.2).

Mithilfe der Ausnahmebehandlung kann man aber eine Gegenmaßnahme erstellen und eine robuste Eingabeaufforderung zur Eingabe einer Integer-Zahl aufbauen. Das werden wir erst einmal ganz einfach halten. Es genügt, dass der Anwender eine Meldung sieht und das Programm nicht mit einem Fehler beendet wird (*Ausnahme1.py*):

```
try:
 n = input("Bitte eine Ganzzahl (integer) eingeben: ")
 print(int(n) * 4)
except ValueError:
 print("Das war keine Zahl! Bitte nochmals versuchen ...")
print("Programmende")
```

Wenn das Programm gestartet wird, werden die Anweisungen des *try*-Blocks nacheinander ausgeführt. Falls keine Ausnahme während der Ausführung auftritt, wird das Ende im *try*-Block erreicht und der Programmfluss springt hinter den *try-except*-Block. Der *except*-Block wird übersprungen (Abb. 12.1.4).

Wenn jedoch eine Ausnahme auftritt, wird sofort der *try*-Block verlassen und der *except*-Block wird ausgeführt (Abb. 12.3). Aber auch in dem Fall wird das Programm **nicht** beendet und mit den Anweisungen nach dem *try-except*-Block weitergemacht.

Beachten Sie, dass in dem Beispiel explizit der Typ der Ausnahme – in unserem Fall *ValueError* – angegeben wurde und die erzeugte Ausnahme mit dem Typ übereinstimmen muss. Wir werden noch sehen, wie man dieses Verfahren erweitert.

```
Bitte eine Ganzzahl (integer) eingeben: Eins
Traceback (most recent call last):
 File "F:/PythonQuellcodes/kap12/OhneAbsicherung.py", line 2, in <module>
 print(int(n) * 4)
ValueError: invalid literal for int() with base 10: 'Eins'
>>>
```

**Abb. 12.2** Eine unpassende Eingabe des Anwenders

**Abb. 12.3** Die Ausnahme wurde abgefangen

```
Bitte eine Ganzzahl (integer) eingeben: Eins
Das war keine Zahl! Bitte nochmals versuchen ...
Programmende
>>> |
```

**Abb. 12.4** Es gab keine Ausnahme

```
Bitte eine Ganzzahl (integer) eingeben: 42
168
Programmende
>>> |
```

### 12.4.2 Mehrere Ausnahmeblöcke

Zu einem *try*-Block können **mehrere** *except*-Blöcke gehören. Aber höchstens einer der Blöcke kann ausgeführt werden. Der Block mit der **ersten** Übereinstimmung wird beim Auftreten einer Ausnahme ausgewählt. Dabei gibt es zwei Notationen, die man wählen kann.

- Notation aller *except*-Blöcke hintereinander. Das erlaubt eine qualifizierte Behandlung jeder einzelnen Situation.
- Eine einzelne *except*-Anweisung kann auch gleichzeitig mehrere Fehler abfangen. Die verschiedenen Fehlerarten werden dann in einem **Tupel** (also in runden Klammern) gelistet. Damit behandelt man mehrere Ausnahmesituationen gemeinsam.

### 12.4.3 Die **finally**-Anweisung

Man kann *try*-Anweisungen in Verbindung mit *except*-Klauseln alleine benutzen. Aber es gibt noch eine andere Möglichkeit für *try*-Anweisungen. Der *try*-Anweisung kann eine *finally*-Klausel folgen. Man bezeichnet sie auch als Finalisierungs- oder Terminierungsaktionen, weil sie immer unter allen Umständen ausgeführt werden müssen.

Diese Finalisierungsaktion wird jedoch in der Literatur ab und zu falsch oder zumindest ungenau erklärt. Oft findet man die Erklärung, dass diese Anweisung unabhängig davon ausgeführt wird, ob eine Ausnahme im *try*-Block aufgetreten ist oder nicht. Das stimmt, verschleiert aber die wirkliche Bedeutung von *finally*. Die hinter dem letzten *except*-Block optionale *finally*-Anweisung erlaubt durchaus die Abwicklung wichtiger Abläufe, bevor die Ausführung des gesamten *try-except-finally*-Blocks unterbrochen wird. Unabhängig davon, ob innerhalb des *try*-Blocks eine Ausnahme auftritt oder nicht, werden die Anweisungen in dem Block *finally* ausgeführt. Tritt eine Ausnahme auf, so wird der jeweilige *except*-Block ausgeführt und im Anschluss daran erst der Block hinter der *finally*-Anweisung.

Nun sollte aus den bisherigen Ausführungen und dem letzten Beispiel aufgefallen sein, dass Anweisungen hinter der gesamten *try-except*-Struktur auch auf jeden Fall ausgeführt werden, sowohl wenn eine Ausnahme aufgetreten und behandelt wurde, als auch wenn keine Ausnahme aufgetreten ist. Wozu dann die *finally*-Klausel? Man kann doch alle unbedingt notwendigen Anweisungen einfach hinter die *try-except*-Struktur schreiben.

Der einzige wirklich relevante Grund ist, dass beim Auslösen einer **Sprunganweisung** (wie *break*) innerhalb von *try-except finally* immer noch ausgeführt wird, bevor eventuell eine umgebende Schleife verlassen wird. Also hat diese Klausel immer dann ihre Bedeutung, wenn der Programmfluss umgeleitet wird und bestimmte Schritte vor der Umleitung unumgänglich sind.

### 12.4.4 Praktische Beispiele

Belegen wir nun die letzten Aussagen zur Ausnahmebehandlung mit Beispielen. Beginnen wir mit der Notation mehrerer *except*-Blöcke und der *finally*-Anweisung (*Ausnahme2.py*):

```
while True:
 try:
 x = float(input("Bitte eine Zahl eingeben:"))
 inverse = 1.0 / x
 print(inverse)
 except ValueError:
 print ("Entweder einen int oder einen float eingeben")
 except ZeroDivisionError:
 print("Infinity")
 finally:
 print("Das ist das Finale")
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q":
 break
print("Und nun ist das Programm zu Ende")
```

Sie sehen, dass in dem Beispiel mehrere *except*-Blöcke untereinander notiert werden. Je nach aufgetretener Ausnahme wird der erste passende Block ausgewählt (Abb. 12.5). Das erlaubt die qualifizierte Reaktion je nach Ausnahmetyp. Wenn etwa keine Zahl eingegeben wird, wird eine Ausnahme vom Typ *ValueError* geworfen und aufgefangen. Wird versucht, durch 0 zu teilen, kommt eine Ausnahme vom Typ *ZeroDivisionError*. Der Anwender kann dabei immer wieder neue Eingaben vornehmen, da mit einer Schleife die Eingabe immer wieder aufgerufen wird, bis der Anwender das Zeichen *q* auf Nachfrage eingibt. Erst dann wird das Programm beendet.

**Abb. 12.5** Qualifizierte Reaktion je nach Ausnahmetyp

```
Bitte eine Zahl eingeben:Eins
Entweder einen int oder einen float eingeben
Das ist das Finale
Ende mit q - weiter mit jeder anderen Taste:
Bitte eine Zahl eingeben:0
Infinity
Das ist das Finale
Ende mit q - weiter mit jeder anderen Taste:
Bitte eine Zahl eingeben:5
0.2
Das ist das Finale
Ende mit q - weiter mit jeder anderen Taste: q
Und nun ist das Programm zu Ende
>>> |
```

Jetzt soll ein Tupel mit mehreren Ausnahmen gemeinsam behandelt werden (*Ausnahme3.py*):

```
while True:
 try:
 x = float(input("Bitte eine Zahl eingeben:"))
 inverse = 1.0 / x
 print(inverse)
 except (ValueError,ZeroDivisionError):
 print ("Entweder einen int oder einen float eingeben.",
 "Aber keine 0.")
 finally:
 print("Das ist das Finale")
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q":
 break
print("Und nun ist das Programm zu Ende")
```

Hier sehen Sie, wie ein Tupel mit Ausnahmetypen für eine gemeinsame Behandlung notiert wird. Das erzwingt aber eine gemeinsame Behandlung bei mehreren Ausnahmetypen.

Nun schauen wir uns noch einmal eine konservative Behandlung von potenziellen Fehlern an, bei der ein Problem – die Division durch den Wert 0 – bekannt ist und ohne Ausnahmebehandlung bereits im Vorfeld abgefangen wird (*Ausnahme4.py*):

```
while True:
 try:
 x = float(input("Bitte eine Zahl eingeben:"))
 if x==0:
 print ("Keine 0 eingeben.")
 continue
 inverse = 1.0 / x
 print(inverse)
 except (ValueError):
 print ("Entweder einen int oder einen float eingeben.")
 finally:
 print("Finale.")
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q":
 break
print("Ende")
```

**Abb. 12.6** Trotz break wird finally ausgeführt

```
Bitte eine Zahl eingeben:0
Keine 0 eingeben,
Das ist das Finale - trotz break.
Und nun ist das Programm zu Ende
>>> |
```

Wenn der Anwender eine 0 eingibt, wird die „gefährliche“ Division gar nicht durchgeführt. Das ist bedeutend besser für einen gradlinigen Programmfluss und den Ressourcenverbrauch.

Abschließend soll die Bedeutung von *finally* noch bewiesen werden (*Ausnahme5.py*):

```
while True:
 try:
 x = float(input("Bitte eine Zahl eingeben:"))
 if x==0:
 print ("Keine 0 eingeben,")
 break
 inverse = 1.0 / x
 print(inverse)
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q":
 break
 except (ValueError):
 print ("Entweder einen int oder einen float eingeben.")
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q":
 break
 finally:
 print("Das ist das Finale - trotz break.")
 print ("Auf zum nächsten Schleifendurchlauf")
print("Und nun ist das Programm zu Ende")
```

In dem Beispiel wird die wirkliche Bedeutung von *finally* deutlich. Obwohl bei der Eingabe eine 0 eine *break*-Anweisung ausführt und damit die Schleife verlassen wird, wird dennoch der *finally*-Block vorher ausgeführt. Die letzte Anweisung in der Schleife aber explizit **nicht** (Abb. 12.6)!

---

## 12.5 Standard Exceptions

Es gibt in Python eine Reihe von Standardausnahmen, die nachfolgend kurz aufgelistet werden sollen:

- BaseException
- Exception
- ArithmeticError
- LookupError
- AssertionError
- AttributeError
- EOFError
- EnvironmentError
- FloatingPointError
- IOError
- ImportError
- IndexError
- KeyError
- KeyboardInterrupt
- MemoryError
- NameError
- NotImplemented
- OSError
- OverflowError
- ReferenceError
- RuntimeError
- SyntaxError
- SystemError
- SystemExit
- TypeError
- ValueError
- WindowsError
- ZeroDivisionError

Für die Details sei auf die Dokumentation von Python verwiesen. Aber zwei Ausnahmeklassen sollen hervorgehoben werden. Eine Ausnahme ist in Python eine Instanz der Klasse *BaseException* oder eine Instanz einer der Subklassen von *BaseException*. Das bedeutet, dass alle Ausnahmeklassen auf diese Klasse zurückgehen. Die wichtigste Subklasse ist *Exception*, von der wiederum diverse weitere Ausnahmen erben.

### 12.5.1 Die Reihenfolge bei mehreren Ausnahmetypen

Beachten Sie bei der Notation von mehreren *except*-Blöcken die Reihenfolge. Es wird immer der erste *except*-Block ausgewählt, der zutrifft. Wenn Sie also Ausnahmen vom Typ einer Superklasse oberhalb von dem einer Subklasse notieren (in dem folgenden Fall

*BaseException*), wird der Block der Subklasse niemals ausgelöst. Man erzeugt also unerreichbaren Code ("unreachable code").

```
while True:
 try:
 x = float(input("Bitte eine Zahl eingeben:"))
 inverse = 1.0 / x
 print(inverse)
 except (BaseException):
 print ("Entweder einen int oder einen float eingeben")
 except (ZeroDivisionError):
 print ("Keine 0 eingeben")
 finally:
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q": break
print("Und nun ist das Programm zu Ende")
```

Der Zweig *except (ZeroDivisionError)* wird niemals erreicht, weil *BaseException* die Superklasse von *ZeroDivisionError* ist und deshalb immer dieser Zweig ausgelöst wird, wenn eine Division durch 0 ausgeführt wird. Diese Konstruktion wird der Python-Interpreter nicht akzeptieren.

---

## 12.6 Der else-Block

Die *try ... except*-Anweisung hat in Python noch eine optionale *else*-Klausel. Ein *else*-Block wird ausgeführt, falls keine Ausnahme im *try*-Block auftritt.

- ▶ Ein *else*-Block muss immer **hinter** allen *except*-Anweisungen, aber **vor** *finally* positioniert werden.

Beispiel (*Ausnahme7.py*):

```
while True:
 try:
 x = float(input("Bitte eine Zahl eingeben:"))
 if x==0:
 print ("Keine 0 eingeben.")
 break
 inverse = 1.0 / x
 print(inverse)
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q":
```

```
Bitte eine Zahl eingeben:4
0.25
Ende mit q - weiter mit jeder anderen Taste:
Es ist keine Ausnahme aufgetreten.
Das ist das Finale
Auf zum nächsten Schleifendurchlauf
Bitte eine Zahl eingeben:0
Keine 0 eingeben.
Das ist das Finale
Und nun ist das Programm zu Ende
>>> |
```

Abb. 12.7 Keine Ausnahme führt zur Verzweigung in den else-Zweig

```
break
except (ValueError):
 print ("Entweder einen int oder einen float eingeben.")
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q":
 break
else:
 print("Es ist keine Ausnahme aufgetreten.")
finally:
 print("Das ist das Finale")
print ("Auf zum nächsten Schleifendurchlauf")
print("Und nun ist das Programm zu Ende")
```

Wenn keine Ausnahme ausgelöst wird, wird der Code vom *else*-Zweig ausgeführt (Abb. 12.7).

---

## 12.7 Ausnahmeobjekte auswerten

Eine Ausnahme ist – wie schon erwähnt – ein Mitteilungsobjekt. Wenn Sie an dessen Informationen kommen wollen, können Sie es mit der *as*-Anweisung bei *except* spezifizieren und dann auswerten. Schematisch geht das etwa so:

```
try:

except sqlite3.Error as e:
 print e.args[0]
```

```

Bitte eine Zahl eingeben:Eins
could not convert string to float: 'Eins'
Ende mit q - weiter mit jeder anderen Taste:
Bitte eine Zahl eingeben:0
float division by zero
Ende mit q - weiter mit jeder anderen Taste:
Bitte eine Zahl eingeben:5
0.2
Ende mit q - weiter mit jeder anderen Taste: q
Und nun ist das Programm zu Ende
>>> |

```

**Abb. 12.8** Man kann die Mitteilungen des Exception-Objekts auswerten

In `args[0]` steht standardmäßig die Meldung, welche das Mitteilungsobjekt enthält (Abb. 12.8). Schauen wir uns ein vollständiges Beispiel an.

```

while True:
 try:
 x = float(input("Bitte eine Zahl eingeben:"))
 inverse = 1.0 / x
 print(inverse)
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q":
 break
 except (ValueError,ZeroDivisionError) as e:
 print(e.args[0])
 if input("Ende mit q - weiter mit jeder anderen Taste:
")=="q":
 break
print("Und nun ist das Programm zu Ende")

```

---

## 12.8 Werfen von Ausnahmen mit `raise`

Sie können in Python auch selbst eine Ausnahme erzeugen und werfen – mit der Anweisung `raise`. Das ist eine Sprunganweisung mit Rückgabewert – eben das Ausnahmobjekt. Verwandt ist die Situation mit einer `return`-Ausweisung.

Beispiel (*Ausnahme8.py*):

```
i = 0
def werfen():
 global i
 i+=1
 if i > 3: raise
while True:
 try:
 werfen()
 print(i)
 except :
 print ("Ausnahme")
 break
print("Und nun ist das Programm zu Ende")
```

In der Funktion *werfen()* wird eine Variable erhöht, und wenn diese den Wert 3 erreicht hat, wird die Ausnahme geworfen. In einer Endlosschleife wird die Funktion aufgerufen, nachdem drei Ausgaben der Variable erfolgt sind, wird die Schleife durch die Ausnahme verlassen.

Man kann auch gezielt einen bestimmten Ausnahmetyp werfen.

Beispiel (*Ausnahme9.py*):

```
i = 0
def werfen():
 global i
 i+=1
 if i > 3: raise (IOError())
while True:
 try:
 werfen()
 print(i)
 except TypeError:
 print ("Ausnahme")
 break
print("Und nun ist das Programm zu Ende")
```

Hier wird *IOError* nicht richtig abgefangen (es wird ja nur *TypeError* angegeben) und es kommt zum *traceback*-Fehler. Das wird in dem Beispiel aber nur aus didaktischen Gründen so gemacht und sollte natürlich in der Praxis korrekt zusammenpassen.

## 12.9 Eigene Ausnahmeklassen definieren

Sie können in Python sogar eigene Ausnahmeklassen definieren, einfach über Vererbung von einer der Standardausnahmeklassen von Python.

Beispiel (*Ausnahme10.py*):

```
i = 0
class MeineAusnahme(Exception):
 pass
def werfen():
 global i
 i+=1
 if i > 3: raise MeineAusnahme()
while True:
 try:
 werfen()
 print(i)
 except MeineAusnahme:
 print ("Ausnahme")
 break
print("Und nun ist das Programm zu Ende")
```

---

## 12.10 Die assert-Anweisung

Die *assert*-Anweisung ist in der Regel im Rahmen des sogenannten **Unit-Testing** üblich. Man trifft damit eine Annahme, und wenn diese eintritt, wird eine Maßnahme durchgeführt. Sie kann als abgekürzte Schreibweise für eine bedingte *raise*-Anweisung angesehen werden, das heißt, eine Ausnahme wird nur dann generiert, wenn eine bestimmte Bedingung nicht wahr ist.

Schematisch geht das so:

---

### Beispiel

*assert <Test>, <Nachricht>* ◀

Die obige Zeile kann wie folgt verstanden werden: Falls der Test als *False* ausgewertet wird, wird eine Ausnahme generiert und die Nachricht wird ausgegeben. Beispiel:

```
x = 5
y = 3
assert x < y, "x muss kleiner als y sein"
```

Die Ausnahme ist vom Typ *AssertionError*.



# String-Verarbeitung in Python – Programmierte Textverarbeitung

13

## 13.1 Was behandeln wir in diesem Kapitel?

Der Umgang mit Zeichenketten beziehungsweise Strings beschränkt sich nicht nur darauf, dass man Strings verketten kann. In nahezu jeder modernen Programmiersprache gibt es eine Vielzahl an Techniken zur Manipulation und Auswertung von Strings. Wir haben die seltsame Situation vorliegen, dass die „gewöhnliche“ String-Verarbeitung sicher zu den meistgenutzten und wichtigsten Techniken bei der Programmierung überhaupt gehört, aber in den meisten Facetten<sup>1</sup> auch zu den denkbar einfachsten Routinen. Es gibt daher Bücher, die der einfachen String-Verarbeitung eine Vielzahl an Seiten widmen, und dafür gibt es auch gute Gründe. Denn Computerlaien und blutigen Anfängern muss man durchaus die Möglichkeiten vorstellen, welche eine „Textverarbeitung“ grundsätzlich und erst recht aus der Programmierebene heraus bietet. Ich neige jedoch zu der Einschätzung, dass sich Anwender mit Programmierwissen gerade den Umgang mit Strings sehr gut intuitiv selbst aneignen können. Deshalb möchte ich das Thema sogar sehr knapp halten und nur die wichtigsten Dinge kurz vorstellen. Es soll zudem ausdrücklich auf die Dokumentation von Python verwiesen werden, in der jede noch so kleine Spezialität beim Umgang mit Strings beschrieben wird. Allerdings werden wir ausführlicher auf sogenannte reguläre Ausdrücke im Zusammenhang mit der String-Verarbeitung eingehen, denn das Thema ist dann weder trivial noch wirklich intuitiv.

<sup>1</sup> Sicher mit Ausnahme des Umgangs mit regulären Ausdrücken.

## 13.2 Typische String-Verarbeitungstechniken

Da Strings in Python ja sequenzielle Datenstrukturen (und sowieso Objekte) sind, greifen erst einmal alle Techniken, die man beispielsweise von Tupeln oder Listen kennt, etwa das Iterieren über Strings, die Auswahl einzelner Zeichen oder das Sortieren. Das soll nicht weiter wiederholt werden.

Aber man kann in Python natürlich auch bestimmte Zeichen ersetzen, auf vielfältige Weise Dinge suchen und vergleichen oder Texte konvertieren. Sie können auch Zeichen extrahieren, Vergleiche durchführen oder Teile von Zeichenketten ersetzen, löschen oder austauschen oder auch Strings in andere sequenzielle Datenstrukturen aufspalten. Auch reguläre Ausdrücke werden wie erwähnt unterstützt, wozu wir gleich kommen (Abschn. 13.4).

- ▶ Strings in Python sind immutable und können nach der Erzeugung nicht mehr geändert werden. Das führt dazu, dass jede String-Veränderung einen neuen String generiert. String-Funktionen und -Methoden, die einen String manipulieren, liefern diesen dann als Rückgabewert.

---

## 13.3 Das konkrete Vorgehen in Python

Bei der konkreten String-Verarbeitung bietet Python mehrere Ansätze. Hier soll nur ein kurzer Überblick erfolgen. Details sind wie gesagt in der Dokumentation ausführlich beschrieben, und es wird zudem vorausgesetzt, dass einem Programmierer die grundsätzliche String-Verarbeitung bekannt ist.

### 13.3.1 String-Konstanten und die Format Specification Mini-Language

Es gibt in Python beispielsweise einige Konstanten, die bei der String-Verarbeitung eingesetzt werden (Tab. 13.1).

**Tab. 13.1** Beispiele für String-Konstanten

| Konstante                           | Bedeutung                                                                             |
|-------------------------------------|---------------------------------------------------------------------------------------|
| <code>string.ascii_lowercase</code> | Ein String mit allen Kleinbuchstaben der ASCII-Tabelle ('abcdefghijklmnopqrstuvwxyz') |
| <code>string.ascii_uppercase</code> | Ein String mit allen Großbuchstaben der ASCII-Tabelle ('ABCDEFGHIJKLMNOPQRSTUVWXYZ'). |
| <code>string.digits</code>          | Ein String mit allen Zahlen ('0123456789').                                           |
| <code>string.hexdigits</code>       | Ein String mit allen hexadezimalen Zahlen '0123456789abcdefABCDEF'.                   |

Neben diesen Konstanten gibt es eine spezielle „Sprache“ mit einer Formatierungssyntax, die nur kurz erwähnt werden soll.

### 13.3.2 String-Funktionen

Für kaum ein Gebiet der Programmierung gibt es in Python so viele Built-in Functions wie für die String-Verarbeitung. Wichtig sind etwa *format()*, um Strings auf vielfältige Weise zu formatieren – etwa mit der sogenannten Format Specification Mini-Language –, und natürlich die Funktion *len()*, die bei allen abzählbaren sequenziellen Strukturen die Anzahl der Elemente liefert.

Viele Funktionen sind aber mittlerweile als deprecated (engl. für veraltet, überholt) erklärt worden. Die moderne Art der String-Verarbeitung setzt auch mit wenigen Ausnahmen grundsätzlich auf String-Methoden. Deshalb verfolgen wir nur diese Methoden etwas genauer.

### 13.3.3 String-Methoden

Alle Strings stellen auch unter Python als Objekte über die Punktnotation die typischen Methoden bereit, die man von der String-Verarbeitung kennt. Man kann sie direkt auf das String-Literal oder eine Variable mit einem String als Inhalt anwenden. Tab. 13.2 zeigt nur eine kleine Auswahl.

**Tab. 13.2** Wichtige String-Methoden

| Methode                           | Beschreibung                                                                                                                                                                                                                                                                   |
|-----------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>capitalize()</i>               | Erster Buchstabe großgeschrieben.                                                                                                                                                                                                                                              |
| <i>count(Str)</i>                 | Häufigkeit des Wortes str.                                                                                                                                                                                                                                                     |
| <i>find(Str[, start[, ende]])</i> | Suchen des Strings.                                                                                                                                                                                                                                                            |
| <i>index(Str)</i>                 | Die Position des ersten Vorkommens eines Strings.                                                                                                                                                                                                                              |
| <i>isdecimal()</i>                | Gibt True zurück, wenn Zeichenfolge dezimal ist.                                                                                                                                                                                                                               |
| <i>isdigit()</i>                  | Gibt True zurück, wenn String nur Ziffern enthält.                                                                                                                                                                                                                             |
| <i>join(iterable)</i>             | Verbindet einen sequenziellen Datentyp zu einem String.                                                                                                                                                                                                                        |
| <i>lower()</i>                    | Retouren-Zeichenfolge in Kleinbuchstaben.                                                                                                                                                                                                                                      |
| <i>replace (alt, neu)</i>         | Das erste Argument ersetzt mit dem zweiten Argument.                                                                                                                                                                                                                           |
| <i>split([sep[, maxsplit]])</i>   | Gib eine Liste der Wörter in der Zeichenfolge zurück, wobei sep als Trennzeichen verwendet wird. Wenn maxsplit gegeben ist, werden höchstens maxsplit Splits durchgeführt. Wenn maxsplit nicht angegeben ist oder -1, dann gibt es keine Begrenzung auf die Anzahl der Splits. |
| <i>upper()</i>                    | Retouren-Zeichenfolge in Großbuchstaben.                                                                                                                                                                                                                                       |

Schauen wir uns ein kleines Beispiel an (*String1.py*).

```
t1 = "Ein Kratzer?! Ihr Arm ist ab!"
t2 = "Es ist lustig, nicht wahr? Wie Ihr bester Freund einfach so explodieren kann?"
t3 = "Dieser Papagei ist nicht mehr."
t4 = "Und jetzt zu etwas vollkommen anderem."
t5 = "42"
print(t1.upper())
print(t1.lower())
print("nein, das stimmt nicht.".capitalize())
print("Die Länge von t2.",
 "Hier einmal keine String-Methode, sondern eine String-Funktion():",
 len(t2))
print("Stehen in t5 nur Zahlen?", t5.isdigit())
print(t3.index("ist"))
print(t2.count("e"))
print(t4.replace("e", "i"))
print(t4.split(" "))
```

Das ist die Ausgabe:

*EIN KRATZER?! IHR ARM IST AB!*

*ein kratzer?! ihr arm ist ab!*

*Nein, das stimmt nicht.*

*Die Länge von t2. Hier einmal keine String-Methode, sondern eine String-Funktion(): 77*

*Stehen in t5 nur Zahlen? True*

*15*

*8*

*Und jetzt zu etwas vollkommen andirim.*

*['Und', 'jetzt', 'zu', 'etwas', 'vollkommen', 'anderem.']}*

Sie erkennen sicherlich die nahezu triviale Anwendung der String-Verarbeitungsfunktionen und -methoden, und ich möchte die Ausführungen dazu deshalb beenden.

---

## 13.4 Umgang mit regulären Ausdrücken

Das aus meiner Sicht einzig wirklich zu erklärende Thema im Zusammenhang mit String-Verarbeitung ist wie erwähnt der Umgang mit regulären Ausdrücken. Denn diese sind wie gesagt alles andere als trivial. Zudem greift man dazu in Python auf ein Modul *re* und darüber bereitgestellte zusätzliche Methoden zurück. Das bedeutet auch, dass wir das erste Mal in diesem Buch explizit auf die Einbindung eines zusätzlichen Moduls aus dem Standard-API von Python zurückgreifen werden.

### 13.4.1 Was sind allgemein reguläre Ausdrücke?

Sie kennen sicher die Möglichkeiten moderner Editoren und Textverarbeitungssysteme, mit denen Sie in einem Dokument nach Textpassagen suchen und diese gegebenenfalls auch ersetzen können. Solche Suchmöglichkeiten in Texten gibt es auch in der Programmierung, wobei dabei meist der Inhalt von Variablen oder eine Benutzereingabe durchsucht werden.

In den meisten Programmiersprachen können Sie mittels geeigneter Funktionen und Methoden in Strings bestimmte Bestandteile suchen und diese bei Bedarf auch ersetzen. Das haben wir ja gerade auch gesehen.

Allerdings kann es komplexere Situationen geben. Diese sind genau zu untersuchen. Als Beispiele seien die folgenden Fälle angeführt. Sie suchen in einem String:

- alle Zahlen, aber nur diese Zeichen,
- nur Kleinbuchstaben,
- die Buchstaben A, C und x,
- alle Sonderzeichen,
- alle Whitespace-Zeichen (Leerzeichen, aber auch Tabulatorzeichen etc.),
- einen beliebigen Teilstring, der aber mindestens fünf Zeichen lang sein muss, oder
- alle deutschen Umlaute.

In jedem dieser Fälle können Sie gar nicht oder nur sehr umständlich konkret ein Zeichen angeben, sondern nur eine gewisse Regel, was genau für Zeichen Sie suchen.

Zum formalen Beschreiben dieser Regeln gibt es die **regulären Ausdrücke** ("regular expressions"), über die Sie **Suchmuster** (Pattern) statt einzelner Zeichen definieren können, wobei das nicht bedeutet, dass man bei regulären Ausdrücken nicht auch direkt Zeichen wie konkrete Buchstaben, Zahlen oder einige Sonderzeichen notieren kann.<sup>2</sup> Diese Zeichen können insbesondere mit abstrakten regulären Formulierungen kombiniert werden.

### 13.4.2 Wo setzt man reguläre Ausdrücke ein?

Reguläre Ausdrücke können direkt bei den Werten von Variablen eingesetzt werden, um diese zu durchsuchen und vor allen Dingen nach gewissen Regeln zu manipulieren. Sie können diese Variablenwerte etwa aufspalten oder gewisse Strukturen darin suchen und gegebenenfalls ersetzen. Die Hauptanwendung von regulären Ausdrücken ist aber der

---

<sup>2</sup>Ein Suchmuster besteht dann aus eben diesem Zeichen oder in dem Suchmuster muss an einer gewissen Stelle genau dieses Zeichen vorkommen. Ein konkretes Zeichen ist also eine Vereinfachung eines allgemeinen Suchmusters.

Einsatz in Verbindung mit freien Benutzereingaben. Diese Plausibilisierung<sup>3</sup> solcher Benutzereingaben zählt in vielen Applikationen zu den ganz wichtigen Vorgängen.

Plausibilisierung bedeutet allgemein, die Schlüssigkeit von Daten und Strukturen zu überprüfen. Das beinhaltet zum Beispiel bei Eingaben eines Anwenders die Überprüfung auf Pflichtfelder<sup>4</sup> in einem Webformular, aber auch die Festlegung einer Mindestlänge von bestimmten Informationen oder die zwingende Existenz gewisser Zeichen bei einer Eingabe oder einem Variablenwert (beispielsweise die zwingende Existenz des @-Zeichens bei einer E-Mail-Adresse) kann bei einer Plausibilisierung überprüft werden.

### 13.4.3 Details zu Pattern

**Suchmuster** (sogenannte Pattern) bezeichnen bei einem regulären Ausdruck zu suchende Ausdrücke, die aus einem Suchtext mit

- konkreten Zeichen,
- Platzhaltersymbolen und
- gewissen Steuerzeichen

bestehen können. Gegebenenfalls kann man einen allgemeinen regulären Ausdruck mit einem sogenannten **Modifikator** hinter dem eigentlichen Suchausdruck genauer spezifizieren. Damit kann man angeben,

- ob nach dem Suchstring so oft gesucht wird, wie er in dem gesamten zu durchsuchenden Bereich vorkommt, oder nur einmal,
- ob Klein- und Großschreibung beachtet wird oder nicht und
- ob in mehreren Zeilen gesucht wird.

Sie können bei Funktionen und Methoden in Programmiersprachen, die mit dem sogenannten Perl-Stil arbeiten, meist diese Modifikatoren verwenden. Andere Methoden und Funktionen stellen solche Variationen manchmal implizit bereit.

Bei der Formulierung eines Patterns verwenden Sie also neben konkreten Zeichen auch Klammern, Steuerzeichen und Optionen, die in einem Ausdruck kombiniert werden können.

- ▶ Sofern keine Widersprüche ausgelöst werden, kann man auch mit dem Pipe-Symbol | als Oder-Verbindung mehrere Ausdrücke in einem Pattern verwenden. Das erlaubt die Angabe von Alternativen.

---

<sup>3</sup>Überprüfung nach gewissen logischen Kriterien.

<sup>4</sup>Felder, die einfach nur mit irgendeinem Inhalt gefüllt sein müssen.

### 13.4.3.1 Eckige und runde Klammern – Brackets

Eine Anwendung eines regulären Ausdrucks besteht darin, dass Sie eine Gruppe an erlaubten Zeichen festlegen.

#### 13.4.3.1.1 Eckige Klammern

In eckigen Klammern folgt dabei eine Auflistung von allen Zeichen, die für **genau ein** konkretes Zeichen an einer Stelle erlaubt sind. Wenn ein zusammenhängender Bereich angegeben werden soll, kann man den Beginn und das Ende des Bereichs mit einem Minuszeichen verbinden.

Insgesamt steht der **gesamte rechteckig geklammerte Ausdruck** immer also nur für genau **ein** Zeichen. Mehrere Paare mit rechteckigen Klammern in einem Suchausdruck entsprechen damit mehreren Zeichen (genauso viele wie Klammernpaare), ebenso mehrere Zeichen oder Steuerzeichen, die nicht in eckigen Klammern notiert werden.

Hier ist ein Beispiel, bei dem als Treffer an der angegebenen Stelle eine Zahl zwischen 1 und 7 stehen darf:

##### Beispiel

[1234567] ◀

Das kann man auch so schreiben:

##### Beispiel

[1-7] ◀

Wenn eine Lücke in einer Sequenz notwendig ist, kann man das so formulieren – auch das **genau ein** Zeichen an:

##### Beispiel

[1-37-9] ◀

An der so spezifizierten Stelle kann eine Zahl zwischen 1 und 3 oder 7 bis 9 stehen. So etwas kann man auch für Buchstaben formulieren, wobei die Buchstaben dann entsprechend der alphabetischen Sortierung zu verstehen sind:<sup>5</sup>

##### Beispiel

[e-r] ◀

<sup>5</sup>Genau genommen ist die Position im ASCII-Code relevant.

Die verschiedenen Ziffern und Buchstaben können in einem Suchausdruck beliebig kombiniert werden, und auch manche Sonderzeichen sind erlaubt. Dies wäre ein Beispiel für einen zweistelligen Ausdruck, bei dem

- für beide Stellen Ziffern und Kleinbuchstaben von *a* bis *f* als Treffer erlaubt sind und
- zusätzlich für das erste Zeichen noch der Unterstrich erlaubt ist:

---

#### Beispiel

[\_0-9a-f][0-9a-f] ◀

##### 13.4.3.1.2 Runde Klammern

Wenn Sie eine Auswahl in **runden Klammern** notieren und dazwischen das | -Zeichen setzen, dann geben Sie eine Oder-Beziehung an. Aber runde Klammern können auch einfach Zeichen zusammenfassen, die mit weiteren Angaben bei Bedarf als Gruppe weiter konkretisiert werden.

Das nachfolgende Pattern legt fest, dass **entweder**

- ein zweistelliger Ausdruck gültig ist, bei dem
  - das erste Zeichen x, y, oder z und
  - das zweite Zeichen eine Zahl zwischen 1 und 5 ist

**oder**

- ein dreistelliger Ausdruck ist gültig, bei dem
  - das erste eine Zahl 1, 2 oder 3,
  - das zweite Zeichen ein Kleinbuchstabe zwischen a und u und
  - das dritte Zeichen die 0 ist:

---

#### Beispiel

([xyz][1-5]][[123][a-u]0) ◀

Beachten Sie, dass bei vielen Methoden und Funktionen in Python bereits die einfache Notation eines Ausdrucks in runden Klammern für einen regulären Ausdruck steht.

##### 13.4.3.2 Steuerzeichen

Bei einer regulären Suche ist es oft von Belang, **wo** genau sich ein Suchmuster befindet, zum Beispiel zu Beginn einer Zeichenkette, am Ende oder an einer Wortgrenze. Dazu kann man bestimmte Zeichen wie den Slash nicht direkt suchen, denn sie haben in regulären Ausdrücken teils eine spezielle Bedeutung. Wenn man diese suchen möchte, muss man sie verschlüsselt eingeben. Ebenso kann man bestimmte Arten von Zeichen (etwa nur Ziffern oder nur Buchstaben) durch spezielle Codes ausdrücken.

- Das Verschlüsseln von Sonderzeichen durch eine besondere Syntax nennt man das **Maskieren** von Zeichen.

Tabelle 13.3 zeigt erst einmal eine Auswahl wichtiger allgemeiner Steuerzeichen bei regulären Ausdrücken nach dem Perl-Stil. Diese werden auch Meta-Zeichen (**Metacharacters**) genannt, weil sie Zeichen eine spezielle Zusatzbedeutung geben. Sie werden allerdings nicht in allen Programmiersprachen vollständig unterstützt. Insbesondere werden bereits spezialisierte Funktionen und Methoden diese oft nicht unterstützen.

#### 13.4.4 Optionen für die Häufigkeit

Bei regulären Ausdrücken ist es ganz wichtig, dass man auch eine **Anzahl** angeben kann, wie oft ein bestimmtes Zeichen vorkommen muss oder darf. Hier sehen Sie die wichtigsten Optionen beziehungsweise **Quantifizierer** (Quantifiers) für die Formulierung von Suchpattern, die den Suchausdruck über die Zeichen hinaus hinsichtlich der Anzahl nach dem Perl-Stil genauer beschreiben können (Tab. 13.4). Dabei muss aber wieder beachtet werden, dass nicht alle Methoden und Funktionen diese unterstützen und manchmal auch gar nicht benötigen.

- Wenn bei einem regulären Ausdruck kein Zeichen für eine Option notiert wird, steht das also für ein zwingendes Zeichen, das genau einmal vorkommt.

#### 13.4.5 Die Umsetzung von regulären Ausdrücken in Python – das Modul `re`

Prinzipiell werden reguläre Ausdrücke in Python als Strings dargestellt. Der eigentliche reguläre Ausdruck (also alles außer „normalen“ Zeichen) in dem Pattern wird dabei mit einem Backslash (also über ein Escape-Zeichen) eingeleitet. Das bedeutet aber, dass Backslashes normalerweise aus dem regulären Ausdruck entfernt beziehungsweise mit dem folgenden Zeichen einer Sonderbedeutung zugeführt werden. Man kann dies verhindern, indem man jeden Backslash eines regulären Ausdrucks als "\\" schreibt. Allerdings verhält sich Python mit den Methoden des `re`-Moduls leider nicht ganz konsistent, und manchmal kann man auf die Maskierung des Backslashes verzichten. Einige Methoden und Funktionen gestatten also die Angabe eines einfachen Backslashes, andere brauchen die Maskierung.

##### 13.4.5.1 Raw-Strings

Eine alternative Lösung besteht darin, sogenannte **Raw-Strings** zu verwenden. Dabei wird ein String mit einem vorgestellten `r` markiert, dann werden keine Escape-Sequenzen im String ausgeführt. Diese Notation ist also in der Regel der sicherste Weg und in Python üblich.

**Tab. 13.3** Steuerzeichen für reguläre Ausdrücke zum Suchen in Strings

| Steuerzeichen | Beispiel | Beschreibung                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|---------------|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \$            | in\$     | Die Notation des Dollarzeichens am Ende eines Suchstrings führt zu Treffern in allen Strings, bei denen diese Textpassage am Ende vorkommt, beispielsweise in den Strings "in", "kein" oder "worin", aber nicht in den Strings "innen" oder "alleine", denn hier steht "in" nicht am Ende.                                                                                                                                                                                                                                                                                       |
| .             | .in      | Die Notation eines Punktes fordert an der Stelle vor dem Suchbegriff mindestens ein beliebiges Zeichen. Notieren Sie einen Punkt, muss vor dem Suchbegriff mindestens ein anderes Zeichen in dem durchsuchten Ausdruck stehen. Notieren Sie zwei Punkte, müssen mindestens zwei beliebige Zeichen dort vorhanden sein und so fort.<br>Die Suche nach ...in führt beispielsweise zu Treffern in "alleine" und "worin", aber nicht in den Strings "innen" oder "in".<br>Achtung – wenn Sie explizit einen Punkt an einer bestimmten Stelle fordern, müssen Sie ihn maskieren (\.). |
| \.            | /w\./d   | An der Stelle muss explizit ein Punkt stehen. In dem Beispiel muss an der zweiten Stelle des Patterns ein Punkt stehen.                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| \b            | \bin     | Die Notation von \b (achten Sie auf die Kleinschreibung) sucht den vorangestellten oder den nachfolgenden Suchstring an einer Wortgrenze. Vorangestellt muss der Suchstring am Anfang eines Worts stehen (die Suche nach \bin führt beispielsweise zu Treffern in "innen" und "in", jedoch nicht in den Strings "alleine" oder "worin") und nachgestellt am Ende.                                                                                                                                                                                                                |
| \B            | in\B     | Die Notation von \B (achten Sie auf die Großschreibung) sucht den vorangestellten oder nachfolgenden Suchstring und führt zu einem Treffer, wenn dieser nicht an der entsprechenden Wortgrenze steht. Vorangestellt darf der Suchstring nicht am Anfang eines Worts stehen (die Suche nach \Bin führt beispielsweise zu keinen Treffern in "innen" und "in", jedoch in den Strings "alleine" oder "worin" – an der hinteren Wortgrenze kann der Suchbegriff stehen) und nachgestellt zu keinen Treffern am Ende eines Worts.                                                     |
| \d            | \d       | Die Notation von \d (achten Sie auf die Kleinschreibung) sucht nach einer beliebigen ganzen Zahl. Das ist äquivalent zu der Angabe [0-9].                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| \D            | \D       | Die Notation von \D (achten Sie auf die Großschreibung) sucht nach einem beliebigen Zeichen, das keine Ziffer ist.                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| \f            | \f       | Sie können in regulären Ausdrücken nach den üblichen Maskierungen für Sonderzeichen suchen. Diese Sequenz führt zur Suche nach einem Seitenvorschub.                                                                                                                                                                                                                                                                                                                                                                                                                             |
| \n            | \n       | Eine übliche Maskierung für Sonderzeichen. Diese Sequenz führt zur Suche nach einem Zeilenvorschubzeichen.                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| \r            | \r       | Eine übliche Maskierung für Sonderzeichen. Diese Sequenz führt zur Suche nach einem Wagenrücklaufzeichen.                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |

(Fortsetzung)

**Tab. 13.3** (Fortsetzung)

| Steuerzeichen | Beispiel | Beschreibung                                                                                                                                                                                                                                                                                                       |
|---------------|----------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \s            | \s       | Die Notation von \s (achten Sie auf die Kleinschreibung) führt zu Treffern in allen Strings, in denen irgendeine Art von Whitespace vorkommt. Das sind Leerzeichen und alle maskierten Sequenzen für diese Zeichen (also Seitenvorschub, Zeilenschaltung, Tabulator und vertikaler Tabulator – \f, \n, \t und \v). |
| \S            | \S       | Die Notation von \S (achten Sie auf die Großschreibung) sucht nach einem beliebigen Zeichen, das kein Whitespace ist.                                                                                                                                                                                              |
| \t            | \t       | Eine übliche Maskierung für Sonderzeichen. Diese Sequenz führt zur Suche nach einem Tabulator.                                                                                                                                                                                                                     |
| \v            | \v       | Eine übliche Maskierung für Sonderzeichen. Diese Sequenz führt zur Suche nach einem vertikalen Tabulator.                                                                                                                                                                                                          |
| \w            | \w       | Die Notation von \w (achten Sie auf die Kleinschreibung) führt zu Treffern bei allen alphanumerischen Zeichen und dem Unterstrich.                                                                                                                                                                                 |
| \W            | \W       | Die Notation von \W (achten Sie auf die Großschreibung) führt zu Treffern bei allen Zeichen, die keine alphanumerischen Zeichen und nicht ein Unterstrich sind.                                                                                                                                                    |
| ^             | ^in      | Die Notation des ^-Zeichens am Anfang eines Suchstrings führt zu Treffern in allen Strings, in denen diese Textpassage am Anfang vorkommt, beispielsweise in "in" oder "innen", aber nicht in den Strings "kein", "alleine" oder "worin", denn hier steht "in" nicht am Anfang.                                    |

**Tab. 13.4** Optionen bei regulären Ausdrücken

| Steuerzeichen | Beispiel | Beschreibung                                                                                                                                                                                                                                                                    |
|---------------|----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| *             | I[a-h]*  | Die Notation des Sterns bedeutet, dass das letzte Zeichen vor dem Stern kein Mal oder beliebig oft hintereinander stehen kann.                                                                                                                                                  |
| ?             | [0-9]?   | Ein vorangestelltes Zeichen kann genau einmal vorkommen, muss aber nicht. Also ein optionales Zeichen.                                                                                                                                                                          |
| {min, max}    | a{1,3}b  | Angabe einer <b>Mindestanzahl</b> und einer <b>Maximalanzahl</b> , wie oft ein vorangestelltes Zeichen auftreten muss beziehungsweise kann. In dem Beispiel muss das Zeichen a mindestens einmal vorkommen und darf maximal dreimal vorkommen, bevor ein Zeichen b folgen muss. |
| {min}         | a{3}b    | Angabe einer Mindestanzahl, wie oft ein vorangestelltes Zeichen auftreten muss. Die maximale Anzahl bleibt offen.                                                                                                                                                               |
| +             | .+in     | Das +-Zeichen bedeutet, dass das letzte Zeichen davor für einen Treffer mindestens ein Mal oder beliebig oft dort stehen muss.                                                                                                                                                  |

### 13.4.5.2 Das *re*-Modul

Nun muss man wie gesagt noch ein Modul *re* importieren, um Pattern in passenden Methoden nutzen zu können. Dieses Modul unterstützt eine Syntax, die der in Perl ähnelt. Sowohl das zu durchsuchenden Muster als auch das Pattern können Unicode-Strings sowie 8-Bit-Strings (Bytes) sein. Unicode-Strings und 8-Bit-Strings können jedoch nicht gemischt werden. Das heißt, Sie können keine Unicode-Zeichenfolge mit einem Byte-Muster abgleichen oder umgekehrt. Wenn Sie nach einer Ersetzung fragen, muss die Ersetzungszeichenfolge vom gleichen Typ sein wie das Muster und die Suchzeichenfolge.

#### ► Tipp

1. Die meisten regulären Ausdrücke stehen in kompilierter Form zur Verfügung, sowohl über Funktionen als auch Methoden. Bei den Funktionen handelt es sich jedoch um Verknüpfungen, bei denen Sie zwar nicht zuerst ein Regex-Objekt kompilieren müssen, aber einige der oben schon genannten Feinabstimmungsparameter nicht zur Verfügung haben.
2. Das *Regex*-Modul, das für Python von einem Drittanbieter bereitgestellt wird, hat ein zu dem *re*-Modul kompatibles API, bietet jedoch zusätzliche Funktionen und eine gründlichere Unicode-Unterstützung.

Das Modul *re* stellt nun eine ganze Reihe an Methoden und Funktionen bereit, von der hier eine Auswahl notiert wird (Tab. 13.5).

### 13.4.6 Die Match-Objekte

Als Treffer bei Suchen mit *match()* und *search()* erhalten Sie Match-Objekte (Übereinstimmungsobjekte). Diese haben immer einen booleschen Wert von *True*, wenn es denn einen Treffer gab. Da *match()* und *search()* *None* zurückgeben, wenn es keine Übereinstimmung gibt, können Sie mit einer einfachen *if*-Anweisung testen, ob es eine Übereinstimmung gab, etwa so:

---

#### Beispiel

```
match = re.search(Muster, Zeichenfolge)
if match:
 ...
 ▶
```

Übereinstimmungsobjekte unterstützen aber auch eine Reihe von Methoden und Attributen.

- Besonders wichtig ist die Methode *group()*, denn diese gibt eine oder mehrere Untergruppen der Übereinstimmung zurück. Wenn es nur ein einziges Argument gibt, ist das Ergebnis eine einzelne Zeichenfolge; wenn es mehrere Argumente gibt, ist das Ergebnis

**Tab. 13.5** Methoden des Moduls re

| Methode                                                             | Beschreibung                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>re.compile (Muster, Flags = 0)</code>                         | Mit der Methode kompilieren Sie ein Pattern in ein reguläres Ausdrucksobjekt, das mit den Methoden <code>match()</code> , <code>search()</code> und anderen Methoden zum Abgleich verwendet werden kann. Das Verhalten des Ausdrucks kann geändert werden, indem ein Flag-Wert angegeben wird, der mit dem bitweisem ODER (der Operator <code> </code> ) kombiniert werden kann.<br>Die Flags <code>re.I</code> oder <code>re.IGNORECASE</code> sorgen etwa dafür, dass Groß- und Kleinschreibung ignoriert wird, oder <code>re.M</code> beziehungsweise <code>re.MULTILINE</code> sorgen dafür, dass gegebenenfalls über mehrere Zeilen hinweg gesucht wird. Die Verwendung von <code>re.compile()</code> und das Speichern des resultierenden regulären Ausdrucksobjekts zur Wiederverwendung ist effizienter als das direkte Anwenden von <code>match()</code> oder <code>search()</code> , wenn der Ausdruck in einem Programm mehrmals verwendet wird. |
| <code>re.findall (Muster, String, Flags = 0)</code>                 | Die Methode liefert alle nicht überlappenden Übereinstimmungen des Musters in der Zeichenkette als Liste von Zeichenketten zurück. Die Zeichenfolge wird von links nach rechts gescannt, und Übereinstimmungen werden in der gefundenen Reihenfolge zurückgegeben. Wenn eine oder mehrere Gruppen im Muster vorhanden sind, erhalten Sie eine Liste von Gruppen.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>re.match (Muster, String, Flags = 0)</code>                   | Wenn null oder mehr Zeichen am Anfang der Zeichenfolge mit dem Pattern übereinstimmen, erhalten Sie ein entsprechendes Match-Objekt (Abschn. 1.4.6). Die Rückgabe ist <code>None</code> , wenn die Zeichenfolge nicht mit dem Muster übereinstimmt.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>re.search(Muster, String, Flags=0))</code>                    | Mit der Methode durchsuchen Sie die Zeichenfolge nach dem ersten Speicherort, an dem das Pattern eine Übereinstimmung erzeugt. Die Rückgabe ist ein Match-Objekt (Abschn. 1.4.6). Die Rückgabe ist <code>None</code> , wenn keine Position in der Zeichenfolge mit dem Muster übereinstimmt.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| <code>re.split (Muster, String, Maxsplit = 0, Flags = 0)</code>     | Mit der Methode teilen Sie die Zeichenfolge nach den Vorkommen des Musters, und zwar in die optional angegebene maximale Anzahl von Aufteilungen. Wenn <code>Maxsplit</code> ungleich null ist, wird ein eventueller Rest der Zeichenfolge als letztes Element der Liste zurückgegeben. Beachten Sie, dass das Trennzeichen selbst nicht mehr in der Ergebnismenge auftaucht.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <code>re.sub (Muster, Ersatz, String, Anzahl = 0, Flags = 0)</code> | Die Methode gibt den String zurück, bei dem die am weitesten links liegenden, nicht überlappenden Vorkommen des Suchmusters durch den Ersatz ausgetauscht wurden.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

ein Tupel mit einem Element pro Argument. Ohne Argumente wird *group()* standardmäßig auf null gesetzt, und die gesamte Übereinstimmung wird zurückgegeben.

- Sehr ähnlich ist die Methode *groups(default=None)*. Sie liefert ein Tupel zurück, das alle Untergruppen des Matches enthält, von 1 bis zu der letzten Gruppe im Muster.
- Mit dem Attribut *lastindex* erhalten Sie den ganzzahligen Index der letzten Übereinstimmung oder *None*, wenn überhaupt keine Gruppe gefunden wurde.

### 13.4.7 Ein paar Beispiele mit regulären Ausdrücken

Kommen wir nun zu einigen Beispielen, die die Anwendung verdeutlichen sollen.

#### 13.4.7.1 Texte aufsplitten

Das erste Beispiel soll die Methode *split()* zeigen und auch explizit die Verwendung von Raw-Strings bei regulären Ausdrücken der von „normalen“ Strings gegenüberstellen (*Reg1.py*).

```
import re
t1 = "Die Antwort ist 42."
print(re.split('\\d', t1))
print(re.split('\W+', "Die Antwort ist 42."))
print(re.split('\\d', "Die Antwort ist 42."))
print(re.split(r"\d", "Die Antwort ist 42."))
print(re.split("\d", "Die Antwort ist 42."))
print(re.split(r"\d", "Die Antwort ist 42.))
```

Das ist die Ausgabe:

```
['Die Antwort ist ', ',', '']
['Die', 'Antwort', 'ist', '42', '']
['Die Antwort ist ', ',', '']
['Die Antwort ist ', ',', '']
['Die Antwort ist ', ',', '']
['Die Antwort ist 42.']]
```

Wir wenden die Methode sowohl auf ein Literal als auch auf eine Variable an. Das geht, wie Sie sehen. Analysieren wir genauer das Ergebnis.

- Die erste Anweisung mit *split()* nutzt das Pattern \d, um den String, der in der Variable t1 gespeichert ist, an allen Ziffern aufzuspalten. Der Backslash ist maskiert. Es gibt zwei Ziffern (4 und 2), deshalb müssen drei Elemente in der resultierenden Liste sein. Allerdings gibt es zwischen der 4 und der 2 kein anderes Zeichen, deshalb ist das zweite Element leer.

- Die zweite Anweisung mit `split()` nutzt das Pattern `\W`, um das Stringliteral an allen Zeichen, die keine alphanumerischen Zeichen und nicht ein Unterstrich sind, aufzuspalten. Das Pluszeichen gibt an, dass so ein Zeichen mindestens einmal, aber auch beliebig oft folgend kann. Mehrere Leerzeichen hintereinander sorgen so nicht für mehrere Trefferelemente. Der Backslash ist wieder maskiert.
- Die dritte Anwendung entspricht der ersten Anweisung mit `split()` – nur auf ein Stringliteral angewendet.
- Die vierte Anweisung mit `split()` verwendet einen Raw-String in einem unmaskierten Backslash. Damit entspricht das Ergebnis exakt dem der ersten und der dritten Anwendung.
- Die fünfte Anweisung mit `split()` verwendet dieses Mal für das Suchmuster die doppelten Hochkommata. Das soll nur deutlich machen, dass es keinen Unterschied macht, ob man das Suchmuster in einfache oder doppelte Hochkommata notiert. Das Ergebnis entspricht damit dem der ersten, dritten und vierten Anwendung.
- Die sechste Anweisung von `split()` verwendet wieder einen Raw-String, in dem aber zwei Backslashes stehen. Aber das ist dann eben keine (!) Maskierung, und der zweite Backslash ist der reguläre Ausdruck. Mit dem gibt es keine Übereinstimmung und deshalb auch keine Aufspaltung.

### 13.4.7.2 Muster suchen und in Match-Objekten bereitstellen

Betrachten wir nun ein Beispiel mit Match-Objekten (*Reg2.py*).

```
import re
t1 = "Die Antwort ist 42."
menge = re.search('\d', t1)
print(menge.group(0))
print(menge.group())
menge = re.match(r"(\d+)\.(\d+)", "3.14")
print(menge.groups())
print(menge.lastindex)
print(menge.group(menge.lastindex))
for i in menge.groups():
 print(i)
```

Das ist die Ausgabe:

```
4
4
('3', '14')
2
14
3
14
```

Wir wenden die Methoden *search()* und *match()* an und betrachten das resultierende Match-Objekt. Analysieren wir auch wieder genauer das Ergebnis.

- Zuerst suchen wir mit *search()* nach der ersten Zahl in einem String. Damit erhalten wir ein Element, das in *group(0)* zur Verfügung steht.
- Die Ausgabe über *group()* führt zu dem gleichen Resultat, weil es eben nur ein Element in der Treffermenge gibt.
- Mit `(\d+)\.( \d+)` suchen wir mit *match()* nach einem Pattern, das mindestens eine Zahl, dann einen Punkt und dann mindestens noch eine Zahl enthält. Die Bestandteile werden über die Methode *groups()* bereitgestellt.
- Die Anzahl der Elemente in der Ergebnismenge wird mit *lastIndex* abgefragt (null-indiziert).
- Mit dem letzten Index kann man auf das letzte Element zugreifen, dass sich in der Treffermenge befindet.
- Mit der *for*-Schleife wird über die Treffermenge iteriert.

#### 13.4.7.3 Kompilierte Suchausdrücke

Kompilieren wir auch einmal einen Suchausdruck (*Reg3.py*):

```
import re
t1 = "45 EUR"
prog = re.compile(r"\d+")
menge = prog.match(t1)
print(menge.group())
```

Damit wird das erste Auftauchen einer Zahl gesucht und diese in dem Match-Objekt gespeichert. Das liefert also 45.



# Datei-, Datenträger- und Datenbankzugriffe – Dauerhafte Daten

14

## 14.1 Was behandeln wir in diesem Kapitel?

In Python Dateien lesen und schreiben ist sehr einfach – wie die meisten Dinge in dieser Sprache. Sie öffnen einfach einen **Datenstrom** zum Lesen oder Schreiben und greifen dann entsprechend darauf zu. Dateizugriffe sind in den Kern von Python integriert. Deshalb ist es nicht notwendig, spezielle Module einzubinden. Wir wollen uns aber in diesem Kapitel auch mit allgemeinen Datei- und Verzeichnisoperationen sowie einem kurzen Einstieg zur Verwendung von Datenbanken unter Python beschäftigen. Diese benötigen dann die Einbindung der passenden Module des Standard-APIs.

## 14.2 Datenströme für die Ein- und Ausgabe

Die Ein- und Ausgabe von Daten basiert in fast allen modernen Programmiersprachen auf sogenannten **Strömen** oder **Streams**. Ein Datenstrom ("data stream") besteht aus einer kontinuierlichen Folge von Daten. Man unterscheidet **Downstreams** (eingehende Datenströme) und **Upstreams** (ausgehende Datenströme). Ein Datenstrom kann von jeder beliebigen Quelle kommen und zu jedem beliebigen Ziel gehen. Das bedeutet, der Ursprungsort und das Ziel spielen für die konkrete Codierung überhaupt keine Rolle. Programmiersprachen bieten eine Vielzahl von unterschiedlichen Strömen, die für verschiedene Zwecke optimiert sind.

Das beginnt bei der Ein- und Ausgabe über Tastatur und Bildschirm und geht über Dateioperationen bis hin zu Ein- und Ausgabe über ein Netzwerk. Es gibt in Python zwei

Datenströme, die vielen zunächst einmal als solche nicht bewusst sind: die Standardeingabe (**stdin**) und die Standardausgabe (**stdout**).

Da Ströme Objekte sind,<sup>1</sup> besitzen sie auch charakteristische Methoden zum Lesen und Schreiben von Daten. Alle Methoden, die sich mit Eingabe- und Ausgabeoperationen beschäftigen, werden in der Regel eine Ausnahme vom Typ *IOError* oder eine davon abgeleitete Ausnahme auswerfen, und diese sollte entsprechend aufgefangen weitergegeben werden.

### Hintergrundinformation

Darüber hinaus kann bei Zugriffen auf Datenströme eine Ausnahme vom Typ *EOFError* wichtig werden. EOF steht für „End of File“. Eine Ausnahme dieses Typs wird in vielen Programmiersprachen geworfen, wenn beim Einlesen einer Datei das **Dateiende** erreicht wird. Man kann damit sehr sauber und stabil bis zum Ende einer Datei lesen und diese dann in einer Ausnahmehandlung schließen.

Allerdings geht Python mit dem Dateiende etwas anders um und wirft beim Erreichen mit den nachfolgend noch beschriebenen Methoden **keine** Ausnahme. Die Ausnahme vom Typ *EOFError* wird nur beim Einlesen von der Standardeingabe (etwa mit *input()*) geworfen – was eben auch einen Datenstrom darstellt.

## 14.2.1 Das Öffnen und Schließen einer Datei

Am Anfang jedes Dateizugriffs steht das Öffnen einer Datei und am Ende das Schließen. Das ist im Grunde offensichtlich. Dabei muss das Öffnen einer Datei explizit durchgeführt werden, während das explizite Schließen der offenen Datei zwar ebenso durchgeführt werden *sollte*, aber im Fall des Unterbleibens wird die Datei meist automatisch geschlossen.<sup>2</sup>

### 14.2.1.1 Die Built-in Function *open()*

Das Öffnen einer Datei erfolgt in Python mit der Built-in Function *open()*. Die Funktion liefert ein Objekt mit einem Zeiger auf die Datei zurück. Als Parameter geben Sie die Datei (gegebenenfalls mit Pfad) und einen **Dateimodus** an.

- Der Dateimodus "r" steht für „read“ (also lesen). Er ist Vorgabe, und wenn kein Dateimodus angegeben wird, gilt der Lesemodus.
- Der Dateimodus "w" steht für „write“ (schreiben). Die Datei wird neu erstellt und vorhandener Inhalt ersetzt.
- Der Dateimodus "a" steht für „append“ (anhängen). Neuer Inhalt wird an bestehenden Inhalt angehängt.

---

<sup>1</sup> In Python ist „alles“ ein Objekt.

<sup>2</sup> Auch wenn es dabei Probleme geben kann und man sich nicht auf diesen Automatismus verlassen sollte.

- Mit dem Anhängen des Pluszeichens kann man eine Datei sowohl für das Lesen als auch Schreiben öffnen ("r+", "w+" und "a+"). Mit einem nachgestellten *b* wird eine Datei binär geöffnet (etwa "wb" oder "rb").

### 14.2.1.2 Die Built-in Function *close()*

Mit der Methode *close()* wird eine Datei wieder geschlossen. Diese stellt das Objekt bereit, das den Zeiger auf die Datei enthält.

### 14.2.1.3 Ein schematisches Gerüst für einen Dateizugriff

Schematisch sieht das so aus:

#### Beispiel

```
fp = open("dat.txt").
```

...

```
fp.close(). ◀
```

### 14.2.2 Schreiben in eine Datei

Zum Schreiben von Inhalten in eine Datei gibt es die Methode *write()*. Hier ist das denkbar einfachste Beispiel (*Schreiben1.py*):

```
fp = open("zitat1MP.txt", "w")
fp.write("Ich meine, was hast du schon zu verlieren? " +
 "Du weißt du kommst aus dem Nichts " +
 "und du gehst wieder ins Nichts zurück. " +
 "Was hast du also verloren? Nichts!")
fp.close()
```

Natürlich kann man auch mehrere Ausgaben vornehmen. Solange die Datei offen ist, kann man die Methode *write()* beliebig oft aufrufen (*Schreiben2.py*):

```
fp = open("zitat2MP.txt", "w")
fp.write('Brian: Ihr seid alle Individuen!\n')
fp.write('Volk: Ja, wir sind alle Individuen!\n')
fp.write('Brian: Und ihr seid alle völlig verschieden!\n')
fp.write('Volk: Ja, wir sind alle völlig verschieden!\n')
fp.write('Einer: Ich nicht!\n')
fp.close()
```

### 14.2.3 Auslesen aus einer Datei

Das Auslesen einer Datei ist im Grunde etwas aufwendiger als das Schreiben, denn man muss irgendwie festlegen, wie viele Daten man einlesen möchte und wann man das Ende einer Datei erreicht hat, zumindest in anderen Programmiersprachen. Aber Python vereinfacht auch hier die Sache erheblich.

#### 14.2.3.1 Die Methode `read()`

Hier ist das denkbar einfachste Beispiel zum Einlesen einer Datei (*Lesen1.py*):

```
fp = open("zitat2MP.txt", "r")
print(fp.read())
fp.close()
```

Die Methode liest einfach bis zum Ende der Datei, und in dem Beispiel geben wir den Rückgabewert einfach aus (Abb. 14.1).

So einfach ist die Sache nicht in vielen Programmiersprachen, wenn man mehrere Zeilen einlesen möchte. Zudem können diverse Sonderzeichen den Lesevorgang etwas erschweren. Doch das Schöne an Python ist, dass die Methode `read()` bereits so gebaut wurde, dass in der Grundeinstellung eine ganze Datei in einen String eingelesen wird.

- ▶ **Tipp** Sie können bei der Methode `read()` als Parameter auch die Anzahl der Zeichen eingeben, die Sie einlesen wollen, etwa so:

```
print(fp.read(10)).
```

Dann werden in dem Beispiel nur maximal zehn Zeichen eingelesen.

#### 14.2.3.2 Zeilenweises Einlesen

Wenn man nicht die ganze Datei einlesen möchte, gibt es weitere Methoden. Das folgende Programm liest eine Datei zeilenweise ein, wobei mit der Methode `rstrip()` etwaige Leerzeichen und Newlines vom rechten Rand entfernt werden (*Lesen2.py*):

```
fp = open("zitat2MP.txt", "r")
for line in fp:
 print(line.rstrip())
fp.close()
```

**Abb. 14.1** Die Datei wurde ausgelesen

```
Brian: Ihr seid alle Individuen!
Volk: Ja, wir sind alle Individuen!
Brian: Und ihr seid alle völlig verschieden!
Volk: Ja, wir sind alle völlig verschieden!
Einer: Ich nicht!
```

Das letzte Beispiel hat die Datei Zeile für Zeile mit einer Schleife verarbeitet. Aber es kommt öfter vor, dass man eine Datei gerne in eine komplette Datenstruktur einlesen will. Dazu gibt es die Methode `readlines()`. Dabei werden allerdings Sonderzeichen und Escape-Sequenzen nicht eliminiert, und es gibt Unterschiede, was genau als Zeilenende zu sehen ist (Abb. 14.2). Beispiel (*Lesen3.py*):

```
fp = open("zitat2MP.txt", "r")
print(fp.readlines())
fp.close()
```

#### 14.2.4 Die `with`-Anweisung nutzen

Eine Datei sollte nach dem Öffnen und der Verarbeitung sauber geschlossen werden. Das ist Teil einer übergeordneten Vorgehensweise. Die `with`-Anweisung in Python wird oft in Verbindung mit sogenannten **Kontextmanagern** verwendet und dient dazu, Ressourcen wie Dateien oder Netzwerkverbindungen zu verwalten. Sie stellt sicher, dass diese Ressourcen korrekt initialisiert und freigegeben werden, auch wenn während der Verarbeitung ein Fehler auftritt. Dies vermeidet Ressourcenlecks in Python-Anwendungen. Das Hauptziel der `with`-Anweisung ist es, den Code sauberer und wartbarer zu gestalten.

Eine häufige Anwendung für die Verwendung der `with`-Anweisung ist das Öffnen und Lesen/Schreiben einer Datei, die danach automatisch geschlossen wird. Hier ist eine allgemeine Syntax für die Verwendung von `with` mit einer Datei:

##### Beispiel

```
with open('datei.txt', 'r') as datei:
 # Code, um die Datei zu verarbeiten.
 # Die Datei wird automatisch geschlossen, wenn der Block beendet ist. ◀
```

#### 14.2.5 Lese- und Schreibvorgänge absichern

Die bisherigen nahezu trivialen Schreib- und Lesevorgänge sollen nicht verschleiern, dass es bei Dateizugriffen immer Probleme geben kann. Beispielsweise kann eine Datei gesperrt sein. Oder beim Lesezugriff ist die Datei gar nicht vorhanden oder die Datei

```
['Brian: Ihr seid alle Individuen!\n', 'Volk: Ja, wir sind alle Individuen!\n', 'Brian: Und ihr seid alle völlig verschieden!\n',
'Volk: Ja, wir sind alle völlig verschieden!\n', 'Einer: Ich nicht!\n']
>>>
```

**Abb. 14.2** Die Datei wurde ausgelesen, aber Sonderzeichen wurden nicht interpretiert

ist nicht offen. Mit der *with*-Anweisung wurden schon erste Maßnahmen beschrieben, um eine Ressourcenverwaltung aufzubauen. Aber man sollte noch weiter gehen und Dateizugriffe immer mit einer Ausnahmebehandlung absichern und dort gegebenenfalls Gegenmaßnahmen ergreifen – auch wenn man nur eine Meldung ausgibt (Abb. 14.3). Dazu dient am besten *IOError*. Beispiel (*Lesen4.py*):

```
try:
 fp = open("dat.txt", "r")
 print(fp.read())
 fp.close()
except IOError:
 print("Fehler beim Zugriff auf die Datei")
```

- ▶ Ausnahmebehandlung und Ressourcenverwaltung über *with* können kombiniert werden. Es ist nicht identisch, denn Ausnahmebehandlung kümmert sich um die Konsistenz des Programmablaufs, während Kontextmanager die Ressourcen in den Fokus stellen.

#### 14.2.5.1 Binäre Dateien und das Konzept von Decode und Encode

In Python gibt es beim Darstellungsformat von Daten unterschiedliche Varianten, die sich umwandeln lassen. Man nennt das Decodieren und Codieren. Dies ist besonders wichtig, wenn Sie mit Zeichenketten arbeiten, die in verschiedenen Zeichencodierungen vorliegen, z. B. ASCII, UTF-8, UTF-16 usw.

Unter *encode* (Codieren) fasst man den Vorgang, bei dem Zeichenketten in eine bestimmte Zeichencodierung umgewandelt werden. Dies ist besonders wichtig, wenn Sie Text in eine binäre Darstellung umwandeln möchten, die in Dateien oder Netzwerk-kommunikation verwendet werden kann. Das gebräuchlichste Codierungsschema ist UTF-8. In Python können Sie die *encode*-Methode auf Zeichenketten anwenden, um sie in eine bestimmte Zeichencodierung zu konvertieren. Beispiel (*encode.py*):

---

#### Beispiel

```
text = "Hallo Welt!".
utf8_encoded = text.encode('utf-8').
print(utf8_encoded). ◀
```

```
Fehler beim Zugriff auf die Datei
>>>
```

**Abb. 14.3** Es gab eine Exception beim Dateizugriff und es wurde eine qualifizierte Gegenmaßnahme ergriffen

Dies wird die Zeichenkette "Hallo Welt!" in eine UTF-8-codierte Bytesequenz umwandeln.

Umgekehrt überführt *decode* (Decodieren) Bytes wieder in eine „normale“ Zeichenkette, indem die Bytes mit einer bestimmten Zeichencodierung interpretiert werden. In Python können Sie die *decode*-Methode auf Bytes aufrufen, um sie in eine Zeichenkette umzuwandeln. Beispiel (*decode.py*):

### Beispiel

```
utf8_bytes = b'Hello Welt!'.
decoded_text = utf8_bytes.decode('utf-8').
print(decoded_text). ◀
```

Dies wird die UTF-8-codierte Bytesequenz in die Zeichenkette "Hallo Welt!" umwandeln. Beachten Sie das vorangestellte *b* vor dem Text in der ersten Zeile. Dies markiert einen String als binär bzw. ein Byte-Array.

Es ist wichtig zu beachten, dass die Wahl der richtigen Zeichencodierung von entscheidender Bedeutung ist, um sicherzustellen, dass Ihre Textdaten korrekt dargestellt werden, gerade beim Schreiben und Lesen von Dateien. UTF-8 wird oft empfohlen, da es eine weit verbreitete und flexible Zeichencodierung ist, die viele verschiedene Zeichen unterstützt. Beim Arbeiten mit Dateien oder Netzwerkkommunikation ist es auch wichtig sicherzustellen, dass sowohl beim Codieren als auch beim Decodieren die gleiche Zeichencodierung verwendet wird, um Datenverlust und Probleme mit der Textdarstellung zu vermeiden.

Schauen wir uns das Schreiben und Lesen von binären Datei mit zwei Beispielcodes an. Sie können in Python den Text "Guten Tag" wie folgt in binäre Daten umwandeln und diese in eine Datei schreiben (*binaerschreiben.py*).

### Beispiel

```
Text, den wir in binäre Daten umwandeln wollen.
text = "Guten Tag".
Text in eine binäre Datenfolge (Bytes) mit der UTF-8 Zeichencodierung umwandeln.
binary_data = text.encode('utf-8').
Datei öffnen und die binären Daten schreiben.
with open('binäre_datei.bin', 'wb') as file:
 file.write(binary_data).
print(f'Die binären Daten wurden in die Datei "binäre_datei.bin" geschrieben.'). ◀
```

In diesem Code verwenden wir die *encode*-Methode, um den Text "Guten Tag" in binäre Daten mit der UTF-8 Zeichencodierung umzuwandeln. Dann öffnen wir eine Datei

im Binärmodus ('wb' steht für "write binary") und schreiben die binären Daten in diese Datei.

Nachdem der Code ausgeführt wurde, wird eine Datei mit dem Namen *binäre\_datei.bin* im aktuellen Verzeichnis erstellt, die die binären Daten enthält, die aus dem Text "*Guten Tag*" generiert wurden. Die binären Daten können dann später mit der entsprechenden Decodierungsmethode zurück in Text umgewandelt werden, wenn Sie die Daten wieder lesen. Das machen wir mit diesem Quellcode (*binaerlesen.py*).

---

### Beispiel

```
Datei öffnen und binäre Daten aus der Datei lesen.
with open('binäre_datei.bin', 'rb') as file:
 binary_data = file.read()
Binäre Daten in Text mit UTF-8 Zeichencodierung umwandeln.
text = binary_data.decode('utf-8').
Den Text ausgeben.
print(f'Gelesener Text aus der binären Datei: {text}'). ◀
```

In diesem Code öffnen wir die Datei *binäre\_datei.bin* im Binärmodus ('rb' für "read binary") und lesen die darin enthaltenen binären Daten. Anschließend verwenden wir die *decode*-Methode, um die binären Daten in Text mit der UTF-8 Zeichencodierung umzuwandeln. Schließlich wird der Text auf der Konsole ausgegeben.

---

## 14.3 Allgemeine Datei- und Verzeichnisoperationen

Python stellt auch Funktionen und Methoden für die üblichen Datei- und Verzeichnisoperationen bereit, etwa zum Löschen einer Datei (*remove()*) oder eines Verzeichnisses (*rmdir()*), zum Kopieren (*copy()*) und Verschieben (*move()*) von Dateien oder zum Überprüfen des Dateistatus etc. Die Basis stellen die folgenden Module bereit, die hier nur aufgelistet werden (es gibt noch weitere, für Details sei auf die Dokumentation verwiesen):

- `pathlib`
- `os.path`
- `fileinput`
- `stat`
- `filecmp`
- `tempfile`
- `glob`
- `shutil`

Das nachfolgende Beispiel überprüft einige wichtige Informationen zu einer Datei (Abb. 14.4). Der Einfachheit halber wird die Datei der Quelltext selbst sein (*Dateiinfos1.py*):

```
import os.path
import time
p='Dateiinfos1.py'
if os.path.exists(p):
 print('Die Datei ist vorhanden.')
 if os.path.isfile(p):
 print('Ist eine Datei und kein Verzeichnis')
 print('Größe:',os.path.getsize(p))
 print('Änderungsdatum:', time.ctime(os.path.getmtime(p)))
```

Das zweite Beispiel überprüft wieder einige Informationen von Dateien und erstellt dann ein Backup (also Kopie) der einen Datei. Dazu werden die Schritte mit einer Ausnahmebehandlung abgesichert. Die einzelnen Schritte werden in der Ausgabe dokumentiert (Abb. 14.5). Der Einfachheit halber wird auch hier wieder die Quelldatei der Quelltext selbst sein (*Dateiinfos2.py*).

```
Die Datei ist vorhanden.
Ist eine Datei und kein Verzeichnis
Größe: 320
Änderungsdatum: Wed Nov 15 18:38:59 2017
>>>
```

**Abb. 14.4** Dateiinformationen

**Abb. 14.5** Das Programm wurde zweimal ausgeführt

```
>>>
=====
 RESTART: F:/PythonQuell
codes/kap14/Dateiinfos2.py =====
===
Original existiert
Kopie existiert nicht
copy-Methode ausgeführt
Kopie vorhanden
>>>
=====
 RESTART: F:/PythonQuell
codes/kap14/Dateiinfos2.py =====
===
Original existiert
Kopie existiert
copy-Methode ausgeführt
Kopie vorhanden
>>>
```

```

import shutil
import os.path
ori='Dateiinfos2.py'
kopie='Dateiinfos2.bak'
if os.path.exists(ori):
 print('Original existiert')
else:
 print('Original existiert nicht')
if os.path.exists(kopie):
 print('Kopie existiert')
else:
 print('Kopie existiert nicht')
try:
 shutil.copy(ori,kopie)
 print('copy-Methode ausgeführt')
except FileNotFoundError:
 print('Orginaldatei nicht gefunden')
except FileExistsError:
 print('Zieldatei schon vorhanden - es wird nicht kopiert')
except PermissionError:
 print('Zugriffsprobleme auf Zieldatei - es wird nicht kopiert')
if os.path.exists(kopie):
 print('Kopie vorhanden')
else:
 print('Kopie nicht vorhanden')

```

- **Tipp** Beachten Sie, dass wir mit Ausnahmebehandlung den eigentlichen Kopiervorgang absichern und für die Quelldatei auf *FileNotFoundException* absichern. Das wäre aber nicht notwendig, denn bereits vorher wird ja explizit geprüft, ob die Originaldatei vorhanden ist. Das könnte man also direkt nutzen. Wobei man in der Praxis eigentlich meist umgekehrt vorgehen wird – man prüft nicht (!) vorher, ob die Datei vorhanden ist, sondern verlässt sich explizit auf die Ausnahmebehandlung.

Zusätzlich wird beim Kopieren mit *copy()* noch gegen *FileExistsError* und *PermissionError* gesichert, was etwa bei gesperrten oder auch schon einfach vorhandenen Zielfilen als Ausnahme ausgelöst werden kann.

---

## 14.4 Objekte serialisieren und deserialisieren

Es gibt in Python ein Modul mit Namen *pickle*, mit dem man Objekte serialisieren (engl. „to pickle“, einlegen) und deserialisieren kann. Damit kann man einen aktuellen Objektzustand in einer Datei persistent (dauerhaft) machen oder auch über einen Datenstrom verschicken und an anderer Stelle wieder reproduzieren.

### 14.4.1 Mit `dump()` den Objektzustand persistent machen

Mit der Methode `dump()` wird ein Objekt in einer Datei serialisiert und mit `load()` wieder deserialisiert. Die formale Syntax ist Folgende:

#### Beispiel

```
pickle.dump(obj, file[, protocol]) ◀
```

Eine serialisierte Version des Objektes `obj` wird in die Datei `file` geschrieben. Bei Bedarf kann optional das Protokoll angegeben werden (0=ascii, 1=kompakt, nicht lesbar, 2=opt. Klassen). Hier ist ein ganz einfaches Beispiel (*Serialisieren.py*), in dem wir **Zufallszahlen** mit dem Modul `random` und der Methode `random.randint()` erzeugen und daraus eine Liste aufbauen. Diese ist dann das Objekt, dessen Zustand wir serialisieren und damit dauerhaft machen wollen:

```
import pickle
import random
zahlen = []
Zufallsgenerator initialisieren
random.seed()
i = 0
while i < 10:
 i+=1
 # Zufallszahl zwischen 1 und 1000 ermitteln
 x = random.randint(1,1000)
 zahlen.append(x)
print(zahlen)
output = open('data.pkl', 'wb')
pickle.dump(zahlen, output)
output.close()
```

Das Beispiel wird eine Ausgabe folgender Art bewirken, wobei die Zahlen ja wie gesagt zufällig sind:

[572, 924, 525, 949, 495, 764, 134, 49, 806, 247].

### 14.4.2 Mit `load()` den Objektzustand reproduzieren

Nun ist das Konzept ja dafür da, dass man einen Objektzustand persistent machen und später wieder reproduzieren kann. Das Deserialisieren geht formal so:

**Beispiel**

```
pickle.load(file) ◀
```

Damit erkennt die Methode automatisch, in welchem Format die Daten gespeichert wurden. So würde aus der Datei, die obiges einfaches Beispiel angelegt hat, der Objektzustand wieder reproduziert (*Deserialisieren.py*):

```
import pickle
input = open('data.pkl', 'rb')
data=pickle.load(input)
input.close()
for i in data:
 print(i)
```

---

## 14.5 Datenbankzugriffe

Der Umgang mit Dateien ist nur eine Facette zum persistenten Speichern und Auslesen von Daten. Wenn die Daten komplexer werden, benötigt man strukturierte Informationen, die man qualifiziert verwerten kann. Und das führt zu Datenbankkonzepten. Das Python-API stellt aber unter anderem auch eine Unterstützung für den Umgang mit einer SQLite-Datenbank (und auch anderen Datenbanken) zur Verfügung. Dort kann man mit der Datenbank die Abfragesprache SQL arbeiten.

SQL steht für Structured Query Language oder auch Standard Query Language. Die Sprache dient zur Definition von Datenstrukturen in relationalen Datenbanken sowie zum Bearbeiten und Abfragen von darauf basierenden Datenbeständen über einen definierten Befehlssatz, der von fast allen gängigen Datenbanken unterstützt wird. Allerdings gibt es leicht voneinander abweichende Dialekte. In dem Buch werden wir SQL voraussetzen, bei Bedarf sollten Sie sich in ergänzenden Quellen dazu informieren.

### 14.5.1 Was ist SQLite?

Bei SQLite handelt es sich um eine Bibliothek beziehungsweise ein API zur Programmierung eines relationalen Datenbanksystems. Die Bibliothek unterstützt dazu einen Großteil des normalen SQL-Sprachumfangs. Das geht hin bis zu Transaktionen, Views und Trigger sowie benutzerdefinierte Funktionen. SQLite ist aber extrem schlank. Allerdings ist SQLite kein vollständiges Datenbankmanagementsystem (DBMS), ist nicht typsicher, und die Verwaltung von Benutzer- und Zugriffsberechtigungen erfolgt nicht auf Datenbankebene, sondern über die Zugriffsberechtigungen des Dateisystems.

### 14.5.2 Zugriff auf SQLite in Python

Um auf SQLite in Python zuzugreifen, braucht man zwei Importe:

#### Beispiel

```
import sqlite3

import os.path
```

Eigentlich benötigt man nur das Modul *sqlite3*, aber man spricht ja ganz konkret eine Datei an, und da kann man vorher dessen Existenz prüfen. Oder man nutzt eine Ausnahmebehandlung, dann kann man auf *os.path* auch verzichten, wenn man die eigentliche Dateisicht nicht weiter benötigt.

#### 14.5.2.1 Die Methode `connect()` zum Öffnen und `close()` zum Schließen

Die entscheidende Methode, um mit einer SQLite-Datenbank zu arbeiten, ist *connect()*. Damit geben Sie den Datenbanknamen vor. Die Methode liefert als Rückgabe ein neues Datenbankobjekt, über das die konkrete Manipulation der Datenbank dann in der Folge abläuft.

Das wäre ein Beispiel für die Erzeugung einer SQLite-Datenbank:

#### Beispiel

```
connection=sqlite3.connect('sqldb.db')
```

Mit der Methode *close()* wird die Verbindung zur Datenbank geschlossen – wie bei „normalen“ Dateien.

#### Beispiel

```
connection.close()
```

#### 14.5.2.2 Konkrete Transaktionen durchführen mit `cursor()` und `execute()` sowie einer Fetch-Methode

Die Methoden *cursor()* und *execute()* erlauben es, eine Datenbanktransaktion auszuführen. Mit *cursor()* holt man sich über das Verbindungsobjekt einen Zeiger auf die Datenbank.

#### Beispiel

```
cursor=connection.cursor()
```

Mit dem Datenbankzeigerobjekt lassen sich dann über `execute()` SQL-Anweisungen ausführen.

Beachten Sie, dass die SQL-Statements in Python in **drei** Hochkommata eingeschlossen werden. Das ist dann aus Sicht von Python ein Kommentar. Beispiel:

#### Beispiel

```
cursor.execute('"SELECT * FROM personen"') ◀
```

#### 14.5.2.3 Die Ergebnisse auswerten mit einer Fetch-Methode

Mit einer Methode wie `fetchall()` wird dann auf dem Ergebnis der Datenbankabfrage operiert. Diese Methode liefert eine Datenstruktur mit allen Datenzeilen, die im SQL-Statement selektiert wurden.

#### Beispiel

```
rows = cursor.fetchall()
for row in rows:
 ... ◀
```

#### 14.5.2.4 Die commit()-Methode zum Speichern von Änderungen

Bei Datenbankänderungen müssen diese explizit auf die Datenbank mit `commit()` geschrieben werden, damit sie gespeichert werden.

#### Beispiel:

```
connection.commit() ◀
```

### 14.5.3 Ein konkretes Datenbankbeispiel

Hier ist ein komplettes Datenbankbeispiel, bei dem eine Datenbank erzeugt, dann gefüllt und zum Schluss ausgegeben werden soll.

#### 14.5.3.1 Die Datenbankdatei erstellen

Zuerst soll die Datenbank angelegt werden, wenn Sie nicht vorhanden ist. Wenn sie beim Anlegen schon vorhanden ist, wird das mit `os.path.exists()` erkannt (Abb. 14.6). Den gesamten Prozess kann man direkt mit einem SQL-Statement durchführen (*Datenbankanlegen.py*):

**Abb. 14.6** Das Programm wurde zweimal ausgeführt

```
Datenbank erstellt
>>>
===== RESTART: F:/Pyt
Datenbank schon vorhanden
>>>
```

```
import sqlite3
import os.path
if not os.path.exists('sqldb.db'):
 connection=sqlite3.connect('sqldb.db')
 cursor=connection.cursor()
 cursor.execute('''CREATE TABLE personen(name TEXT, vorname
TEXT) ''')
 print("Datenbank erstellt")
 connection.close()
else:
 print("Datenbank schon vorhanden")
```

#### 14.5.3.2 Daten in die Datenbank schreiben

Nun schreiben wir Daten in die Datenbank. Dazu kann ein Anwender seinen Nachnamen und seinen Vornamen eingeben (Abb. 14.7) und die beiden Werte werden in die Datenbank geschrieben (*Datenbankschreiben.py*):

```
import sqlite3
import os.path
n = input("Eingabe Nachname\n")
v = input("Eingabe Vorname\n")
if os.path.exists('sqldb.db'):
 connection=sqlite3.connect('sqldb.db')
 cursor=connection.cursor()
```

**Abb. 14.7** Mehrere Datensätze wurden angelegt

```
Eingabe Nachname
Steyer
Eingabe Vorname
Ralph
Daten geschrieben
>>>
=====
===== RESTART: F:/_
Eingabe Nachname
Roth
Eingabe Vorname
Florian
Daten geschrieben
>>>
=====
===== RESTART: F:/_
Eingabe Nachname
Roth
Eingabe Vorname
Felix
Daten geschrieben
>>>
```

```

cursor.execute(''')INSERT INTO personen VALUES (?,?)''',(n,v))
print("Daten geschrieben")
connection.commit()
connection.close()
else:
 print("Datenbank nicht vorhanden")

```

#### 14.5.3.3 Die Daten auslesen

Nun soll die Datenbank ausgelesen werden. Dazu werden die Daten mit einer SQL-Anwendung ausgewählt, und mit *fetchall()* wird eine Datenstruktur mit allen Datenzeilen erzeugt, die im SQL-Statement selektiert wurden. Darüber kann man iterieren und die Daten anzeigen (Abb. 14.8 – *Datenbanklesen.py*):

```

import sqlite3
import os.path
if os.path.exists('sql.db'):
 connection=sqlite3.connect('sql.db')
 cursor=connection.cursor()
 cursor.execute(''')SELECT * FROM personen''')
 rows=cursor.fetchall()
 for row in rows:
 print(row[0], row[1])
 print("Alle Daten ausgelesen")
 connection.close()
else:
 print("Datenbank nicht vorhanden")

```

- ▶ **Tipp** Natürlich kann man auch bei Datenbankzugriffen eine Ausnahmebehandlung implementieren. Bei SQL-Statements kann ein *sqlite3.Error* auftreten, den man wie folgt abfangen kann:

```

try:
...
except sqlite3.Error as e:
...

```

**Abb. 14.8** Die Datenbank wurde wieder ausgelesen

|                                                                                      |
|--------------------------------------------------------------------------------------|
| <pre> Steyer Ralph Roth Florian Roth Felix Alle Daten ausgelesen &gt;&gt;&gt; </pre> |
|--------------------------------------------------------------------------------------|



# Umgang mit Datum und Zeit – Terminsachen

15

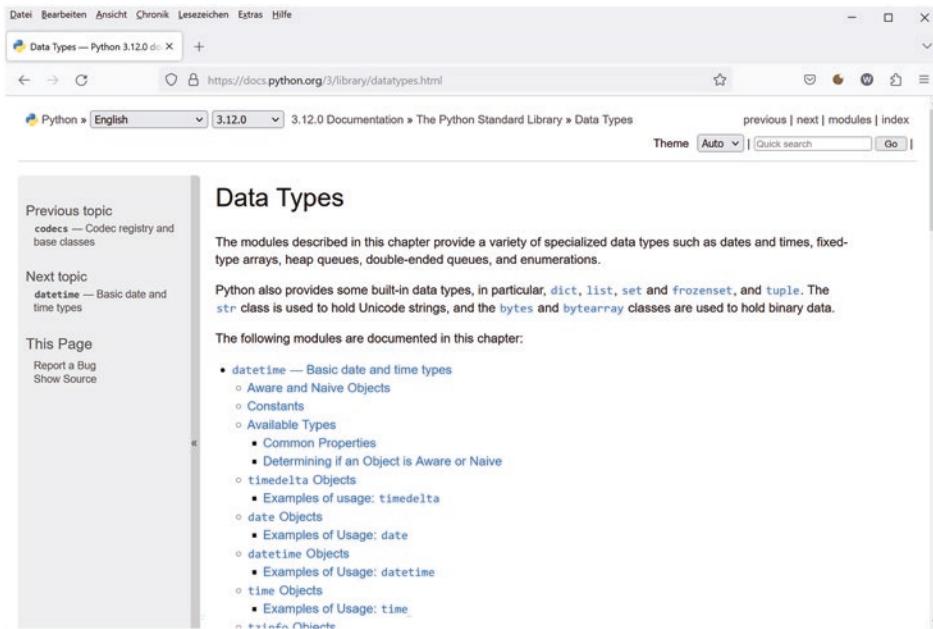
## 15.1 Was behandeln wir in diesem Kapitel?

Der Umgang mit Datum und Zeit gehört zu den Grundaufgaben, die man recht oft bei Programmierung vorfindet. Dafür gibt es die unterschiedlichsten Anwendungen. Wir wollen in diesem sehr kompakten Kapitel einen kleinen Blick darauf werfen, um den grundsätzlichen Ansatz kennenzulernen.

## 15.2 Allgemeines zum Umgang mit Datum und Zeit

Beim Umgang mit Datum und Uhrzeit arbeiten fast alle Computersysteme eigentlich gleich. Der 01. Januar 1970, 0.00 Uhr ist der interne **Zeitnullpunkt**, der in nahezu allen Skript- und Programmiersprachen als Speicherungs- und Berechnungsbasis für alle Operationen mit einem Datum dient. Ab diesem Zeitpunkt wird die Zeit numerisch gezählt, sowohl rückwärts bei Terminen davor als auch natürlich für die Zukunft, und zwar entweder ganz genau in Millisekunden oder zumindest in Sekunden. Und da ein jedes Datum immer intern mit einer ganzen Zahl verwaltet wird, können Sie mit Datumsobjekten meist auch wie mit gewöhnlichen Zahlen rechnen.

Man nennt dieses Verfahren auch die **Unix-Zeit**, die für das Betriebssystem Unix entwickelt und als POSIX-Standard festgelegt wurde. Vor der Unix-Version 6, die 1975 abgelöst wurde, zählte die Unix-Uhr in Hundertstelsekunden. Seit der Unix-Version 6 zählt die Unix-Zeit die vergangenen Sekunden seit dem Zeitnullpunkt. Dieses Startdatum wird auch als **The Epoch** bezeichnet.



**Abb. 15.1** In der Dokumentation sind alle wichtigen Informationen zu den Datums- und Zeitmodulen zu finden

## 15.3 Die Python-Module

Python stellt für nahezu alle Berechnungen mit Datum und Zeit Funktionalitäten bereit, die weitgehend in den Modulen *datetime*, *calendar* oder *time* zu finden und in der Dokumentation unter <https://docs.python.org/3/library/datatypes.html> beschrieben sind (Abb. 15.1).

Darüber hinaus gibt es aber noch diverse weitere Module für spezielle Aufgaben im Umgang mit Datum und Zeit. Sie können aus einem Datum jeden sinnvollen Bestandteil (Tag, Monat, Jahr, Wochentag, Stunde, Minute, Sekunde etc.) extrahieren und verwenden.

## 15.4 Typische Beispiele für Operationen mit Zeit und Datum

Wir schauen uns hier die grundsätzliche Vorgehensweise für Operationen mit Zeit und Datum an und spielen ein paar Beispiele für typische Anwendungen durch.

### 15.4.1 Das aktuelle Systemdatum des Computers auslesen

Zuerst einmal wollen wir einfach das aktuelle Systemdatum des Computers auslesen und ausgeben. Das geht mit der Methode *datetime.now()* ganz einfach (*Dat1.py*).

```
import datetime
now = datetime.datetime.now()
print ("Das aktuelle Datum:", now)
```

Sie sehen am Anfang den Import des Moduls *datetime*, dann wird vollqualifiziert auf die Methode *now()* zugegriffen.

## 15.4.2 Ein beliebiges Datumsobjekt erstellen

Es gibt verschiedene Möglichkeiten, um ein Datumsobjekt für einen beliebigen Zeitpunkt zu erstellen. Hier sehen Sie die Anwendung der Methode *datetime.strptime()* aus dem Modul *datetime*, der man ein String mit einem Datumsformat übergeben kann.

Eine Alternative ist die Methode *strftime()* aus dem Modul *time*, die wir in Verbindung mit *mktime()* aus dem gleichen Modul ebenso anwenden. Mit der zusätzlichen Anwendung von *gmtime()* aus dem gleichen Modul bereiten wir das Datumsobjekt dann auf.

### 15.4.2.1 Direktiven zur Aufbereitung

Sowohl bei *time.gmtime()* als auch *datetime.strptime()* und einigen weiteren Methoden kann man mit sogenannten Direktiven das Format des Übergabestrings verdeutlichen. Hierbei bedeutet

- %d den Tag des Monats,
- %H die Stunden im 24-Stundenformat,
- %I die Stunden im 12-Stundenformat,
- %m den Monat,
- %b den Monatsnamen,
- %y das Jahr,
- %M die Minuten oder
- %S die Sekunden.

Darüber hinaus gibt es noch weitere Direktiven, die in der Dokumentation beschrieben werden.

### 15.4.2.2 Zwei Datumsobjekte erzeugen

Hier ist das konkrete Beispiel, in welchem einfach zwei Datumsobjekte mit den beiden Techniken erzeugt werden (*Dat2.py*).

```
import datetime
import time
datum1 = "10/10/02"
Date1 = datetime.datetime.strptime(datum1, "%m/%d/%y")
```

```

print("Ein Datum in der Vergangenheit:", Date1)
datum2 = (2028, 2, 17, 17, 3, 38, 0, 0, 0)
Date2 = time.mktime(datum2)
print("Das Datum als Zeitstempel:", Date2);
print(time.gmtime(Date2))
print("Ein anderes Datum in der Zukunft:",
 time.strftime("%b %d %Y %H:%M:%S", time.gmtime(Date2)))
datum3 = (2029, 2, 17, 0, 0, 0, 0, 0, 0)
Date3 = time.mktime(datum3)
print("Noch ein anderes Datum in der Zukunft:",
 time.strftime("%b %d %Y %H:%M:%S", time.gmtime(Date3)))

```

Beim ersten Datumsobjekt nutzen wir in dem String die Angabe des Monats, Tages und Jahres, die jeweils durch einen Slash getrennt werden. Dieses Format beschreibt in der Methode *datetime.strptime()* der zweite Parameter durch einen entsprechenden Formatstring mit den Direktiven (Abb. 15.2, Abschn. 15.4.2.1).

Die Methode *time.mktime()* benötigt eine ganze Reihe an Parametern, genau genommen ein Tupel mit neun Parametern, von denen eigentlich nur die ersten sechs von Bedeutung, aber auch die anderen zumindest formal zwingend sind. Zuerst kommt

- das Jahr,
- dann kommen der Monat,
- der Tag des Monats,
- die Uhrzeit,
- die Minuten und
- die Sekunden.

Alle Angaben, die Sie nicht setzen wollen, setzen Sie einfach auf 0. Beachten Sie aber die Zeitzonen und Sommer- wie Winterzeit, die sich auf die Uhrzeit auswirken (Abb. 15.2).

Nun erhalten wir mit der Methode *time.mktime()* aber einen Zeitstempel, und das ist eine Zahl – die Anzahl der Sekunden seit dem Zeitnullpunkt.

Die Methode *time.gmtime()* braucht jedoch genau so eine Zahl, um daraus eine assoziierte Struktur mit den Datumsbestandteilen zu machen. Beachten Sie die Namen der Felder (Abb. 15.2).

```

Ein Datum in der Vergangenheit: 2002-10-10 00:00:00
Das Datum als Zeitstempel: 1834416218.0
time.struct_time(tm_year=2028, tm_mon=2, tm_mday=17, tm_hour=16, tm_min=3, tm_sec=38, tm_wday=3, tm_yday=48,
tm_isdst=0)
Ein anderes Datum in der Zukunft: Feb 17 2028 16:03:38
Noch ein anderes Datum in der Zukunft: Feb 16 2029 23:00:00

```

**Abb. 15.2** Verschiedene Datumsobjekte

Diese assoziierte Struktur bereiten wir mit `time.strptime()` und den oben beschriebenen Direktiven auf.

### 15.4.2.3 Datumsbestandteile extrahieren

Sehr oft braucht man gewisse Bestandteile aus einem vorhandenen Datumsobjekt. Dazu stehen diese Eigenschaften bereit, die man verwenden kann. Leider unterscheiden sich die Namen der Eigenschaften etwas in den verschiedenen Datumsobjekten. Beachten Sie das nachfolgende Beispiel *Dat4.py*.

```
import datetime
import time

datum1 = "10/10/21"
Date1 = datetime.datetime.strptime(datum1, "%m/%d/%y")
print("Das Jahr:", Date1.year)
print("Monat:", Date1.month)
print("Tag:", Date1.day)
print("Stunde", Date1.hour)
print("Minuten:", Date1.minute)
print("Sekunden:", Date1.second)
print("Millisekunden:", Date1.microsecond)
datum2 = (2024, 2, 17, 17, 3, 38, 0, 0, 0)
Date2 = time.mktime(datum2)
d2= time.gmtime(Date2)
print("Das Jahr:", d2.tm_year)
print("Monat:", d2.tm_mon)
print("Tag:", d2.tm_mday)
print("Stunde", d2.tm_hour)
print("Minuten:", d2.tm_min)
print("Sekunden:", d2.tm_sec)
```

Während mit `datetime.strptime()` ausgeschriebene Eigenschaften zum Einsatz kommen, enthält die assoziierte Struktur von `time.gmtime()` die Informationen in leicht codierter Form, die aber auch noch einigermaßen sprechend ist, zumal die Ausgabe `print(time.gmtime(Date2))` aus dem letzten Beispiel diese Bezeichner bereits ausgegeben hat. Das ist die Ausgabe des Listings:

*Das Jahr: 2021.*

*Monat: 10.*

*Tag: 10.*

*Stunde 0.*

*Minuten: 0*

*Sekunden: 0*

*Millisekunden: 0*

*Das Jahr: 2024.*

*Monat: 2*

*Tag: 17.*

*Stunde 16.*

*Minuten: 3*

*Sekunden: 38.*

#### 15.4.2.4 Mit Datumsobjekten rechnen

Da Datumsobjekte nur Zahlen beinhalten, kann man in jeder denkbaren Form damit rechnen, wobei es von der Art des Datumsobjekts abhängt, ob Sie direkt Zahlen oder ein bereits wiederaufbereitetes Resultat erhalten. Beides kann sinnvoll sein. Beachten Sie das nachfolgende Beispiel *Dat5.py*, in dem wir von einem Startdatum aus die Anzahl der Tage bis zu einem Zieldatum rein arithmetisch ermitteln.

```
import datetime
now = datetime.datetime.now()
datum1 = "12/24/17"
Date1 = datetime.datetime.strptime(datum1, "%m/%d/%y")
print(Date1 - now)
```

Die Ausgabe ist so etwas wie dies:

*38 days, 6:22:47.329778.*

Diese ist offensichtlich bereits wieder aufbereitet worden, doch das verschleiert ein bisschen, dass wirklich nur mit Zahlen gearbeitet wird.<sup>1</sup>

Beachten Sie diese Abwandlung *Dat6.py*, in der wir mit *time.mktime()* die Datumsobjekte erstellen.

```
import time
datum1 = (2024, 2, 17, 17, 3, 38, 0, 0, 0)
Date1 = time.mktime(datum1)
datum2 = (2024, 2, 17, 17, 3, 37, 0, 0, 0)
Date2 = time.mktime(datum2)
print(Date1 - Date2)
datum3 = (2024, 12, 24, 0, 0, 0, 0, 0, 0)
Date3 = time.mktime(datum3)
print(Date3 - Date2)
```

Hier erhalten Sie die Zahlen 1.0 und 26808982.0, und das ist jeweils die Anzahl der Sekunden zwischen den beiden Datumsobjekten. Und damit sind Ihrer Phantasie keine Grenzen gesetzt, was Sie alles mit Datumsobjekten arithmetisch anstellen wollen.

---

<sup>1</sup> Man kann aber die Zahlenwerte auch extrahieren, wenn das gewünscht ist.



# Grafische Oberflächen (GUI) mit Python – tkinter & Co als GUI-Framework

16

## 16.1 Was behandeln wir in diesem Kapitel?

Kaum ein modernes Clientprogramm kommt ohne grafische Oberfläche (**GUI – graphical user interface**) daher. Natürlich kann man auch in Python solche grafischen Oberflächen erstellen. Dazu stellt Python ein Framework mit allen zu erwartenden Elementen bereit, damit wollen wir uns in dem abschließenden Kapitel beschäftigen.

## 16.2 Hintergrundinformationen zu modernen grafischen Oberflächen

Die GUI-Programmierung basiert in den meisten modernen Programmiersprachen auf ähnlichen Konzepten. Wie bei fast allen modernen Programmiersprachen stellt auch das GUI-Framework von Python allgemeine Komponenten der Anwenderschnittstelle wie Fenster, Schaltflächen oder Menüs zur Verfügung. Es gibt also Komponenten und Container, in denen die Komponenten integriert werden müssen, damit eine vollständige und sinnvolle Anwenderschnittstelle erstellt werden kann. Der äußerste Container ist in der Regel ein Frame. Ein weiterer bedeutender Bestandteil ist die Technik der Layout-Manager, die sich um das grundsätzliche Layout einer Applikation kümmern. Zusätzlich braucht es aber auch einen Mechanismus, um auf Ereignisse reagieren zu können, die vom Anwender über Komponenten ausgelöst werden.

Eine Python-GUI basiert genau auf dem Grundkonzept, dass jedes Fenster mehrere verschachtelte Ebenen besitzt, die aus Komponenten aufgebaut sind. Dies beginnt mit dem äußersten Fenster und endet mit dem kleinsten eigenständigen Bestandteil der

Benutzeroberfläche. Diese Verschachtelung bildet eine Hierarchiestruktur. Die Klassen des GUI-Frameworks sind so entwickelt worden, dass deren Struktur eine vollständige Benutzeroberfläche abbilden kann, wobei wir das vollständige Konzept in dem Kapitel nur andeuten wollen.

Die einzelnen GUI-Komponenten sind die Elemente, über die eine Interaktion mit dem Endanwender konkret realisiert wird. Sie kennen sie aus den unterschiedlichen grafischen Benutzerschnittstellen. Typische Komponenten sind Schaltflächen, Label, Kontrollkästchen, Optionsfelder, Listen, Auswahlfelder, Textfelder, Textbereiche oder Menüs.

Oberflächenprogrammierung umfasst aber wie gesagt auch die Reaktion auf Benutzeraktionen. Wenn ein Anwender etwa auf eine Schaltfläche klickt, muss im Programm hinterlegt sein, was daraufhin zu tun ist. Das Eventhandling von Python basiert auf Ereignismethoden, die mit Komponenten verbunden werden.

---

## 16.3 Konkrete GUI-Konzepte in Python und das Modul `tkinter`

Pythons GUI-Anwendungen basieren auf dem Modul `tkinter`. Fenster und Steuerelemente werden in dem Framework als solche gerade beschriebenen GUI-Elemente betrachtet, die sich ineinander verschachteln lassen. Um eine GUI-Anwendung zu schreiben, muss man also wissen, wie sich die jeweiligen GUI-Elemente erzeugen lassen und wie man diese dann ineinander verschachtelt.

### 16.3.1 Ein Fenster vom Typ TK als Basis jeder GUI-Applikation

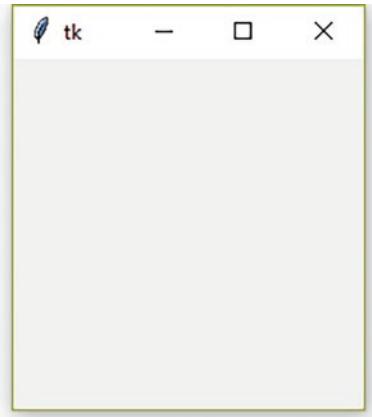
Die einfachste Variante einer GUI nutzt ein Objekt vom Typ `Tk` und eine sogenannte `mainloop`, mit der die Anwendungssteuerung an die GUI übergeht (`GUI1.py`). So ein minimaler Code erzeugt bereits ein grafisches Fenster mit einer Vorgabegröße und allen üblichen Features wie der Möglichkeit zum Vergrößern, Verschieben oder Ablegen in der Taskleiste (Abb. 16.1).

```
from tkinter import *
master=Tk()
master.mainloop()
```

### 16.3.2 Der übliche OO-Ansatz

Aber meist strukturiert man eine GUI-Anwendung etwas besser und setzt zudem besonders bei grafischen Python-Programmen auf ein objektorientiertes Fundament. Man beginnt wie gesagt in der Regel mit einem Hauptfenster, das vom Typ `Tk` ist und aus

**Abb. 16.1** Ein minimales GUI-Programm



einem **Frame** besteht, das über eine Applikationsklasse verwaltet wird. Dazu wird wie beschrieben die *mainloop* zur Anwendungssteuerung an die Applikation mit dem Frame übertragen.

Diese Konstruktion wurde im Kapitel zur objektorientierten Programmierung (Kap. 10) zwar schon eingeführt, aber bisher nicht weiter verwendet. Die Situation wird aber bei einem grafischen Python-Programm immer so oder so ähnlich aussehen.

Ein einfaches, aber schon mit objektorientierten Standardstrukturen versehenes GUI-Programm hat in Python also in der Regel so eine Minimalstruktur (*GUI2.py*):

```
from tkinter import *
class Application(Frame):
 def __init__(self, master=None):
 Frame.__init__(self, master)
root=Tk()
app=Application(master=root)
app.mainloop()
```

Man hat also im Wesentlichen einen Konstruktor mit einem Frame und kann darin dann andere GUI-Elemente wie Label, Button oder Eingabefelder (Objekte vom Typ *Entry*) erstellen und vor allen Dingen im Frame verankern.

### 16.3.3 Die Layout-Manager

Das Hinzufügen von Elementen einer grafischen Oberfläche läuft bei tkinter beziehungsweise in Python über das Konzept besagter Layout-Manager oder Geometry-Manager. Die müssen wir etwas genauer betrachten, denn sie sind der elementare Kern des gesamten Frameworks.

### 16.3.3.1 Was ist die Idee bei Layout-Managern?

Moderne grafische Oberflächen folgen oft einem Konzept von verschachtelten Anordnungscontainern. Ein bedeutender Bestandteil des Konzepts ist dabei meist die Technik solcher Layout-Manager. Wenn man so ein Konzept im GUI-Framework einsetzt, muss man sich als Programmierer beim Hinzufügen von Komponenten (Schaltflächen, Eingabefelder etc.) zu Containerobjekten in vielen Fällen (scheinbar) überhaupt nicht um die genauen Positionen innerhalb der Container oder vor allen Dingen deren Größe kümmern. Diese Komponenten werden einfach „irgendwo“ auf der Oberfläche platziert. Und sie haben irgendeine scheinbar zufällige Größe, die sich manchmal auch verändern kann, wenn sich die Größe des umgebenden Fensters, der Komponente selbst oder des Anordnungscontainers ändert.

Dieses Verhalten ist jedoch kein Zufall und keinesfalls ohne System. Positionen und Größen werden nur automatisch im Hintergrund gemanagt. Das ist eine der großen Stärken dieses GUI-Konzepts, auch wenn man sich erst einmal mit der Konzeption vertraut machen muss.

Schauen wir uns als Alternative erst einmal andere Fensterkonzepte an, in denen man für die Komponenten der Benutzeroberfläche überwiegend exakte Koordinaten und Größen angibt. Ein Programmierer hat (scheinbar) vollständige Kontrolle über das Aussehen der Benutzeroberfläche. Aber dies ist trügerisch. Die hart codierten Angaben ergeben nur Sinn, wenn auf der Zielplattform einer Applikation die gleiche Bildschirmauflösung und Größe eines grafischen Fensters vorhanden ist. Was ist, wenn der Anwender nun eine andere Auflösung hat? Oder wenn der Bildschirm gedreht wird, was bei mobilen Geräten ständig passiert? Die Funktionalität kann durch solche Faktoren massiv beeinträchtigt werden.

Bei konventionellen Fensterkonzepten muss ein Programmierer sämtliche denkbaren Auflösungen, Fenstergrößen und Orientierungen berücksichtigen (zumindest diejenigen, die er unterstützen möchte), diese zur Laufzeit abfragen und für jede Maske eine individuelle Oberfläche für jede Auflösung generieren. Das ist ein erheblicher Aufwand und dennoch immer noch keine befriedigende Lösung, denn unter Umständen hat man nicht alle Situationen bedacht, und es kann sogar zum Verdecken von Komponenten der Oberfläche kommen.

Layout-Manager lösen diese Probleme auf eine intelligente Art und Weise. Das GUI-System kümmert sich weitgehend selbstständig um Größenanpassung und Positionierung der Komponenten auf der Oberfläche. Je nach Plattform und Bedingungen werden die Komponenten auf der Oberfläche optimal angepasst. Und wenn man Container verschachtelt, hat man genügend Kontrolle über Position und Größe von Komponenten, ohne sich so weit festzulegen, dass eine veränderte Bedingung ein System zur Unfähigkeit degradieren kann. Sie können mit dem Konzept allgemein eine Art Hintergrundverwalter der Oberfläche anweisen, wo Ihre Komponenten bei Bedarf im Verhältnis zu den anderen Komponenten stehen sollen. Ein Layout-Manager findet nach gewissen Regeln heraus, an welche Stellen die Komponenten am besten passen und ermittelt die besten Größen der Komponenten. Aber man kann in einem GUI-Framework natürlich auch Layout-Manager erstellen, die eine genaue Positionierung von Komponenten gestatten.

- ▶ Layout-Manager können verschachtelt werden. Sehr oft kombiniert man flexible Layout-Manager bei äußeren Containern mit fixierenden Layout-Managern im Inneren. Das werden wir hier aber nicht ausarbeiten, da GUI-Programmierung nur ein Thema von vielen darstellt.

Das genaue Aussehen einer solchen flexiblen Oberfläche wird im Allgemeinen von mehreren Aspekten festgelegt.

- Von der Plattform und den genauen Bedingungen dort. Darüber kann ein Programmierer keine Aussagen machen und muss es auch nicht. Darum geht es ja gerade.
- Von der Reihenfolge, in der die Komponenten in einem Container eingefügt werden. Dieses Faktum ist naheliegend. Einmal kann gelten: Wer zuerst kommt, mahlt zuerst. Das soll heißen, dass zuerst eingefügte Komponenten auch vorher angeordnet werden. Allerdings kann es auch sein, dass eine später eingefügte Komponente eine vorherige ersetzt oder überlagert oder die Reihenfolge der Einfügung gar keine Rolle spielt. Wie das genau abläuft, hängt ab von dem genauen GUI-Konzept.
- Richtig interessant ist die Art des Layout-Managers. Die drei Layout-Manager von *tkinter* sind diese:
  - *place*
  - *grid*
  - *pack*

### 16.3.3.2 Die Layout-Manager von *tkinter*

Schauen wir uns die Layout-Manager von *tkinter* im Detail an.

- ▶ Attribute wie *width*, *height*, *pady* oder *padx* geben bei Bedarf die Breite, Höhe und den Innenabstand von Komponenten an. Wenn diese nicht gesetzt werden, werden Vorgabewerte gesetzt. Diese sind entweder 0 oder ergeben sich aus dem Inhalt von Komponenten.

#### 16.3.3.2.1 Der *place*-Layout-Manager

Bei der Verwendung von *place* spezifiziert man die Position eines Elements durch Angabe von x/y-Koordinaten als Parameter. Das ist also ein statisches Layoutkonzept, was am ehesten der klassischen Gestaltung von Oberflächen entspricht. Hier ist dazu ein kleines Beispiel (*GUI3.py*):

```
from tkinter import *
class Application(Frame):
 def __init__(self, master=None):
 meinLbl=Label(master, text=" Ein Kratzer?! Ihr Arm ist ab!")
 Frame.__init__(self, master)
 meinLbl.place(x=10,y=10)
```

```
root=Tk()
app=Application(master=root)
app.mainloop()
```

Wir arbeiten hier mit einer Komponente vom Typ *Label*. Das ist ein Text, der auf der grafischen Oberfläche angezeigt werden kann. Das Label wird mit der Methode *place()* der Oberfläche hinzugefügt. Sie sehen in dem Resultat, dass an den Koordinaten 10, 10 der spezifizierte Text angezeigt wird (Abb. 16.2). Die Koordinaten der Methode *place()* spezifizieren die linke obere Ecke der Komponente.

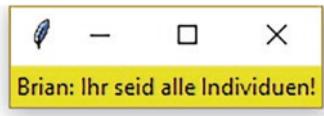
### 16.3.3.2.2 Der pack-Layout-Manager

Bei dem *pack*-Layout-Manager wird ein fließendes Layout geschaffen, das insbesondere den verfügbaren Platz optimal auszunutzen versucht. Betrachten wir die erste Version eines Beispiels dazu, das wir Schritt für Schritt noch ausbauen wollen, um das Verhalten genauer zu verstehen (*GUI4.py*):

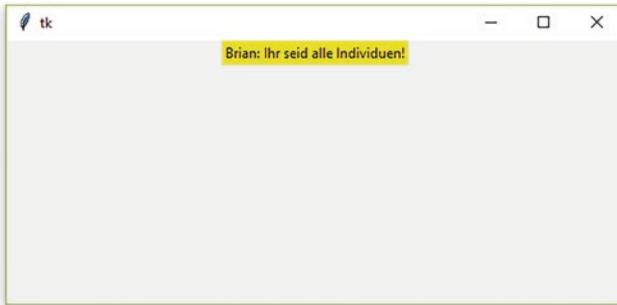
```
from tkinter import *
class Application(Frame):
 def __init__(self,master=None):
 t1 = 'Brian: Ihr seid alle Individuen!'
 t2 = 'Volk: Ja, wir sind alle Individuen!'
 t3 = 'Brian: Und ihr seid alle völlig verschieden!'
 t4 = 'Volk: Ja, wir sind alle völlig verschieden!'
 t5 = 'Einer: Ich nicht!'
 lbl1=Label(master,text=t1, bg="yellow")
 lbl2=Label(master,text=t2, bg="green")
 lbl3=Label(master,text=t3, bg="gray")
 lbl4=Label(master,text=t4, bg="white", fg="blue")
 lbl5=Label(master,text=t5, bg="orange")
```

**Abb. 16.2** Eine GUI mit einem Label und dem place-Layout-Manager





**Abb. 16.3** Nur ein Label ist zu sehen, und die Größe des Fensters wird beim Programmstart dem Label angepasst



**Abb. 16.4** Das Fenster wurde durch den Anwender vergrößert

```
Frame.__init__(self, master)
lbl1.pack()

root=Tk()
app=Application(master=root)
app.mainloop()
```

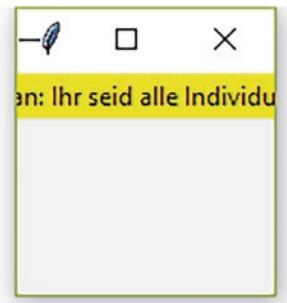
Sie sehen, dass in dem Quellcode fünf Stringvariablen mit Text angelegt werden. Aus diesen werden fünf Label erzeugt. Aber nur eines davon wird auch auf der Oberfläche mit der `pack()`-Methode platziert. Nach dem Programmstart wird die Größe des Fensters dann genau so angepasst, dass das Label vollständig zu sehen ist, aber kein weiterer Raum (Abb. 16.3).

Da die Größe des Fensters nicht fixiert wurde,<sup>1</sup> kann der Anwender jetzt aber dynamisch die Fenstergröße ändern, dann wird entweder ein weiterer Teil des Fensters ohne weiteren Inhalt zu sehen sein (Abb. 16.4), oder ein Teil des Labels verdeckt (Abb. 16.5). Der Layout-Manager legt also erst einmal nur einen Anfangszustand für die Größe des Fensters aufgrund des Inhalts fest. Aber Sie können an der Hintergrundfarbe des Labels gut erkennen, dass der Raum der eigentlichen Label-Komponente ebenso genau auf den Platz optimiert wird, den der Inhalt vorgibt. Das ist auch das Resultat des verwendeten `pack`-Layout-Managers.

---

<sup>1</sup> Das kann man mit Optionen machen.

**Abb. 16.5** Ein Teil des Labels wurde verdeckt



So ganz nebenbei haben Sie bei dem Beispiel auch gesehen, wie Sie mit den Attributen *bg* und *fg* die **Hintergrundfarbe** und die **Vordergrundfarbe** von GUI-Elementen festlegen können.

Richtig spannend wird es, wenn wir nun auch die anderen Label auf der Oberfläche anzeigen. Das folgende Listing zeigt nur die Stellen, an denen Änderungen gegenüber dem vorherigen Code auftreten – der Rest bleibt gleich (*GUI5.py*).

```
...
 Frame.__init__(self, master)
 lbl1.pack()
 lbl2.pack()
 lbl3.pack()
 lbl4.pack()
 lbl5.pack()
...

```

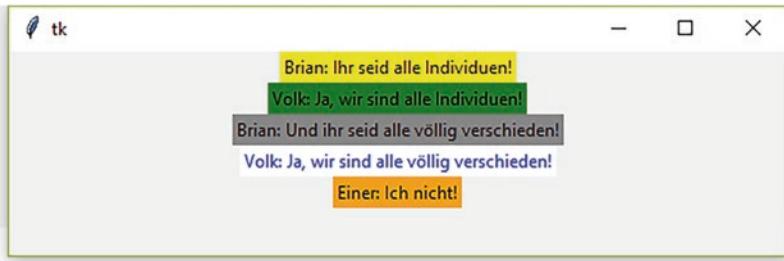
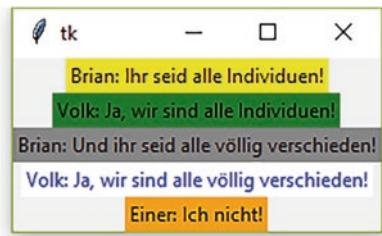
Sie erkennen nach dem Start wieder, dass die Fenstergröße so optimiert wurde, dass alle Label zu sehen sind (Abb. 16.6).

Durch Änderung der Fenstergröße kann man über die Hintergrundfarben gut erkennen, wie sich die Label verhalten. In der Grundeinstellung werden diese zentriert (Abb. 16.7).

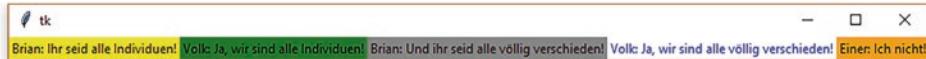
Die Anordnung im umgebenden Container kann man aber anpassen (Abb. 16.8). Dazu gibt es das Attribut *side*, das in dem Beispiel *GUI6.py* verwendet wird.

```
...
 Frame.__init__(self, master)
 lbl1.pack(side=LEFT)
 lbl2.pack(side=LEFT)
```

**Abb. 16.6** Der Zustand nach dem Programmstart



**Abb. 16.7** Das Fenster wurde vergrößert



**Abb. 16.8** Ein fließendes Layout mit Positionierung links

```
lbl3.pack(side=LEFT)
lbl4.pack(side=LEFT)
lbl5.pack(side=LEFT)
...

```

- ▶ Natürlich können Sie mit `side=RIGHT` auch ein rechtsfließendes Layout spezifizieren.
- ▶ Nicht ganz unkritisch ist es, die Ausrichtungen zu mischen. Es kann zu schwierig kontrollierbaren Verhaltensweisen kommen.

#### 16.3.3.2.3 Das Gitterlayout mit `grid`

Der Layout-Manager `grid` gibt ein Gitterlayout (eine Tabelle) an. Dabei geben Sie im Wesentlichen mit den Attributen `row` die nullindizierte Zeile und mit `column` die eben-

falls nullindizierte Spalte an, wo eine Komponente zu sehen sein wird. Wenn eine der beiden Angaben fehlt, wird dafür der Vorgabewert 0 verwendet. Der Layout-Manager errechnet aus den belegten Zellen die Größe des Gitters und den benötigten Platz. Beispiel (*GUI7.py*):

```
...
 Frame.__init__(self, master)
 lbl1.grid(row=0, column=0)
 lbl2.grid(row=0, column=1)
 lbl3.grid(row=1, column=1)
 lbl4.grid(row=1, column=0)
 lbl5.grid(row=2, column=1)
...

```

In dem Beispiel zeigen wir die Label von oben in einem Gitter (Abb. 16.9). Beachten Sie, dass sich gerade solche Gitter hervorragend dafür eignen, um Komponenten an beliebigen Positionen anzuzeigen.

#### 16.3.4 Wichtige GUI-Elemente

Schauen wir uns nun noch ganz kompakt ein paar wichtige GUI-Elemente an, die Sie üblicherweise in Oberflächen verwenden werden.

- Mit der Klasse *Button* werden **Schaltflächen** erstellt.
- Die Klasse *Entry* dient zum Erstellen von **Eingabefeldern**.
- Die Klasse *Label* erzeugt Textfelder zur **Beschriftung**.
- Mit *Text* hat man einen freien **Textbereich** zur Verfügung, der bei Bedarf auch Inhalte scrollbar darstellt.

Bei der Instanziierung werden die Eigenschaften als Parameter mit Wertzuweisung angegeben, die voreingestellt sein sollen.



Abb. 16.9 Ein Gitter zur Anordnung

- 
- Man kann auch nachträglich die Werte der Eigenschaften von Komponenten zuweisen. Das geht auf mehrere Arten. Eine oft genutzte Weise arbeitet mit der Methode `configure()`, um damit die Eigenschaften zu ändern.

---

## 16.4 Die Ereignisbehandlung

Oberflächenprogrammierung umfasst auch die Reaktion auf Benutzeraktionen und einige andere Ereignisse. Wenn ein Anwender etwa auf eine Schaltfläche klickt, muss im Programm hinterlegt sein, was daraufhin zu tun ist. Dabei gibt es verschiedene Modelle, wie auf Ereignisse zu reagieren ist. Aber die meisten modernen Programmiersprachen fokussieren sich auf ein ähnliches Konzept.

Wenn Ereignisse entstehen, sind das Mitteilungsobjekte. Und die müssen irgendwie ausgewertet werden. Dazu gibt es entsprechend aufgebaute Mechanismen, über die eine Reaktion des Programms bewirkt wird. Das Auftreten eines Ereignisobjekts selbst ist also noch keine Reaktion des Programms. Es muss zu einem Auswertungsobjekt transportiert werden.

Dies funktioniert schematisch so:

- Ein Ereignis tritt bei einem Quellobjekt (dem sogenannten Event Source) auf, etwa einem Button oder einem Menüeintrag.
- Das entstandene Ereignisobjekt wird an ein Zuhörerobjekt (das nennt man in manchen Konzepten einen Event Listener oder kurz Listener und in anderen Konzepten einen Event Handler beziehungsweise kurz Handler) weitergeleitet.
- Erst dort wird über die konkrete Ereignisbehandlung entschieden und die Reaktion auch ausgelöst.

### 16.4.1 Die konkrete Ereignisbehandlung in Python

Für die Ereignisbehandlung in Python gibt es die `command`-Anweisung. Diese kann entweder als Attribut beim Instanziieren eines GUI-Objekts wie ein Button angegeben werden.

Beispiel:

```
Button(master,text='OK',width=20,command=self.action)
```

Oder man gibt die Referenz auf eine Methode als Wertzuweisung der Eigenschaft an.

Beispiel:

```
self.okButton['command']=self.action
```

Die Referenz auf eine konkrete Aktion steht in dem Code über das Attribut `action` bereit.

Hier ist nur ein praktisches Beispiel mit dem *grid*-Layout und einer Ereignisbehandlung (*GUI8.py*):

```
from tkinter import *
class Application(Frame):
 def __init__(self, master=None):
 Frame.__init__(self, master)
 Label(master, text="Nachname").grid(row=0)
 Label(master, text="Vorname").grid(row=1)
 Label(master, text="Die Eingabe").grid(row=2)
 self.lbl1=Label(master, bg="yellow", fg="blue")
 self.lbl1.grid(row=2, column=1)
 self.nname = Entry(master)
 self.vname = Entry(master)
 Button(master, text='OK', width=20, command=self.action).grid(
 row=3, column=0)
 Button(master, text='Abbrechen', width=20, command=root.de-
stroy).grid(
 row=3, column=1)
 self.nname.grid(row=0, column=1)
 self.vname.grid(row=1, column=1)
 def action(self):
 self.lbl1.config(text = self.vname.get() + ", " + self.nname.
get())
root=Tk()
app=Application(master=root)
app.mainloop()
```

Wenn das Programm startet, zeigt die Oberfläche zwei Eingabefelder an, in denen ein Anwender Text eintragen kann (Abb. 16.10). Beim Klick auf die OK-Schaltfläche werden die Eingaben dann in einem Label angezeigt (Abb. 16.11).

Da wir ein Gitter-Layout haben, wird wieder mit dem Attribut *row* die Zeile angegeben, in der ein Element angeordnet werden soll, und mit *column* die Spalte. Die

**Abb. 16.10** Die Eingaben werden vorgenommen



**Abb. 16.11** Der OK-Button wurde angeklickt



Ereignisbehandlung wird mit dem *command*-Attribut bei den Schaltflächen implementiert. Sie sehen da einmal eine Referenz auf die Methode *action()*, in der mit der Methode *get()* der Inhalt der beiden Texteingabefelder abgefragt und dieser mit der *config()*-Methode bei dem Label *lbl1* angezeigt wird.

Bei dem zweiten Button wird mit der *destroy()*-Methode das Wurzelement der grafischen Anwendung zerstört und damit das Programm beendet, denn die *mainloop* wurde ja dem Fenster übertragen.

### 16.4.2 Lambda-Ausdrücke verwenden

Bei der Ereignisbehandlung unter *tkinter* bietet es sich an, dass Sie mit Lambda-Ausdrücken arbeiten. Denn diese liefern ja Funktionsreferenzen zurück. Betrachten wir diesen Quellcode, der das vorherige Beispiel so umwandelt, dass die Ereignisbehandlung mit Lambda-Ausdrücken erfolgt. Der Rest bleibt wie gehabt. Hier ist der geänderte Code, soweit er sich unterscheidet (*GUI9.py*):

#### Beispiel

```
...
Button(master, text='OK', width=20, command=lambda: self.action())
grid(row=3, column=0)
...
◀
```

Der *command*-Parameter für den OK-Button wurde geändert, um eine Lambda-Funktion zu verwenden, welche dann die *action*-Methode aufruft. Dadurch wird die Methode aufgerufen, wenn der Button geklickt wird.

Hier ist noch ein Beispiel, das mehrere Lambda-Ausdrücke in *tkinter* verwendet und auch Parameter übergibt. In diesem Beispiel werden drei Buttons erstellt, von denen jeder einen anderen Lambda-Ausdruck für seine Ereignisbehandlung verwendet (*GUI10.py*):

**Beispiel**

```
import tkinter as tk
from tkinter import messagebox
Erstellen des Hauptfensters
root = tk.Tk()
root.title("Mehrere Lambda-Ausdrücke Beispiel")

Funktionen für die Ereignisbehandlung
def show_message(message):
 messagebox.showinfo("Nachricht", message)
Erstellen von drei Buttons mit unterschiedlichen Lambda-Ausdrücken
button1 = tk.Button(root, text="Button 1",
 command=lambda: show_message("Button 1 wurde geklickt"))
button1.pack(pady=10)
x = 2
button2 = tk.Button(root, text="Button 2",
 command=lambda : show_message(f"Button {x} wurde geklickt"))
button2.pack(pady=10)
y = 5
button3 = tk.Button(root, text="Button 3",
 command=lambda: show_message(f"Das Ergebnis von {x} * {y} ist {x * y}"))
button3.pack(pady=10)
Starten der tkinter Hauptloop
root.mainloop() ◀
```

In diesem Beispiel erstellen wir drei Buttons, und für jeden Button verwenden wir einen eigenen Lambda-Ausdruck, um die *show\_message*-Funktion mit einer anderen Nachricht aufzurufen, wenn der Button geklickt wird. Jeder Button zeigt über das *tkinter*-Objekt *messagebox* und dessen Methode *showinfo* eine andere Meldung an, wenn er geklickt wird.

Der erste Lambda-Ausdruck gibt einfach eine hartcodierte Meldung aus, der zweite verwendet den Wert der Variable *x* aus dem übergeordneten Kontext. Der dritte Ausdruck verwendet dazu noch eine weitere Variable *y*.

---

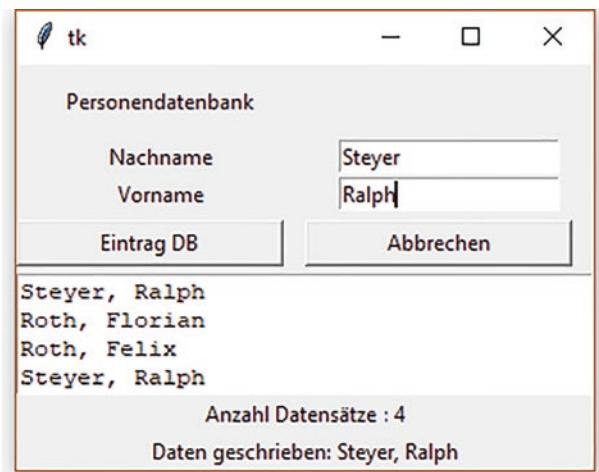
## 16.5 Eine grafische Datenbankapplikation

Aufbauend auf den Ausführungen zur GUI-Technologie mit Python, dem Umgang mit der SQLite-Datenbank und der allgemeinen OOP soll nun zum Abschluss unseres Buches zu Python die etwas umfangreichere folgende Aufgabe umgesetzt werden.

Wir erstellen ein Programm *Datenbankgui.py* mit einer grafischen Eingabemöglichkeit, um den Namen und Vornamen einer Person zu erfassen und in einer SQLite-Datenbank zu speichern. Die Datenstruktur soll so sein, wie sie im Abschnitt zu Datenbanken für die Erfassung einer Person erstellt wurde, also Nachname und Vorname werden gespeichert. Dazu machen wir noch Folgendes:

- Die Datenbankfunktionalität wird in ein Modul *datenbank.py* in einem Paket *rjs* ausgelagert.
- Das Modul *datenbank.py* enthält eine Klasse *DB* mit drei Methoden:
  - Die Methoden *initDB()* dient zum Anlegen der Datenbank, wenn sie noch nicht vorhanden ist. Ein String soll Protokollmeldungen zurückgeben (Abb. 16.12).
  - Die Methode *leseDB()* wird zum Auslesen der Werte in der Datenbank verwendet. Ein String soll im Fehlerfall eine Protokollfehlermeldung liefern. Der Rückgabetyp ist im Erfolgsfall ein sequenzieller Datentyp mit zwei Einträgen. Ein String liefert alle anzugegenden Daten sowie ein zweiter Wert die Anzahl der Datensätze (Abb. 16.12).
  - Die Methode *schreibDB()* zum Schreiben der Werte in die Datenbank. Ein String soll im Fehlerfall eine Protokollfehlermeldung und im Erfolgsfall die geschriebenen Werte zurückgeben.
- Das grundsätzliche Layout soll wie in Abb. 16.12 aussehen und auf dem *grid*-Layout beruhen:
  - Sie haben oben eine Beschriftung „Personendatenbank“ – ein Label.
  - Darunter folgen jeweils ein Label und ein Eingabefeld von Typ *Entry* für den Nachnamen und den Vornamen.

**Abb. 16.12** Der grafische Datenbank-Client



- Darunter befinden sich zwei Schaltflächen.  
Mit dem ersten Button wird ein Eintrag in die Datenbank ausgelöst. Es sollen die Eingaben in den Eingabefeldern übernommen werden.  
Mit dem zweiten Button wird die Applikation beendet.
- Die Anzeige der Datensätze erfolgt in einem Feld vom Typ *Text*.  
Das Feld darf nicht direkt durch den Anwender zu verändern sein. Mit der Methode *configure(state='disabled')* können Sie das einstellen.  
Mit *configure(state='normal')* erlauben Sie die Veränderung des Felds. Das muss eingestellt werden, wenn Daten aus der Datenbank in das Feld geschrieben werden sollen.  
Danach wird der Status wieder zurückgesetzt.
- Unterhalb des Anzeigebereichs der Daten sind zwei Label zur Anzeige von Statusinformationen angeordnet.
- Bei Start der Applikation soll eine Datenbank erstellt werden, wenn es noch keine gibt.
- Die Anzahl der aktuellen Datensätze soll in dem ersten Statuslabel angezeigt werden.
- Im zweiten Statuslabel sollen Datenbankaktionen protokolliert werden.
- Ist die Datenbank schon vorhanden, sollen beim Start alle vorhandenen Datensätze samt deren Anzahl angezeigt werden.
- Wird ein Datensatz geschrieben, soll die Anzeige aktualisiert und eine passende Statusmeldung angezeigt werden. Außerdem muss die Anzahl der vorhandenen Datensätze aktualisiert werden.

Hier wäre eine mögliche Lösung:

```
from rjs.datenbank import *
from tkinter import *
class Application(Frame):
 def __init__(self, master=None):
 Frame.__init__(self, master)
 self.grid(padx=20, pady=20)
 Label(master, text="Personendatenbank").grid(row=0, column=0)
 Label(master, text="").grid(row=0, column=1)
 Label(master, text="Nachname").grid(row=1)
 Label(master, text="Vorname").grid(row=2)
 self.nname = Entry(master)
 self.vname = Entry(master)
 Button(master, text='Eintrag DB', width=20, command=self.action).grid(
 row=3, column=0, sticky=W, pady=4)
 Button(master, text='Abbrechen', width=20, command=root.destroy).grid(
 row=3, column=1, sticky=W, pady=4)
```

```
self.nname.grid(row=1, column=1)
self.vname.grid(row=2, column=1)
self.anzeige = Text(master, height=4, width=40)
self.status=Label(master,text="")
self.status.grid(row=6, columnspan=2)
db = DB()
self.status['text']=db.initDB()
result = db.leseDB()
self.anzeige.insert(END, result[0])
self.anzeige.grid(row=4, columnspan=2)
self.anzeige.configure(state='disabled')
self.datensaetze=Label(master,text="")
self.datensaetze.grid(row=5, columnspan=2)
self.datensaetze['text'] = "Anzahl Datensätze : " + result[1]
def action(self):
 db = DB()
 self.status['text'] =
 db.schreibDB(self.nname.get(),self.vname.get())
 self.anzeige.configure(state='normal')
 result = db.leseDB()
 self.anzeige.delete(1.0,END)
 self.anzeige.insert(END, result[0])
 self.anzeige.configure(state='disabled')
 self.datensaetze['text'] = "Anzahl Datensätzer: " + result[1]
root=Tk()
app=Application(master=root)
app.mainloop()
```

Das ist das Datenbankmodul (*datenbank.py*):

```
import sqlite3
import os.path
class DB:
 def initDB(self):
 if not os.path.exists('sqldb.db'):
 connection=sqlite3.connect('sqldb.db')
 cursor=connection.cursor()
 cursor.execute(
 '''CREATE TABLE personen(name TEXT, vorname TEXT) ''')
 return "Datenbank erstellt"
 else:
 return "Datenbank vorhanden"
 def leseDB(self):
 dbString=""
 counter = 0
```

```

if os.path.exists('sqldb.db'):
 connection=sqlite3.connect('sqldb.db')
 cursor=connection.cursor()
 cursor.execute('''SELECT * FROM personen''')
 rows=cursor.fetchall()
 for row in rows:
 counter += 1
 dbString += row[0] + ", " + row[1] + "\n"
 connection.close()
 return [dbString,str(counter)]
else:
 return "Datenbank nicht vorhanden"
return dbString
def schreibDB(self,n,v):
 if os.path.exists('sqldb.db'):
 connection=sqlite3.connect('sqldb.db')
 cursor=connection.cursor()
 cursor.execute(
 '''INSERT INTO personen VALUES (?,?)''', (n,v))
 connection.commit()
 connection.close()
 return "Daten geschrieben: " + n + ", " + v
 else:
 return "Datenbank nicht vorhanden"

```

---

## 16.6 PyQt als Alternative

Wenngleich *tkinter* als Standard im Python-API bereits vorhanden ist, gibt es diverse Alternativen zur Erstellung grafischer Oberflächen mit Python. Gleich mehrere davon bauen auf Qt (ausgesprochen wie engl. *cute* – „niedlich“) auf. Die Qt-Bibliothek, die ursprünglich in und für C++ entwickelt wurde, bezeichnet eine kostenlose und quellöffentliche plattformübergreifende Software zum Erstellen grafischer Benutzeroberflächen sowie plattformübergreifender Anwendungen, die auf verschiedenen Software- und Hardwareplattformen laufen, z. B. Linux, Windows, macOS, Android oder eingebettete Systeme, wobei sich die zugrunde liegende Codebasis kaum oder gar nicht ändert und es sich dennoch um eine native Anwendung mit nativen Funktionen und Geschwindigkeit handelt. Die meisten mit Qt erstellten GUI-Programme verfügen über eine nativ aussehende Oberfläche. In diesem Fall wird Qt als Widget-Toolkit klassifiziert.

Wir wollen als Alternative zu *tkinter* nun eine solche Qt-Erweiterung betrachten – PyQt. Dies ist eine Python-Bibliothek mit Widgets, Layouts und Tools zur Gestaltung und Interaktion mit grafischen Benutzeroberflächen.

Da PyQt allerdings nicht mit Python direkt ausgeliefert wird, musst PyQt nachträglich installiert werden. Dies kann mit dem Paketmanager *pip* erfolgen, wenn man die Version 5 nimmt:

```
pip install PyQt5
```

Zur praktischen Anwendung wollen wir zuerst ein einfaches PyQt-Fenster erstellen (*PyQt1.py*):

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow
app = QApplication(sys.argv)
window = QMainWindow()
window.show()
sys.exit(app.exec_())
```

In diesem Beispiel wird ein PyQt-Anwendungsfenster erstellt und angezeigt. *QApplication* ist die Anwendungsklasse, die den Anwendungsstatus und die Ereignisschleife verwaltet. *QMainWindow* hingegen ist ein leeres Hauptfenster, mit dessen *show()*-Methode wird das Fenster angezeigt.

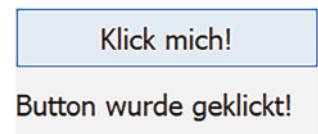
Nun braucht eine GUI natürlich Widgets (z. B. Buttons, Labels, Textfelder), um mit einem Anwender zu interagieren. Die nachfolgende Erweiterung fügt dem Hauptfenster ein einfaches Label hinzu (*PyQt2.py*):

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel
app = QApplication(sys.argv)
window = QMainWindow()
label = QLabel("Hallo Welt!")
window.setCentralWidget(label)
window.show()
sys.exit(app.exec_())
```

In diesem Beispiel wird ein *QLabel*-Widget mit dem Text "Hallo Welt!" erstellt und dem Hauptfenster mit der Methode *setCentralWidget()* hinzugefügt. Dabei erkennen Sie, dass auch PyQt einen Manager verwendet, um die Gestaltung der GUI zu verwalten (wie *tkinter*).

Nun fehlt aber noch die Interaktion mit Anwendern. PyQt ermöglicht es, Benutzerinteraktionen mit sogenannten „Signalen“ und „Slots“ zu verknüpfen. Wenn ein Ereignis (Signal) auftritt, wird eine Funktion (Slot) ausgeführt. Hier ist ein Beispiel mit einem Button (*PyQt3.py*):

**Abb. 16.13** Eine Oberfläche unter PyQt



```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QPushButton
def button_clicked():
 label.setText("Button wurde geklickt!")
app = QApplication(sys.argv)
window = QMainWindow()
button = QPushButton("Klick mich!")
button.clicked.connect(button_clicked)
label = QLabel("Hallo Welt!")
window.setCentralWidget(label)
window.setMenuWidget(button)
window.show()
sys.exit(app.exec_())
```

In diesem Beispiel wird ein Button erstellt, und wenn dieser geklickt wird, ändert sich der Text des Labels (Abb. 16.13). Die Verbindung erfolgt mit der Methode `connect()`, der eine Funktionsreferenz übergeben wird. Sie können hier auch Lambda-Ausdrücke notieren.

Dies soll jetzt nur eine Art „Schnupperkurs“ in PyQt sein, und Sie erkennen einige Ähnlichkeiten zu `tkinter`, aber auch abweichende Konzepte. PyQt bietet auf jeden Fall eine umfangreiche Sammlung von Widgets und Funktionen zur Erstellung von leistungsfähigen und benutzerfreundlichen Desktop-Anwendungen.

---

# Stichwortverzeichnis

## A

Abbruchbedingung

Rekursion, 125

abc.py, 212

abc, 212

Abgeleitete Klasse, 206

abs(), 54, 78

Absolutwert, 54

Abstract Base Class s. abc

Abstrakte Klasse, 212

Abstraktion, 175

add(), 157, 158

Addition, 85

\_\_add\_\_, 180, 212

Add Python.exe to PATH, 10

Ad-hoc-Konvertierung, 80

Aggregation, 176

AI s. KI

Amoeba, 5

Anaconda, 34

and, 91

Anführungszeichen

Maskierung, 79

Anonyme Funktion, 129

anonyme Funktion, 200

Anweisung, 61

Ausdrucks-, 64

Block-, 62

Deklarations-, 64

Kontrollfluss-, 63

leere, 65

Anweisungsblock, 62, 104

Anzahl der Elemente, 143

API, 3

append(), 150

Application Programming Interface s. API

Argument, 44, 120

Arithmetischer Operator, 84

Arithmetischer Zuweisungsoperator, 88

Array, 138

  assoziatives, 152

  Byte-, 255

artificial intelligence s. KI

as, 229

assert, 232

AssertionError, 232

assignment copy, 203

Assoziation, 176

Attribut, 148, 176

  statisches, 183

AttributeError, 201

Ausdruck, 83

  benannter, 100

  regulärer, 237

Ausdrucksanweisung, 64

Ausdrucksbewertung, 101

Ausgabe, 249

Ausgabestrom, 43

Ausnahmebehandlung, 219

Ausnahmeklasse

  eigene, 232

Ausnahmeobjekt

  auswerten, 229

Ausnahme

  Reihenfolge, 227

  werfen, 220

Auswertung

  verkettete, 98

**B**

Backslash  
    Maskierung, 79  
Backspace  
    Maskierung, 79  
BaseException, 227  
bases-Attribut, 202  
Basisklasse, 206  
Beispiel  
    Download, 2  
Benannter Ausdruck, 100  
Beschriftung, 280  
Bezeichner, 57  
Binäre Darstellung  
    Text, 254  
Binäroperator, 93  
Bitweiser Operator, 93  
Blattklasse, 206  
Blockanweisung, 62, 104  
bool, 72  
bool(), 72  
Boolescher Operator, 89  
Boolescher Wert, 72  
Bootstrap, 38  
Bracket, 239  
    regulärer Ausdruck, 240  
    regulärer Ausdruck, 241  
break, 116, 223  
Built-in Function, 42  
Built-in Functions, 41  
Button, 280  
Byte-Array, 138, 255

**C**

C3 Linearization, 208  
C3 Superklassenalgorithmus, 208  
Caesar-Chiffre, 170  
calendar, 266  
Call-by-object, 121  
Call-by-reference, 120  
Call-by-sharing, 121  
Call-by-value, 120, 121  
CamelCase-Notation, 68  
Camelnotation, 68  
case-sensitive, 48  
Casting, 52, 70, 79, 80  
catch, 221  
class, 177

@classmethod, 190  
clear(), 154, 158  
close(), 251, 261  
Closure, 128  
Cloud, 33  
cmd, 22  
Codespace, 33  
collections.deque, 152  
column, 282  
command, 281  
commit(), 262  
compile(), 248  
Compiler, 2  
complex, 73  
Comprehension, 169  
conda, 35  
configure(), 281  
connect(), 261  
continue, 117  
copy, 203  
copy(), 153, 158, 256  
cursor(), 261

**D**

data stream, 249  
Datei  
    binär, 254  
    binär öffnen, 251  
Dateiende, 250  
Dateierweiterung  
    Python, 29  
Dateimodus, 250  
Dateizugriff, 249  
Daten, 67  
Datenbankapplikation  
    grafisch, 284  
Datenbankzugriff, 249, 260  
Datenkapselung, 182  
    Closures, 128  
Datenstrom, 249  
Datenträgerzugriff, 249  
Datentyp, 69  
datetime, 266  
datetime.now(), 266  
datetime.strptime(), 267  
Datum, 265  
Datumsobjekt erstellen, 267  
Debugging, 219

- decode, 254  
deepcopy, 202, 203  
def, 119  
Default-Konstruktor, 179  
Default-Parameter, 132  
Deklaration, 42  
    Funktionen, 119  
Deklarationsanweisung, 64  
del, 153  
    `_del__`, 197  
Deserialisieren  
    Objekte, 258  
destroy(), 283  
Destruktor, 177, 197  
Dezimalpunkt, 57  
Dezimalsystem, 53  
    `_dict__`, 184, 199  
`dict_keys`, 154, 155  
dict-Attribut, 202  
Dictionary, 152  
    Key-Sharing-, 201  
Dictionary Comprehension, 171  
difference(), 158  
difference\_update(), 158  
Direktive  
    Datum und Zeit, 267  
discard(), 158  
Division, 85  
Doppelter Unterstrich, 193  
Doppeltes Anführungszeichen, 79  
Dot-Notation, 148, 149  
Download  
    Python, 7  
Downstream, 249  
Dualsystem, 53  
dump(), 259
- E**  
Eclipse, 33  
Eigenschaft, 176  
    eines Objekts, 148  
Einfachvererbung, 206  
Eingabe, 249  
Eingabeaufforderung  
    Python, 23  
Eingabefeld, 280  
Eingabestrom, 51  
Einrücktiefe, 62
- Einrückung, 21, 62  
Element  
    Anzahl, 143  
    statisches, 176  
elif, 105  
else, 104  
    try ... except, 228  
Elternklasse, 206  
encode, 254  
Endlosschleife, 113  
    Beispiel, 117  
End of File, 250  
Entfernen, 150  
Entry, 280  
Entscheidungsanweisung, 104  
EOF, 250  
EOFError, 250  
Ereignisbehandlung, 281  
Eric Python IDE, 32  
Escape-Darstellung, 78  
Escape-Sequenz, 51  
eval(), 54  
Event Handler, 281  
Event Listener, 281  
except, 221  
Exception-Handling, 219  
Exception  
    Standard, 226  
execute(), 261  
exit(), 26  
Explizite Typumwandlung, 80  
Exponentialschreibweise, 75
- F**  
Fakultätsberechnung, 125  
False, 72  
fechtall(), 262  
Fehler  
    Laufzeit-, 219  
    syntaktischer, 219  
    typografischer, 219  
Fehlermeldung, 148  
Fehlersituation  
    Behandlung, 219  
Feld, 176  
FIFO, 151  
FileExistsError, 258  
FileNotFoundException, 258

- FILO, 125, 150  
optionaler, 132  
unerreichbarer, 134  
Finalisierungsaktion, 223  
finally, 223  
first in last out s. FILO  
float(), 77, 81  
float, 73  
Floating-Point-Zahl, 75  
format()  
    Strings, 235  
Formatierte Zeichenkette, 48  
Format Specification Mini-Language, 234  
Formfeed, 79  
Formularvorschub  
    Maskierung, 79  
for-Schleife, 115  
Frame, 273  
from, 215  
Frozenset, 157  
f-string, 48  
Funktion  
    anonyme, 129  
    aufrufen, 121  
    deklarieren, 119  
    innere, 128  
    Rückgabewert, 121  
    vorinstallierte, 42  
    was ist das, 41  
Funktionsaufruf  
    rekursiver, 124  
Funktionsparameter, 120  
Funktionsreferenz, 129
- G**  
Ganzzahldivision, 85, 95  
Garbage Collection, 197, 204  
Generalisieren, 176  
Geometry-Manager, 273  
get(), 153, 283  
Getter, 195  
Github, 33  
Gleitkommaarithmetik, 76  
Gleitkommaliteral, 75  
Gleitkommazahl, 75  
Gleitpunktzahl s. Gleitkommazahl  
Global, 126  
    Variable, 124
- Global Scope, 180  
gmtime(), 267  
Grafische Oberfläche, 271  
graphical user interface, 271  
grid, 275  
Größer als, 89  
Größer als oder gleich, 89  
group(), 244  
groups(), 246  
GUI, 271  
    Python, 11  
Guido van Rossum, V
- H**  
Handler, 281  
Hash, 152  
Haufen s. Heap  
Häufigkeit  
    regulärer Ausdruck, 241  
Hauptprogramm, 181  
Heap, 124  
help(), 26, 42  
Hervorhebung  
    Syntax, 45  
Hexadezimalsystem, 53  
Hilfemodus, 26  
Hintergrundfarbe  
    tkinter, 278  
Höckerschrift, 68
- I**  
Identifier, 57, 183  
Identitätsoperator, 92  
IDLE, 11, 31  
    Run, 59  
if, 104  
    verschachtelte Anweisung, 106  
Imaginäre Einheit, 76  
Imaginärteil, 76  
import, 214  
in, 92, 161  
    Membership-Operator, 140  
\_\_init\_\_, 179, 180  
    Paketdatei, 216  
\_\_init\_\_.py, 213  
Indexbereich, 145  
IndexError, 144, 145

Information Hiding, 182, 193  
inheritance, 206  
Innere Funktion, 128  
input(), 44, 50  
Installationsort  
    Python, 11  
Instanz, 148, 175  
Instanzeigenschaft, 183  
Instanzelement, 175  
int, 73  
int(), 52, 77, 81  
Integerliterale, 48  
IntelliJ, 33  
Interaktivmodus, 21  
Interface, 212  
Interpreter, 2  
    Python, 11  
intersection(), 158  
IOError, 250, 254  
.ipynb, 38  
IPython, 35  
is, 92  
isdisjoint(), 158  
is not, 92  
issubset(), 158  
issuperset(), 158  
Iterationsanweisung, 113  
Iterator, 164  
  
Klassenbaum, 206  
Klasseneigenschaft, 183, 186  
Klassenelement, 176  
Klassenmethode, 190  
Klassenobjekt, 190  
Kleiner als, 89  
Kleiner als oder gleich, 89  
Klonen, 203  
Komma, 57  
Kommandozeile, 11  
    Python, 21  
Kommentare, 58  
Komplexe Zahl, 76  
Komposition, 176  
Konstruktor, 177  
    parametrisierter, 179  
Kontextmanager, 253  
Kontrollflussanweisung, 63  
Kontrollstruktur, 103  
Konvertierungsfunktion, 81  
Kopie, 153  
Kopieren  
    durch Zuweisung, 203  
    Objekte, 202  
Künstliche Intelligenz s. KI

## J

Java, 174  
jQuery, 37  
JSON, 38, 153, 199  
Jupyter Notebook, 34

## K

Kapselung, 182  
KeyError, 153, 158  
keys(), 154, 156, 161  
Key-Sharing-Dictionary, 201  
Key-Value-Paar, 152  
keywords, 27  
Kindklasse, 206  
KI, V  
Klammer s. Bracket, 57  
Klasse, 148, 175, 177  
    abgeleitete, 206

## L

Label, 280  
lambda, 200  
Lambda-Ausdruck, 129  
lambda-Operator, 129  
Laufzeitfehler, 219  
Launcher  
    Python, 10  
Layout-Manager, 273  
leaf class, 206  
Leere Klasse, 178  
Leere Operation, 84  
Leerzeichen, 56  
len(), 143, 145, 146, 151, 164, 235  
Lesemodus  
    Datei, 250  
Lesezugriff, 195  
Library Reference, 217

Lisp, 173  
 List Comprehension, 170  
 Liste, 146  
     Länge, 143  
     Methoden, 148  
 Listener, 281  
 Listenmethoden, 148  
 Listing  
     Download, 2  
 Literal, 25, 57  
 Literale Erzeugung, 177  
 Lizenz  
     Python, 7  
 load(), 259  
 Logischer Operator, 90  
 Logisches Nicht, 91  
 Logisches Oder, 91  
 Logisches Und, 91  
 Logo, 173  
 Lokale Variable, 120  
 long(), 74  
 long integer literal, 74  
 loose typing, 70  
 Löschen  
     Objekt, 197  
 Lose Typisierung, 70  
 lowerCamelCase, 68

Differenz, 158  
 Schnittmenge, 158  
 messagebox  
     tkinter, 284  
 Metacharacter, 241  
 Metaklasse, 201  
 Methode, 149, 176  
     magische, 179  
     statische, 190  
 Method Resolution Order s. MRO  
 min(), 163  
 Mitgliedschaftsoperator, 92  
 mktime(), 267  
 Modul, 181, 213  
     random, 259  
     tkinter, 272  
 Modularisierung, 42  
 Modulo, 85  
     Gleitkommazahlen, 86  
 Modul re, 236  
 Monty Python, V  
 Monty Python's Flying Circus, V  
 move(), 256  
 MRO, 208  
     abstrakte, 212  
 mro(), 209  
 Multiplikation, 85  
 Multiplikationsoperator, 160

**M**

\_\_main\_\_, 181  
 mainloop, 272, 273  
 Main Program, 181  
 Map, 152  
 Mapping, 152  
 Markenrechte  
     Python, 7  
 Maschinencode, 2  
 Maskieren, 51, 241  
 match(), 247  
 match-case, 110  
 Match-Objekt  
     regulärer Ausdruck, 244  
 Mathematischer Operator, 84  
 MathJax, 38  
 max(), 163  
 Mehrfachvererbung, 206, 207  
 Membership-Operator, 92, 140, 155, 161  
 Menge, 157

**N**  
 \n, 51  
 \_\_name\_\_, 181  
 Name, 57  
 name-Attribut, 202  
 NameError, 122, 130, 133  
 Namensraum, 178  
 Negation, 85  
 NetBeans, 33  
 Neue Zeile (New line), 79  
 new, 179  
     \_\_new\_\_, 179, 180  
 NEWLINE, 64  
 New line  
     Maskierung, 79  
 Nicht  
     logisches, 91  
 None, 71  
 not, 91

Notepad++, 33

not in, 92, 161

## O

Oberklasse, 206

object

Klasse, 207

Objekt, 148, 175, 182

aktueller, 178

dynamisch erweitern, 199

löschen, 197

serialisieren und deserialisieren, 258

Objektorientierte Programmierung, 148

Objektorientierung

Kernkonzepte, 175

Objekttyp, 175, 177

oct(), 52

Oder

Logisches, 91

Oktaldarstellung, 52

Oktalsystem, 53

OOP, 148

open(), 250

Operand, 57, 83

Operation, 84, 87

leere, 84

Operator, 57, 83

arithmetischer, 84

Binär-, 93

bitweiser, 93

boolescher, 89

logischer, 90

mathematischer, 84

Membership-, 140

Plus-, 87

Prioritätenreihenfolge, 87

Verschiebungs-, 93

Walross-, 99

Zuweisungs-, 88

Zuweisungsausdruck, 100

Operatorassoziativität, 102

Operatorenpriorität, 101

Operatorvorrang, 101

Option

regulärer Ausdruck, 241

or, 91

os.path, 261

os.path.exists(), 262

other

match, 111

overloading, 211

Overriding, 210

## P

pack, 275

package, 4

Paket, 4, 213, 216

Paketverwaltung

conda, 35

Paketverwaltungssystem

pip, 4

Parameter, 44, 119

variable Anzahl, 132

Parametrisierter Konstruktor, 179

Parser, 55

PascalCase, 68

pass, 65, 106, 178

Path, 10

Pattern, 237

reguläre Ausdrücke, 238

Pattern-Matching, 110, 164

PermissionError, 258

Pflichtfeld, 238

pickle, 258

pickle.load(), 259

pip, 4, 10, 35, 218

place, 275

Plausibilisierung, 238

Plusoperator, 87, 160

Polymorphie, 176, 210

pop(), 150, 153, 158

popitem(), 153

Potenz, 85, 86

print(), 43, 44

Priorität

Operatoren, 101

Prioritätenreihenfolge

Operatoren, 87

private, 193

Programmfluss, 62, 103

Programmiersprache

strukturierte, 62

Programmierung

objektorientierte, 148

Prompt, 23

property, 196

property(), 196  
protected, 193  
Prozedur, 121  
PSF, 6  
public, 193  
Punkt, 57  
Punktnotation, 148, 149, 183, 184  
Punkt-vor-Strich-Regel, 84  
PyCharm, 32  
PyDev, 33  
pyinstall, 4  
PyPI, 4, 218  
 PyQt, 288  
Python  
    Dateierweiterung, 29  
    Download, 7  
    Eingabeaufforderung, 23  
    GUI, 11  
    Installationsort, 11  
    Kommandozeile, 11, 21  
    laden und installieren, 5  
    Lizenz, 7  
    Lizenz und Markenrechte, 7  
    Referenzversion, 6  
    Shell, 11  
Python 3, 5  
python3, 18  
Python-Editor  
    IDLE, 32  
Python Package Index s. PyPI, 3  
Python Shell, 21  
Python's Integrated Development Environment  
    s. IDLE  
Python Software Foundation s. PSF  
Python-Software-Foundation-Lizenz, 7

**Q**  
QApplication  
    PyQt, 289  
Qt, 288  
Quantifizierer  
    regulärer Ausdruck, 241  
Quellcode, 2, 29  
Quelltext, 29  
Queue, 151  
quit(), 26

**R**  
raise, 118, 230  
randint(), 259  
random  
    Modul, 259  
random.randint(), 259  
Range, 145  
Raw-String, 241  
re  
    Modul, 236, 241  
read(), 252  
readlines(), 253  
read-only, 195  
RecursionError, 125  
Reference Counting, 204  
Referenzbetriebssystem, 5  
Referenznummer  
    Objekte, 204  
Referenzübergabe, 120  
Referenzzyklus, 204  
Regex, 244  
Regulärer Ausdruck, 237  
regulärer Ausdruck  
    regulärer Ausdruck, 238  
regular expression s. Regulärer Ausdruck  
Reihenfolge  
    Ausnahmen, 227  
Rekursion, 124  
remove(), 158, 256  
Ressourcenleck, 253  
Restwertberechnung, 85  
return, 79, 117, 121  
RIA, 33  
Rich Internet Application s. RIA  
rmdir(), 256  
root class, 206  
round(), 76, 77  
row, 282  
rstrip(), 252  
Rückgabewert, 51, 121  
    bedingt, 122  
Rückschritt, 79  
Run, 46  
Rundungsproblem, 76, 85, 86  
    Gleitkommazahl, 76  
Run Module, 46

**S**

Save  
    IDLE, 46  
Save As  
    IDLE, 46  
Save Copy As  
    IDLE, 46  
Schaltfläche, 280  
Schleife, 113  
    verschachteln, 113  
Schlüssel-Objekt-Paar, 152  
Schlüsselwort, 27, 57  
Schnittstelle, 176, 212  
Schreibzugriff, 195  
search(), 247  
Selbstaufruf, 124  
self, 178, 184  
Semikolon  
    Python, 47  
sep  
    print(), 49  
Separator  
    print(), 49  
Sequenz, 61  
Serialisieren  
    Objekte, 258  
set, 157  
set(), 157  
Set Comprehension, 170  
Setter, 195  
shallow copy, 202  
SheBang, 58, 60  
Shell  
    Python, 11, 21  
Sichtbarkeit, 182, 193  
Slot, 200  
    \_\_slots\_\_, 201  
Smalltalk, 173  
Snake Case, 69  
Softwarequalität, 175  
Sourcecode, 29  
Spalte, 21  
Speicherleck, 205  
Speichermanagement, 197  
Spezialisierung, 176, 206  
split(), 246  
Sprunganweisung, 116, 223  
Spyder, 35  
SQL, 260  
SQLite, 260  
sqlite3, 261  
sqlite3.Error, 264  
SSH-Tunnel, 39  
Stack, 124, 150  
Standardausgabe, 250  
Standardausnahme, 226  
Standardeingabe, 250  
Standardmodul, 217  
Standard Query Language, 260  
Stapel s. Stack, 150  
@staticmethod, 191  
Statische Methode, 190  
Statisches Attribut, 183  
Statisches Element, 176  
stdin, 250  
stdout, 250  
    \_\_str\_\_, 198  
str(), 81, 87  
Stream, 249  
strftime(), 267  
String, 78  
string index out of range, 144  
String-Konstante, 234  
Stringliteral, 25  
String-Methode, 235  
String-Verkettung, 49, 160  
String-Verknüpfung, 87  
Strom (Daten), 249  
strptime(), 267  
Structured Query Language, 260  
Strukturierte Programmiersprache, 62  
    \_\_sub\_\_, 180, 212  
Subklasse, 206  
Subtraktion, 85  
Suchmuster s. Pattern  
Suchpfad, 10  
super(), 207, 208, 211  
Superklasse, 206  
switch-case s. match-case  
Syntaktischer Fehler, 219  
:=Syntax, 99  
Syntaxhervorhebung, 45  
Systemdatum, 266

**T**

Tabulatur  
    Maskierung, 79

- Terminierungsaktion, 223  
 Text, 280  
 Textbereich, 280  
 The Epoch, 265  
 this, 178  
 time, 266  
 time.gmtime(), 267  
 time.mktime(), 267  
 time.strftime(), 267  
 Tk, 272  
 tkinter  
   Lambda-Ausdruck, 283  
   Modul, 272  
 Token, 55, 84  
 Top-Level-Script-Environment, 180  
 Tornado, 37  
 Traceback, 148  
 Trennzeichen, 56  
 True, 72  
 try, 221  
 Tupel, 139  
   verschachteltes, 139  
 tuple index out of range, 145  
 Typ s. Datentyp  
 type(), 71, 79, 189, 201  
 TypeError, 131, 132  
 Typisierung, 70  
   lose, 70  
 Typkonvertierung, 80  
 Typografischer Fehler, 219  
 Typsicher, 71  
 Typumwandlung, 52, 70, 79  
   explizite, 80  
 Pattern-Matching, 166
- U**  
 Übereinstimmungsobjekt, 244  
 Übergabewert, 44, 120  
 Überladen, 210, 211  
 Überschreiben, 210  
 Übersetzung  
   zweistufige, 3  
 UnboundLocalError, 133  
 Und  
   logisches, 91  
 undefined  
   anonyme, 200  
 Under\_score-Notation, 69
- Unerreichbarer Code, 134, 228  
 Ungleichheit, 89  
 Unit-Testing, 232  
 Unix-Zeit, 265  
 unreachable code, 134, 228  
 Unterklasse, 206  
 Unterprogramm, 42  
 Unterstrich  
   doppelter, 193  
   einfacher, 193  
 Untitled, 46  
 update(), 154  
 UpperCamelCase, 68  
 Upstream, 249  
 UTF-8, 255
- V**  
 ValueError, 221, 224  
 values(), 156, 161  
   Dictionary, 153, 155  
 van Rossum, Guido, V, 5  
 Variable, 49, 67  
   lokale, 120, 130  
   verdeckte, 131  
 Verallgemeinerung, 206  
 Verdeckte Variable, 131  
 Vererbung, 176, 205  
 Vergleichsoperator, 89  
 Verkettung  
   Strings, 87  
 Verschachtelte if-Anweisung, 106  
 Verschachtelte Schleifen, 113  
 Verschiebungsoperator, 93  
 Vielgestaltigkeit, 176  
 Visual Studio, 33  
 Visual Studio Code, 33  
 Voraussetzungen für Python, 4  
 Vordergrundfarbe  
   tkinter, 278  
 Vorgabewert  
   Parameter, 132
- W**  
 Wagenrücklauf, 79  
 Walross-Operator, 99, 171  
 Warteschlange, 151  
 Wertübergabe, 120

- where, 10
- which, 18
- while-Schleife, 113
- Widget-Toolkit, 288
- Wiederverwendbarkeit, 174
- Wildcard
  - Pattern-Matching, 111, 165
- winreg, 218
- with, 253
- write(), 251
- write-only, 195
- Wurzelberechnung, 86
- Wurzelklasse, 206
  
- Z**
- Zahlensystem, 53
- Zählvariable, 113
- Zeichenkette, 78
  - formatierte, 48
- sequenzielle Datentypen, 138
- Zeichenliteral, 78
- Zeile
  - logische, 64
- Zeilenumbruch, 51
- Zeit, 265
- Zeitnullpunkt, 265
- ZeroDivisionError, 224
- ZeroMQ, 37
- Zufallszahl, 259
- Zustand, 183
- Zuweisungsausdruck, 100
- Zuweisungsausdrucksoperator, 100
- Zuweisungsoperator, 67, 88
- Zweig
  - Entscheidungsanweisungen, 103
- Zweistufige Übersetzung, 3
- Zwischencode, 3