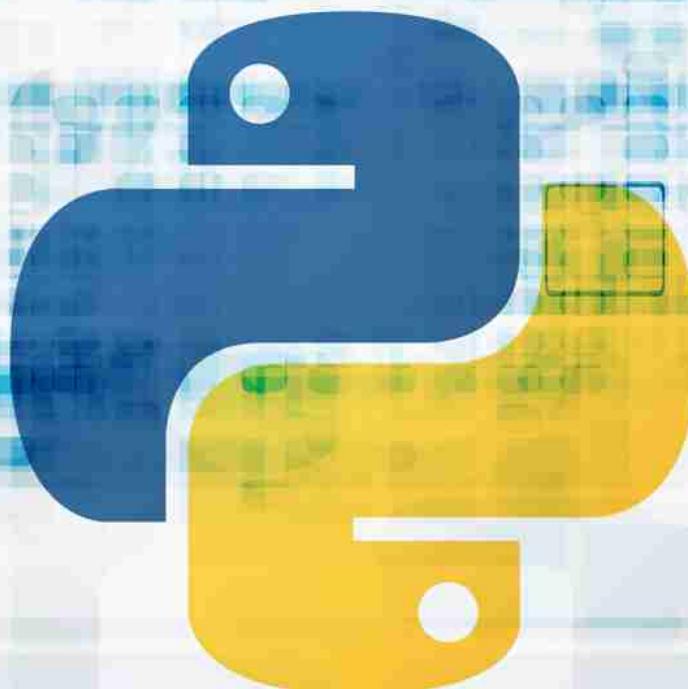


# PYTHON KOMPENDIUM

Professionell Python Programmieren lernen



Ohne  
Vorwissen  
loslegen

Eine umfassende Einführung in Python!

- Alle Grundlagen der Programmierung verständlich erklärt
- GUIs, Datenbanken, Parallele Programmierung, Datenanalyse u.v.m.
- Übungsaufgaben mit Musterlösungen nach jedem Kapitel



Inklusive eBook zum Download

{BMU VERLAG}

# **Python-Kompendium**

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Informationen sind im Internet über <http://dnb.d-nb.de> abrufbar.

©2021 BMU Media GmbH

[www.bmu-verlag.de](http://www.bmu-verlag.de)

[info@bmu-verlag.de](mailto:info@bmu-verlag.de)

Einbandgestaltung: Pro ebookcovers Angie

Druck und Bindung: Wydawnictwo Poligraf sp. zo.o. (Polen)

Taschenbuch-ISBN: 978-3-96645-021-8

Hardcover-ISBN: 978-3-96645-049-2

E-Book-ISBN: 978-3-96645-020-1

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte (Übersetzung, Nachdruck und Vervielfältigung) vorbehalten. Kein Teil des Werks darf ohne schriftliche Genehmigung des Verlags in irgendeiner Form – auch nicht für Zwecke der Unterrichtsgestaltung – reproduziert, verarbeitet, vervielfältigt oder verbreitet werden.

Dieses Buch wurde mit größter Sorgfalt erstellt, ungeachtet dessen können weder Verlag noch Autorin, Herausgeber oder Übersetzer für mögliche Fehler und deren Folgen eine juristische Verantwortung oder irgendeine Haftung übernehmen. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären.

# Python-Kompendium

*Ein umfassender Einstieg in die Programmierung  
mit Python 3*

Sarah Schmitt

# Inhaltsverzeichnis

<b>1. Einleitung</b>	<b>9</b>
1.1 Über dieses Buch.....	10
1.2 Python: eine einfach zu erlernende Programmiersprache.....	12
1.3 Die Entwicklungsgeschichte .....	13
1.4 Die Ziele bei der Entwicklung von Python .....	14
1.5 Interpretiert vs. kompiliert.....	16
<b>2. Die Vorbereitung</b>	<b>18</b>
2.1 Die integrierte Entwicklungsumgebung.....	18
2.2 Den Python-Interpreter installieren .....	19
2.2.1 Unter Windows installieren.....	20
2.2.2 Unter Linux installieren.....	22
2.2.3 Unter macOS installieren .....	22
2.3 PyCharm – Eigenschaften und Installation.....	22
2.3.1 PyCharm: Installation unter Windows.....	24
2.3.2 PyCharm: Installation unter Linux.....	25
2.3.3 PyCharm: Installation unter macOS .....	26
2.4 PyCharm anpassen.....	27
<b>3. Der interaktive Modus: ideal für den ersten Kontakt mit Python</b>	<b>30</b>
3.1 Den Python-Prompt aufrufen.....	30
3.2 Erste Befehle im Python-Prompt.....	31
<b>4. Python-Programme in eine Datei schreiben</b>	<b>35</b>
4.1 Ein Programm für eine einfache Textausgabe erstellen .....	35
4.1.1 PEP – Guidelines zum Layout und Styling von Code .....	37
4.2 Die Ausführung im Python-Interpreter .....	38
4.3 Kommentare: hilfreich für das Verständnis des Programms.....	39
4.4 Übung: Eigene Inhalte zum Programm hinzufügen .....	42
<b>5. Variablen: unverzichtbar für die Programmierung mit Python</b>	<b>47</b>
5.1 Die Aufgabe von Variablen .....	47
5.2 Variablen in Python verwenden.....	48
5.3 Den Wert einer Variablen durch eine Nutzereingabe festlegen .....	52
5.4 Dynamische Typisierung: viele Freiheiten bei der Nutzung von Variablen .....	55
5.5 Datentypen sind auch in Python von Bedeutung .....	58
5.6 Übung: Mit Variablen arbeiten .....	62
<b>6. Datenstrukturen in Python</b>	<b>65</b>
6.1 Datenstrukturen: praktische Methoden zur Datenerfassung .....	65
6.2 Listen: mehrere Informationen zusammenfassen .....	66
6.3 Dictionaries: Zugriff über einen Schlüsselbegriff.....	72
6.4 Tupel: unveränderliche Daten.....	75
6.5 Mit Mengen arbeiten .....	78
6.6 Weiterführendes über Strings .....	82

6.7 Operatoren in Python .....	87
Arithmetische Operatoren .....	87
Vergleichsoperatoren .....	88
Boolesche Operatoren .....	90
Zugehörigkeitsoperatoren .....	92
6.8 Übung: Mit unterschiedlichen Datenstrukturen arbeiten .....	93
<b>7. Entscheidungen treffen – Programmabläufe steuern</b>	<b>100</b>
7.1 Der Schlüsselbegriff if.....	100
7.2 Vergleiche: wichtig für das Aufstellen der Bedingung .....	101
7.3 Mehrere Bedingungen verknüpfen.....	104
7.4 else und elif: weitere Alternativen hinzufügen .....	105
7.4.1 else.....	106
7.4.1 elif.....	108
7.5 Übung: Eigene Abfragen erstellen.....	110
<b>8. Schleifen: Programmteile wiederholen</b>	<b>115</b>
8.1 Die while-Schleife: bedingte Wiederholungen .....	115
8.2 Die for-Schleife: elementweises Iterieren .....	118
8.3 break und continue: weitere Werkzeuge für die Steuerung von Schleifen.....	121
8.3.1 Ausnahmen mit break handhaben .....	122
8.3.2 Das Schleifenende mit continue überspringen.....	124
8.4 Verschachtelte Schleifen.....	125
8.5 Die Platzhalteroption pass .....	127
8.6 Übung: Mit verschiedenen Schleifen arbeiten.....	129
<b>9. Funktionen in Python</b>	<b>134</b>
9.1 Funktionen selbst definieren.....	134
9.2 Argumente für Funktionen verwenden.....	136
9.2.1 Funktionen mit mehreren Parametern.....	139
9.3 Einen Rückgabewert verwenden .....	143
9.4 Rekursion.....	146
9.5 Funktionen in einer eigenen Datei abspeichern .....	148
9.6 Übung: Funktionen selbst gestalten .....	151
<b>10. Mit Modulen aus der Standardbibliothek arbeiten</b>	<b>156</b>
10.1 Was ist die Standardbibliothek und welche Module enthält sie? .....	156
10.2 Die Verwendung der Standardbibliothek .....	157
10.2.1 Datum- und Zeitanzeigen mit time, datetime und calendar.....	158
10.2.2 math: ein häufig verwendetes Modul .....	161
10.2.3 Den Zufall mit random programmieren.....	164
10.3 Übung: Mit der Standardbibliothek arbeiten .....	169
<b>11. Objektorientierte Programmierung</b>	<b>174</b>
11.1 Was ist objektorientierte Programmierung?.....	174
11.2 Klassen: die Grundlage der objektorientierten Programmierung .....	176
11.2.1 Docstrings .....	179
11.3 Objekte: Instanzen der Klassen .....	182
11.3.1 Objekt- und Klassenattribute .....	185
11.4 Methoden: Funktionen für Objekte .....	188
11.4.1 Statische Methoden.....	189
11.4.2 Instanzmethoden.....	190

## Inhaltsverzeichnis

11.4.3 Die dir-Funktion .....	192
11.4.4 Magische Methoden .....	193
<b>11.5 Datenkapselung .....</b>	<b>196</b>
11.5.1 Getter und Setter .....	198
11.6 Vererbung: Ein wichtiges Prinzip der OOP .....	203
11.7 Übung: Mit Objekten arbeiten .....	208
<b>12. Fehler und Ausnahmen behandeln</b>	<b>214</b>
12.1 Was sind Fehler und Ausnahmen in Python? .....	214
12.2 try ... except: Ausnahmen behandeln .....	218
12.3 finally: der Abschluss der Ausnahmebehandlung .....	224
12.4 Selbst definierte Ausnahmen festlegen .....	225
12.5 Den PyCharm-Debugger verwenden .....	228
12.6 Übung: Ausnahmen im Programm behandeln .....	233
<b>13. Textdateien für die Datenspeicherung verwenden</b>	<b>237</b>
13.1 Daten aus einer Textdatei auslesen .....	237
13.2 Daten in eine Textdatei schreiben .....	243
13.3 Weitere Lese- und Schreibeoptionen .....	246
13.4 Übung: Mit Dateien für die Datenspeicherung arbeiten .....	248
<b>14. Mit Datenbanken arbeiten</b>	<b>252</b>
14.1 Was ist eine Datenbank? .....	252
14.2 SQLite-Datenbanken erstellen und bearbeiten .....	254
14.2.1 SQLite-Tabellen erstellen .....	256
14.2.2 Zeilen aus einer Tabelle auslesen .....	259
14.2.3 SQLite-Tabellen aktualisieren .....	264
14.2.4 Tabelleninhalte löschen .....	265
14.3 NoSQL-Datenbanken .....	267
Schlüssel-Werte-Datenbanken .....	268
Dokumentorientierte Datenbanken .....	268
Spaltenorientierte Datenbanken .....	268
Graphdatenbanken .....	269
14.4 XML-Datenbanken .....	270
14.5 Übung: Eine Datenbank mit SQLite anlegen und bearbeiten .....	272
<b>15. Grafische Benutzeroberflächen mit PyQt erstellen</b>	<b>277</b>
15.1 PyQt installieren .....	278
15.2 Erste einfache Fenster erstellen .....	280
15.3 Den Qt Designer verwenden .....	284
15.3.1 Layouts für ein Fenster angeben .....	286
15.3.2 Eigenschaften der Fenster und Widgets anpassen .....	288
15.4 In Qt Designer erstellte Fenster in einem PyQt-Programm verwenden .....	291
15.4.1 Signale und Slots .....	293
15.4.2 Eine GUI aus einer .exe-Datei starten .....	297
15.5 Widgetoptionen: Klassen, Methoden und Signale .....	298
15.5.1 Buttons .....	299
15.5.2 Item Widgets .....	302
15.5.3 Container .....	303
15.5.4 Input Widgets .....	305
15.5.5 Display Widgets .....	307
15.6 Anwendungsbeispiel: Bibliotheksprogramm .....	308

**Inhaltsverzeichnis**

15.6.1	Das Aussehen des ersten Fensters festlegen.....	309
15.6.2	Die Fensterwidgets mit Signalen und Slots ausstatten .....	310
15.6.3	Das Aussehen des zweiten Fensters vorgeben.....	313
15.6.4	Die Funktionalität des Gesamtbestand-Fensters festlegen.....	314
15.6.5	Das Bibliotheksprogramm verwenden.....	314
15.7	Übung: Programme mit Fenstern selbst gestalten.....	319
<b>16.</b>	<b>E-Mails versenden und verwalten</b>	<b>326</b>
16.1	Die Funktionsweise von E-Mails .....	326
16.2	Eine E-Mailnachricht versenden.....	327
16.2.1	Erweiterte Funktionen mit dem Modul email .....	329
16.3	Empfangene E-Mailnachrichten mit imaplib bearbeiten .....	332
16.4	Übung: E-Mails mit Python bearbeiten.....	336
<b>17.</b>	<b>Webseiten mit Django erstellen: Ein Einstieg</b>	<b>340</b>
17.1	Django installieren und ein neues Projekt starten .....	341
17.2	Eine erste Webseite erstellen .....	346
17.2.1	Templates anlegen und verwenden.....	349
17.2.2	Templates: Variablen, Filter und Tags .....	352
17.2.3	Templates-Erweiterungen .....	354
17.3	App vs. Projekt .....	356
17.4	Das Layout einer Webseite mit Bootstrap anpassen.....	359
17.5	Mit Datenbanken arbeiten .....	363
17.5.1	Die admin-Seite .....	366
17.6	Eine Webseite online stellen .....	369
17.7	Übung: Webseiten mit Django erstellen .....	371
<b>18.</b>	<b>Datenanalyse mit Python</b>	<b>376</b>
18.1	Anaconda installieren und JupyterLab starten.....	377
18.2	pandas .....	380
18.2.1	Ein pandas-Dataframe erstellen .....	381
18.2.2	Daten ein- und auslesen .....	385
18.2.3	Ein Dataframe bearbeiten .....	386
18.3	NumPy.....	389
18.3.1	Ein NumPy-Array erstellen .....	389
18.3.2	Arrays bearbeiten .....	390
18.4	Datenvisualisierung mit Seaborn .....	395
18.4.1	Daten für Visualisierungen einlesen .....	396
18.4.2	Verschiedene Plottingfunktionen .....	399
18.5	Übung: Datenanalyse mit Python.....	403
<b>19.</b>	<b>Glossar</b>	<b>409</b>

## **Über die Autorin**

Sarah Schmitt studierte Mathematik und Philosophie in Heidelberg, Neu-Delhi und Berlin. Seit 2014 ist sie freiberuflich in den Bereichen Datenanalyse, Statistik und Algorithmen tätig. Darüber hinaus arbeitet sie als Übersetzerin, Lektorin und Autorin in mehreren Sprachen. Sie reist gerne und lebt in Berlin.

## **Widmung**

Ich widme dieses Buch allen Frauen, die programmieren lernen wollen oder es bereits können. Weitermachen!

## **Danksagung**

Ich danke meinen Freundinnen Doina Proorocu, Mahya & Mahsa Ilaghi und Jessica de Jesus de Pinho Pinhal für die wertvollen fachlichen Ratschläge, die stets zur richtigen Zeit kamen, sowie Albrecht Werner für das ebenso wertvolle Lektorat. Den eben genannten samt Yasmina Zian, Rahim Abdullayev, Hannelore Besser und Katie Pope danke ich für die freundschaftliche Unterstützung während der vielen Monate, die ich an diesem Buch schrieb. Ohne euch hätte ich es nicht geschafft!

# Kapitel 1

## Einleitung

1

Es macht Spaß, eine Programmiersprache zu erlernen. Die Fähigkeit, programmieren zu können, kann Ihnen eine ganz neue, faszinierende Welt eröffnen. Ideen, die Sie eventuell schon länger mit sich getragen haben, werden Sie effizient in die Tat umsetzen können, um damit nützliche Ergebnisse zu erzielen.

Ganz gleich, ob Sie für Ihre berufliche Weiterentwicklung programmieren lernen wollen oder Programme schreiben möchten, um damit Aufgaben des Alltags und eigene Projekte mithilfe neuer Methoden präzise und zeitsparend zu meistern – Programmierkenntnisse können Ihnen dabei helfen, Ihr analytisches Denkvermögen zu erweitern und gänzlich neue Wege aufzuzeigen, die Sie zuvor womöglich nicht gesehen haben.

Indem Sie lernen, die immanente Logik unterschiedlicher Problemstellungen zu erfassen und diese anschließend in strukturierter Form durch eine formalisierte Sprache auszudrücken, können Sie konkrete Lösungen für spezifische Aufgaben erzeugen. Womöglich mag Ihnen diese neue Denk- und Herangehensweise anfangs befreudlich oder sogar unnatürlich vorkommen. Sie werden jedoch schnell feststellen, dass sie zusätzlich zur intellektuellen Herausforderung ebenfalls ganz praktische Vorteile bringt und Ihnen dabei einen großen Freiraum an Kreativität lässt.

Da die Digitalisierung mittlerweile Einzug in die meisten unserer Lebens- und Arbeitsbereiche gehalten hat, ist programmiertechnisches Know-how in der heutigen Zeit sehr förderlich. Die Nachfrage nach Fachkräften mit dem entsprechenden Wissen wächst stetig, und zwar nicht nur im IT-Bereich: In den unterschiedlichsten Berufsfeldern sind solide Programmierkenntnisse mittlerweile eine gern gesehene Zusatzqualifikation und mitunter sogar eine wesentliche Voraussetzung für eine Anstellung. Die Entscheidung, eine Programmiersprache zu erlernen, ist oftmals bereits der erste Schritt in eine zeitgemäßere Arbeitswelt.

In diesem Buch werden wir die Programmiersprache *Python* ausführlich behandeln. Python ist einerseits leicht zu erlernen, lässt sich jedoch gleichzeitig sehr vielseitig und fachspezifisch einsetzen. Im Vergleich zu älteren Programmiersprachen wie *C* handelt es sich bei Python um eine neuere Sprache, in der moderne Programmiermethoden implementiert sind. Aus diesem Grund bietet Python zusätzlich eine gute Voraussetzung, um weitere Programmiersprachen schneller zu erlernen.

Die Sprache erfreut sich insbesondere seit einigen Jahren immer größer werdender Beliebtheit und wird von vielen großen Unternehmen und Institutionen innerhalb

## 1 Einleitung

von Wirtschaft, Forschung und Wissenschaft eingesetzt. So wurde beispielsweise die Google-Plattform mithilfe von Python entwickelt, und auch die Weltraumbehörde NASA setzt Python für ihre Berechnungen ein. Mit Python lassen sich Webseiten erstellen, komplexe Berechnungen effizient umsetzen, Datenanalysen vornehmen, Spiele kreieren, graphische Benutzeroberflächen erstellen, E-Mails versenden und vieles mehr.

Selbst, wenn Sie keinen technischen Hintergrund haben, ist es also eine gute Idee, mit Python in die Welt des Programmierens einzusteigen. In diesem Kapitel werden wir Ihnen zunächst einen kurzen Einblick darüber geben, was Python ist und welche Vorteile diese Sprache bietet, bevor wir in den darauffolgenden Kapiteln ihre Funktionsweise genauer kennen lernen.

### 1.1 Über dieses Buch

Dieses Buch will Anfänger und Menschen, die bereits Erfahrungen mit anderen Programmiersprachen gesammelt haben, gleichermaßen ansprechen. Es stellt eine praktische Einführung in die Programmierung mit Python dar und will fundiertes Wissen vermitteln, ohne sich dabei in allgemeinen, allzu theoretischen Konzepten des Programmierens zu verlieren. Auch Anfänger ohne jegliche mathematischen oder programmiertechnischen Vorkenntnisse sollten keinerlei Schwierigkeiten haben, den Inhalten dieses Buches zu folgen.

Als Einstieg in die Programmierung mit Python empfiehlt sich in jedem Fall eine gründliche Bearbeitung der Kapitel 1 bis 12. Für einen umfassenderen Einblick in Python können Sie danach sukzessive mit den nachfolgenden Kapiteln fortfahren, oder – je nach Schwerpunkt und Interessengebiet – einzelne für Sie relevante Themenbereiche aus den Kapiteln 13 bis 18 auswählen. Die Themengebiete werden dabei ausgehend von ihren grundlegenden Bausteinen bis in eine differenziertere Tiefe vorgestellt und ausführlich behandelt. Neu eingeführte Begriffe werden direkt an Ort und Stelle erklärt.

Ab Kapitel 4 sind die Kapitel zusätzlich mit vielen Beispielen und Übungsaufgaben versehen, anhand derer Sie das Gelernte wiederholen und verfestigen können. Alle Programmbeispiele aus dem Buch finden Sie zudem als Download unter <https://bmu-verlag.de/books/python-kompendium/>. Wir ermutigen Sie jedoch stets dazu, auf eigene Faust Beispiele zu verfassen und diese zu testen, anstatt lediglich die Beispiele in diesem Buch zu kopieren. Dadurch erlangen Sie sogleich mehr Sicherheit und eine spielerische Handlungsfreiheit im Umgang mit Python.

Menschen, die bereits Vorkenntnisse in anderen Programmiersprachen gesammelt haben, werden eventuell an einigen Stellen Inhalte wiedererkennen oder gewisse Ähnlichkeiten zu bereits erworbenem Wissen feststellen. Sie können insbesondere

die Einleitungen zu den einzelnen Kapiteln überfliegen. Wir empfehlen diesen Lesenden dennoch, die Themen aufmerksam durchzuarbeiten, da diese in der Regel zügig über das Grundwissen hinaus vertieft werden und somit auch für Menschen mit Vorkenntnissen eine wertvolle Wissenserweiterung bieten.

Wir haben versucht, geschlechtsspezifische Formulierungen weitestgehend zu umgehen und stattdessen neutrale Geschlechtsbezeichnungen zu verwenden. An einigen wenigen Stellen (etwa: „Anfänger“ oder „Nutzer“), wo dies die Einfachheit des Leseflusses unnötig verkomplizieren würde, verwenden wir jedoch ausschließlich die männliche Form. Frauen und Menschen anderer oder keiner geschlechterspezifischen Orientierung sollten dies keineswegs als Nichtbeachtung ihrer Präsenz verstehen und können sich beim Lesen in jedem Fall gleichermaßen angesprochen fühlen.

Es ist sicherlich sinnvoll, bei der Arbeit mit Python im Hinterkopf zu behalten, dass es im Internet zahlreiche Community-Plattformen und Foren für Python gibt, in denen man im Zweifelsfall Fragen stellen, über verschiedene Themen nachlesen oder eigene Beiträge verfassen kann. Spätestens dann, wenn man das erste eigene, größere Programm schreiben möchte, kommt man meistens nicht umhin, im Internet nach weiteren Informationen oder Ratschlägen zu suchen. Sie sollten es sich daher so früh wie möglich zur Gewohnheit machen, bei Bedarf selbstständige Recherchen im Internet anzustellen.

Einige der größten Wissens-Plattformen für Python sind:

- [www.python-forum.de/](http://www.python-forum.de/) (deutschsprachig)
- [www.stackoverflow.com](http://www.stackoverflow.com) (die Internetplattform für Softwareentwicklung)
- [www.python.org](http://www.python.org) (die offizielle Python-Webseite)
- [www.pypi.org](http://www.pypi.org) (*Python Package Index*, Sammlung verschiedener Pakete)
- <https://pyslackers.com/> (eine Webseite der Python-Community)

Online-Material sollte man dabei immer mit einer Prise Skepsis begegnen. Denn oftmals beeinträchtigt die schiere Menge an Informationen im Internet aufgrund ihrer leichten Zugänglichkeit und Erzeugbarkeit gleichzeitig die Qualität der Beiträge, sodass das Heranziehen von Fachliteratur in Form von Büchern, insbesondere bei fortgeschritteneren Themen, dennoch äußerst ratsam und vorteilhaft ist.

An dieser Stelle sei auch erwähnt, dass die englischsprachige (Online-)Literatur, wie in der gesamten IT-Welt, am umfangreichsten und detailliertesten ist, und dass es sich lohnt, wenn nicht sogar unumgänglich ist, sich von Anfang an parallel zur deutschen auch die englische Terminologie anzueignen. In diesem Buch werden Sie die wichtigsten Begriffe daher stets ebenfalls in Klammern auf Englisch angegeben vorfinden.

Sollten Sie einmal in der Bearbeitung dieses Buches nicht weiterkommen – vielleicht, weil Ihnen bestimmte Inhalte unverständlich waren oder Sie den Überblick verloren

## 1 Einleitung

haben – so nehmen Sie es mit Gelassenheit. Legen Sie eine Pause ein, machen Sie einen Spaziergang, schauen Sie sich einen Monty-Python-Sketch an, beschäftigen Sie sich mit etwas Anderem. Zu einem späteren Zeitpunkt können Sie sich wieder entspannt und mit klarem Kopf an die Lektüre setzen.

Wir wünschen viel Spaß und Freude beim Lernen!

### 1.2 Python: eine einfach zu erlernende Programmiersprache

Python ist eine flexible, einfache und elegante Programmiersprache, die sowohl für Anfänger als auch Fortgeschrittene bestens geeignet ist. Die klare, übersichtliche Syntaxstruktur macht es insbesondere Anfängern vergleichsweise leicht, mit Python den Einstieg in das Programmieren zu meistern.

In Bezug auf Programmiersprachen bezeichnet die *Syntax* – ähnlich wie bei natürlichen Sprachen – die den Satzbau regelnden formalen Vorschriften. Diese geben an, welche Kombinationen aus Symbolen (Zeichenketten, Satzzeichen, Sätzen, etc.) eine korrekte Struktur innerhalb der jeweiligen Sprache sind. Syntaxfehler sind dann beispielsweise Symbole, die an der falschen Stelle stehen oder Rechtschreibfehler enthalten. Ebenfalls als Syntaxfehler einzuordnen sind Wörter oder Befehle, die die Programmiersprache nicht kennt oder auch Ausdrücke, die in einem falschen Kontext verwendet werden. Enthält der Programmcode Syntaxfehler, lässt sich das Programm nicht zum Laufen bringen. Syntaxfehler passieren häufig und können selbst erfahrenen Programmierern und Programmiererinnen unterlaufen. Die Suche nach der fehlerhaften Stelle ist dabei oftmals mühsam und zeitaufwändig, da der Fehler zumeist nicht in den eigentlichen Befehlen liegt, sondern beispielsweise bloß durch eine fehlende Klammer, einen fehlenden Doppelpunkt oder einen Rechtschreibfehler verursacht wurde. Gerade Neulinge tun sich oftmals schwer damit, die Fehler in der Syntax zu finden – und können ihr Programm dann nicht starten.

Python überzeugt hier durch gute Lesbarkeit und ein ordentliches visuelles Layout. So kommt diese Programmiersprache beinahe gänzlich ohne Klammern aus: Zusammengehörende Blöcke werden anstatt durch Satzzeichen (wie z. B. Klammern) oder Befehlwörter bloß durch Einrückungen im Code ausgewiesen und sogleich sinnvoll gruppiert. Während in vielen anderen Programmiersprachen zudem verlangt wird, dass auf jeden Befehl ein Semikolon folgt, verzichtet Python auf den Einsatz von Semikolons nach Befehlen. All dies reduziert die Fehleranfälligkeit beim Schreiben von Programmcode erheblich und trägt zu Pythons Benutzerfreundlichkeit bei.

Wenn Sie sich bereits mit Programmiersprachen beschäftigt haben, so wissen Sie, was eine *Variable* ist. Ähnlich wie Variablen in der Mathematik enthalten Variablen beim Programmieren bestimmte Informationen, auf die später mittels eines Variablenamens zugegriffen werden kann, um damit weitere Operationen und Berech-

### 1.3 Die Entwicklungsgeschichte

1

nungen auszuführen. Variablen können dabei nicht nur aus unterschiedlichen Zahlentypen (ganzen Zahlen, Fließkommazahlen, etc.), sondern ebenso aus Buchstaben (sog. *Strings* bzw. *Zeichenketten*) oder auch anderen, komplexeren Datenstrukturen bestehen. In einigen Programmiersprachen müssen Variablen anfangs deklariert und mit einem festen, unveränderlichen Datentyp versehen werden. Dies kann mitunter zu Fehlern führen, wenn man die Deklaration am Anfang vergisst oder versucht, mit verschiedenen Variablentypen zu arbeiten, die per Definition inkompatibel sind.

Python ist hier anders. Anwendende haben viele Freiheiten beim Gebrauch unterschiedlicher Variablentypen, da diese anfangs nicht deklariert werden müssen – der jeweilige Variablentyp wird dynamisch vergeben und kann auch später noch innerhalb des Programms verändert werden. Diese spezielle Verwendung der Variablen in Python nennt man *dynamische Typisierung*.

All diese Eigenschaften tragen dazu bei, dass Python einfach zu erlernen ist und auch auf fortgeschrittenem Niveau Spaß in der Anwendung macht.

## 1.3 Die Entwicklungsgeschichte

Python wurde Anfang der 90er Jahre vom niederländischen Programmierer Guido van Rossum am *Centrum Wiskunde & Informatica* in Amsterdam entwickelt. Über seine Beweggründe, eine neue Programmiersprache zu entwickeln, schrieb von Rossum im Jahr 1996 Folgendes:

*„Vor über sechs Jahren, im Dezember 1989, suchte ich nach einem ‚Hobby‘-Programmierprojekt, das mich über die Weihnachtswoche beschäftigen würde. Mein Büro [...] in Amsterdam würde geschlossen bleiben, aber ich hatte zu Hause einen PC und sonst nicht viel zu tun. Ich beschloss, einen Interpreter für die neue Skriptsprache zu schreiben, über die ich in letzter Zeit nachdachte: Ein Nachfolger von ABC, der auch Unix- und C-Hacker ansprechen würde. Ich wählte Python als Arbeitstitel für das Projekt, da ich in einer leicht respektlosen Stimmung (und ein großer Fan des Monty Python’s Flying Circus) war.“*

– frei übersetzt aus der Einleitung von *Programming Python*  
[Autor: Mark Lutz; O'Reilly Verlag]

Im Jahr 1994 erschien mit Python 1.0 die erste Vollversion. In den darauffolgenden Jahren kamen zahlreiche weitere Versionen heraus, die den Funktionsumfang von Python erheblich ausbauten, bis im Jahr 2000 sodann Python 2.0 erschien. Diese Version enthielt erstmals die sogenannte *Garbage Collection* (engl. für *Müllabfuhr*), bei der der Speicherplatz, den ein Programm besetzt, automatisch belegt und wieder freigegeben wird, sodass die Programmiererin oder der Programmierer die Speicherplatzverwaltung nicht manuell vornehmen muss – was wiederum Zeit spart und den

## 1 Einleitung

Programmievorgang erleichtert. Bemerkenswert an Python 2.0 war auch der Entwicklungsprozess an sich, in dem Transparenz und die Mitwirkung der Community eine große Rolle spielten.

Nach einer langen Testphase erschien im Dezember 2008 schließlich Python 3.0, auch „Python 3000“ oder „Py3K“ genannt. Diese dritte Version war eine komplette Überarbeitung der Sprache, in der viele Unstimmigkeiten und Designfehler behoben wurden. Dabei musste bei der Entwicklung jedoch auf die Option der Abwärtskompatibilität verzichtet werden. Das bedeutet, dass Programme, die mit Python 2 geschrieben wurden, nicht mit einem für Python 3 konzipierten Interpreter ausgelesen werden können. Python 2.0 wurde bis zur Version 2.7, die im Jahr 2010 herauskam, zwei Jahre lang parallel zur Version 3.0 weiterentwickelt. In dieser Zeit gab es daher stets zwei aktuelle Python-Versionen, zwischen denen sich die Anwender entscheiden mussten. Obwohl ältere Programme häufig noch in Python 2 geschrieben sind, wird für neuere Programme in der Regel Python 3 verwendet. Darüber hinaus finden sich noch viele ältere Lehrbücher zu Python 2. Der Support und Updates für Python 2 wurden ab 1. Januar 2020 jedoch eingestellt.

In diesem Buch werden wir uns dem aktuellen Stand entsprechend ausschließlich mit Python 3 befassen, sodass kein Wissen über die genauen Unterschiede zwischen Python 2 und Python 3 vollenötigen sein wird. Sie sollten sich lediglich der Möglichkeit bewusst sein, dass Code aus anderen Quellen mitunter nicht kompatibel zur aktuellen Version sein könnte. Beim Verwenden von fremdem Code ist es daher stets ratsam, vorab zu überprüfen, in welcher Version von Python dieser geschrieben wurde.

### 1.4 Die Ziele bei der Entwicklung von Python

Wie bereits erwähnt, wurde Python ursprünglich als Nachfolger für die Programmier-Lehrsprache ABC entwickelt. Im Vergleich zu ABC sollte Python überschaubarer und leicht zu erweitern sein. Gute Lesbarkeit und Einfachheit standen dabei an erster Stelle. So beschränkt die reduzierte Syntax beispielsweise den Programmcode auf dessen wesentliche Elemente, sodass diese im Vordergrund bleiben.

In der 2004 erschienenen Reihe *The Zen of Python* wird die minimalistische Philosophie dieser Programmiersprache in knappen, humoristischen Aphorismen dargelegt. Das Werk enthält Leitsätze wie:

„Schön ist besser als hässlich.  
Explizit ist besser als implizit.  
Einfach ist besser als komplex.  
Komplex ist besser als kompliziert.  
Lesbarkeit zählt.“

– Tim Peters, *The Zen of Python* (frei übersetzt)

## 1.4 Die Ziele bei der Entwicklung von Python

1

Darüber hinaus sollte Python intuitiv und genauso einfach zu verstehen sein wie die englische Sprache; die Anwendung sollte Spaß machen. Dies verdeutlicht nicht zuletzt die originelle Namensgebung: Python ist nicht, wie man zunächst vermuten könnte, nach der gleichnamigen Würgeschlangenart benannt, sondern nach der britischen Fernsehsendung *Monty Python's Flying Circus*. Obwohl inzwischen tatsächlich eine Schlange als Symbol der Programmiersprache dient, ist die ursprüngliche Bedeutung des Namens noch an vielen Stellen erkennbar. So sind die Webseite der Python-Dokumentation und viele andere Lehrbücher über Python mit Zitaten aus Monty-Python-Sketchen versehen, und Anspielungen auf die Fernsehsendung im Code werden gerne eingesetzt.

Der eigentliche Programmcode ist in Python sehr knapp gehalten. Die gleichen Programme mit denselben Funktionen, in anderen Sprachen verfasst, benötigen dort in der Regel mehr Zeilen als in Python. Anfänger sind hier also im Vorteil: übersichtlichere, kürzere Codebeispiele sind leichter zu lesen und zu verstehen. Auch die Zeit, die zum Schreiben eines neuen Programms benötigt wird, bleibt hierdurch vergleichsweise gering. Aus diesem Grund wird Python gerne für Projekte mit hohem Zeitdruck eingesetzt.

Darüber hinaus verfügt Python über viele weitere Eigenschaften, die zur Übersichtlichkeit und leichten Verständlichkeit des Codes beitragen. Im Vergleich zu anderen Programmiersprachen kommt Python etwa nur mit sehr wenigen grundlegenden Schlüsselwörtern aus. Die Standardbibliothek, eine von Pythons größten Stärken, wurde bewusst überschaubar gehalten und lässt sich anschließend leicht erweitern. Das Potential der Sprache wird somit keineswegs durch diese anfängliche Einschränkung beeinträchtigt. Für Neulinge hat dies jedoch den Vorteil, dass sie anfangs vergleichsweise wenige Begriffe auswendig lernen müssen und sich leichter in die neue Sprache einarbeiten können.

Wichtig zu erwähnen ist außerdem, dass es sich bei Python um ein Open-Source-Projekt handelt. Hiermit wird ein Modell innerhalb der Softwareentwicklung bezeichnet, das um 2000 mit der zunehmenden Verbreitung des Internets entstand. *Open Source* bedeutet, dass die Öffentlichkeit den Quellcode (einer Sprache oder eines Programms) – im Gegensatz zu urheberrechtlich oder lizenzierten geschützten Programmen – kostenlos einsehen, modifizieren oder weiterverbreiten kann. Open-Source-Projekte fördern infolgedessen die freiwillige Unterstützung von Anwendern, die eventuelle Verbesserungen oder Problemlösungen innerhalb der Community teilen können und somit zur Weiterentwicklung des Projekts beitragen. Pythons Einfachheit und Vielseitigkeit ließen so neben einer hohen Nutzerschaft auch sehr schnell eine große, aktive Community entstehen.

Die oberste Entscheidungsmacht über die Entwicklung von Python lag seit dessen Entstehung jedoch weiterhin bei Guido van Rossum als sog. *Benevolent Dictator for*

## 1 Einleitung

Life („wohlwollender Diktator auf Lebenszeit“). Obwohl er sich im Juli 2018 von dieser Position verabschiedete, beteiligt er sich nach wie vor an der Weiterentwicklung von Python.

### 1.5 Interpretiert vs. kompiliert

Bevor Sie die für das Programmieren mit Python nötige Software installieren und kennen lernen, ist es eine gute Idee, über die unterschiedlichen Implementierungsmöglichkeiten von Programmen Bescheid zu wissen. Dazu wollen wir ein wenig auf die internen Abläufe eingehen, die beim Ausführen eines Programms stattfinden.

Computerprogramme werden in gewöhnlichem Text geschrieben. Anders als natürliche Sprachen bestehen sie jedoch aus fest vorgegebenen Symbolen und Schlüsselbegriffen, die einer ebenfalls fest vorgegebenen Struktur folgen – der Syntax (Kapitel 1.2.). Diese Begriffe und Symbole sind sehr stark an die englische Sprache und an mathematische Symbole angelehnt, sodass selbst Anfänger mit grundlegenden Englisch- und Mathematikkenntnissen beim Lesen eines Computerprogramms oftmals bereits erahnen können, welche Aufgaben es ausführen soll.

Auf einer weniger abstrakten Ebene haben Programmabläufe jedoch eine ganz andere Gestalt. Zu Anfang bestehen alle Informationen nämlich aus reinem *Binärkode*. Das bedeutet, dass jede Information lediglich zwei verschiedene Zustände annehmen kann. Auf der Hardwareebene handelt es sich dabei in der Regel ursprünglich um elektrische Impulse mit den beiden möglichen Zuständen „niedrig“ und „hoch“, die im Rechner stellvertretend mit den Zahlen 0 und 1 bezeichnet werden. Jede Null und jede Eins ist ein sogenanntes *Bit* (von engl. *binary digit*). Kombinationen dieser kleinsten Informationseinheiten werden als Wörter bezeichnet.

In modernen Computern kommen 64-Bit-Wörter zum Einsatz, wohingegen ältere Rechner mit 32-, 16- oder sogar 8-Bit-Wörtern arbeiteten. Jedes Wort, das im Prozessor ankommt, bewirkt eine bestimmte Ausgabe. Die Abfolge aller Wörter eines Programms führt somit dazu, dass der Prozessor das gewünschte Ergebnis ausgibt. Diesen aus Wörtern von Binärzahlen bestehenden Code nennt man *Maschinensprache* bzw. *Maschinencode*. Ein Beispiel für Maschinencode könnte etwa die folgende Form haben:

1	0000	11111010
2	0001	00110011
3	0002	11000000
4	0003	10001110
5	0004	11010000
6	0005	10111100
7	0006	00000000
8	0007	00000001

## 1.5 Interpretiert vs. kompiliert

1

Höhere Programmiersprachen wie Python beruhen ebenfalls auf den Prinzipien der Maschinensprache, sind jedoch weitaus abstrakter und komplexer entwickelt. Damit ein in einer Hochsprache geschriebenes Computerprogramm ausgeführt werden kann, muss es zunächst in Maschinensprache übersetzt werden. Hierfür gibt es klassischerweise zwei Möglichkeiten:

Zum einen kann der Programmcode mithilfe eines *Compilers* in Maschinencode umgewandelt werden. Dies hat den Vorteil, dass bereits kompilierte Programme bei Folgeausführungen nicht erneut übersetzt werden müssen, was offensichtlich Zeit spart. Zudem überprüft der Kompilierer das Programm vor dessen Erstellung auf Fehler, sodass diese bereits davor behoben werden können. Der Nachteil an dieser Methode ist jedoch, dass die jeweilige Kompilation nicht plattformunabhängig ist und dass das Programm somit nicht ohne Weiteres auf verschiedenen Betriebssystemen laufen kann. Programmiersprachen wie *C* oder *C++* laufen beispielsweise über einen Compiler.

Eine andere Möglichkeit besteht darin, einen *Interpreter* zu verwenden. Ein Interpreter ist ein Programm, das den Code während seiner Ausführung einliest und ihn Zeile für Zeile direkt in Maschinensprache übersetzt. Ein interpretiertes Programm kann auf jedem Rechner laufen, der mit einem entsprechenden Interpreter ausgestattet ist, ohne dass das Betriebssystem des Rechners dabei eine Rolle spielt. Der Entwicklungsprozess gestaltet sich effizienter, da das Programm bei jedem erneuten Testen nicht aufs Neue kompiliert werden muss, sondern direkt mithilfe des Interpreters ausgeführt werden kann. Insbesondere bei größeren Programmen und mehreren hundert Kompiliervorgängen spart dies beim Entwickeln viel Zeit.

In Wirklichkeit schließen sich diese beiden Begriffe nicht gegenseitig aus, da jede Programmiersprache theoretisch betrachtet entweder interpretiert oder kompiliert werden kann. Während die Einteilung in *kompiliert* und *interpretiert* früher eine größere Rolle spielte, wird in modernen Programmiersprachen-Implementierungen zunehmend eine Kombination aus beiden Möglichkeiten innerhalb einer Plattform bevorzugt, sodass die einstige Unterscheidung weniger relevant ist. Heutzutage gibt es nur noch sehr wenige klassische Interpreter. Die meisten, darunter auch Python, kompilieren den Programmcode zunächst in einem Zwischenschritt in Bytecode, der anschließend durch einen Interpreter auf einer virtuellen Maschine (im Falle von Python von der *PVM*, **Python Virtual Machine**) ausgeführt wird.

## Kapitel 2

# Die Vorbereitung

Bevor wir mit dem eigentlichen Programmieren anfangen können, müssen wir einige Vorbereitungen treffen. Wie Sie bereits aus dem vorigen Kapitel wissen, bestehen Computerprogramme aus reinem Text. Um diesen Text zu schreiben und ihn in einer Datei abzuspeichern, die vom Interpreter anschließend ausgelesen wird, benötigt man zunächst bloß ein entsprechendes Textverarbeitungsprogramm zum Erstellen von Textdokumenten – Microsoft Word oder LibreOffice sind Beispiele für derartige Software. Jedoch enthalten diese Programme zusätzlich zum Text noch weitaus mehr Informationen – etwa Angaben zur Schriftart, zur Schriftgröße sowie zu vielen anderen Formatierungsdetails –, die auf eine zum Programmieren ungeeignete Art und Weise gespeichert werden. Aus diesem Grund kommt beim Programmieren ein Texteditor zum Einsatz.

Auf Computern mit Windows als Betriebssystem ist bereits ein solcher Texteditor vorinstalliert: Notepad. Obwohl man mit dem Notepad-Editor theoretisch Programme schreiben könnte, ist dessen Funktionalität sehr begrenzt und nicht an die spezifischen Bedürfnisse des Programmierens angepasst. Nötig ist hier also eine Software mit umfangreicheren Möglichkeiten. Ein geeigneter Editor sollte unter anderem bestimmte Befehle automatisch erkennen und zur besseren Lesbarkeit farblich markieren können, selbstständig notwendige Einrückungen von zusammengehörenden Textblöcken vornehmen, über eine Suchen- und Ersetzen-Funktion verfügen und vieles mehr.

Um den geschriebenen Code sogleich ausführen und testen zu können, ist darüber hinaus ein Erstellungsprogramm notwendig. Zusätzlich sollte die Entwicklungsumgebung über einen Debugger verfügen, der beim Programmieren dabei hilft, eventuelle Fehler im Skript, die das Programm an der Ausführung hindern, zu lokalisieren und zu beheben.

### 2.1 Die integrierte Entwicklungsumgebung

Eine *IDE* (*Integrierte Entwicklungsumgebung*, von engl. *Integrated Development Environment*) ist dasjenige Tool, das all diese Anforderungen vereint. In einer IDE können Sie bequem Code verfassen, speichern, wiederholt testen und ausführen. Das Angebot an unterschiedlichen IDEs ist dabei schier unbegrenzt. Es gibt IDEs, die Sie für viele verschiedene Programmiersprachen verwenden können und solche, die nur jeweils eine Sprache unterstützen. Anwender werden schnell feststellen, dass der Aufbau der

## 2.2 Den Python-Interpreter installieren

2

meisten IDEs im Grunde recht ähnlich ist, während sie sich hinsichtlich ihres Designs, ihrer Funktionalität und Nutzerfreundlichkeit stark unterscheiden können.

Die Wahl der Entwicklungsumgebung sollte sich vor allem nach Ihren persönlichen Programmierzielen und Ihrem Interessenfeld richten. Mittlerweile gibt es sehr spezifische, mächtige Tools für die unterschiedlichsten Anwendungsbereiche, die mitunter bereits Pakete für spezifische Anwendungen beinhalten. Für Python stehen Ihnen über 50 solcher IDEs zur Verfügung, die meisten davon sind kostenfrei.

Anfänger haben zunächst die Möglichkeit, diejenige IDE zu verwenden, die bei der Installation des Python-Interpreters (unter Windows und macOS) automatisch mit enthalten ist: *IDLE* (engl. **I**ntegrated **D**evelopment and **L**earning **E**nvironment) ist ein einfach gehaltenes Tool, das besonders für Lernzwecke völlig ausreichend ist. Im Anschluss an die in den nächsten Abschnitten beschriebene Installation des Interpreters können Sie IDLE sofort testen und sehen, ob Sie gut damit zurechtkommen.

Im Hinblick auf fortgeschrittenere Ziele und Anforderungen, die auch Bestandteil dieses Buchs sein werden, wird IDLE jedoch womöglich nicht ausreichend sein. Einige der hierfür in Frage kommenden, empfehlenswerten IDEs sind etwa *Thonny* (für Anfänger; <https://thonny.org/>), *Visual Studio Code* (<https://code.visualstudio.com/>), *Spyder* (insbesondere für Datenanalyse; [www.spyder-ide.org](http://www.spyder-ide.org)) oder *PyCharm* ([www.jetbrains.com/pycharm](http://www.jetbrains.com/pycharm)). In den Beispielen dieses Buchs wird PyCharm zum Einsatz kommen. Hierbei handelt es sich um ein sehr beliebtes, komfortables und zugleich starkes Entwicklungstool.

Linux-Anwender verfügen in der Regel bereits über einen entsprechenden einfachen Texteditor, nicht jedoch über eine umfassende IDE. Bei Ubuntu ist dies *gEdit* und bei KDE *Kate*. Auch andere Distributionen sind meistens mit einem passenden Texteditor ausgestattet.

Im nächsten Abschnitt 2.2 werden wir erklären, wie Sie den Python-Interpreter, der den Code in Maschinensprache übersetzt, auf verschiedenen Betriebssystemen installieren. Im darauffolgenden Abschnitt 2.3. werden wir uns sodann genauer der Verwendung und Installation der PyCharm-IDE widmen und Ihnen in 2.4. zeigen, wie Sie diese anpassen können.

## 2.2 Den Python-Interpreter installieren

Wie bereits erwähnt, arbeitet Python mit einem Interpreter, der den großen Vorteil der Plattformunabhängigkeit bietet. Das bedeutet, dass Python-Code auf allen Betriebssystemen laufen kann, ohne dass dieser hierfür abgeändert werden muss. Den Python-Interpreter können Sie unter der folgenden Internetadresse herunterladen: <https://www.python.org/downloads/>.

## 2 Die Vorbereitung



**Abb. 2.1** Die Startseite der Python-Entwicklergemeinschaft mit dem Download des Interpreters

Durch Anklicken der gelben Schaltfläche öffnet sich ein Fenster zum Download des Installations-Assistenten. In der Regel wird die Webseite automatisch die für das Betriebssystem Ihres Computers passende Version auswählen. Sollte dies nicht der Fall sein, können Sie die richtige Version auf derselben Seite auswählen. Hier finden sich zudem auch frühere Python-Versionen, darunter beispielsweise auch Python 2.7.

### 2.2.1 Unter Windows installieren

Python läuft erst ab Version 3.5 auf Windows Vista oder neueren Betriebssystemen.

Nachdem der Download beendet wurde, können Sie die Installation des Interpreters durch Anklicken der Datei starten. Das folgende Fenster wird nun angezeigt:

## 2.2 Den Python-Interpreter installieren

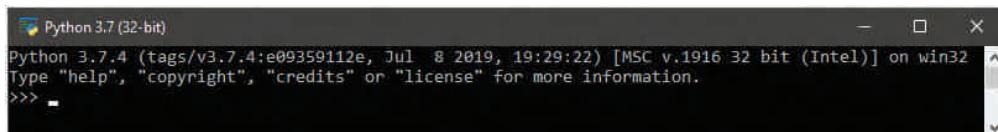


2

**Abb. 2.2** Der Installations-Assistent für den Python-Interpreter

Hier müssen Sie darauf achten, dass Sie in der unteren Checkbox **Add Python 3.7 to PATH** einen Haken setzen. Dadurch wird es möglich sein, Python von der Eingabeaufforderung (Kommandozeile) aus zu starten, anstatt jedes Mal in das Verzeichnis wechseln zu müssen, in dem der Interpreter installiert wurde. Die eigenen Programme können somit nicht nur im Interpreter-Verzeichnis, sondern auch in anderen Verzeichnissen gespeichert und von dort aus aufgerufen werden. Mit **Install Now** bestätigen Sie anschließend die Installation, woraufhin der Python-Interpreter startklar ist.

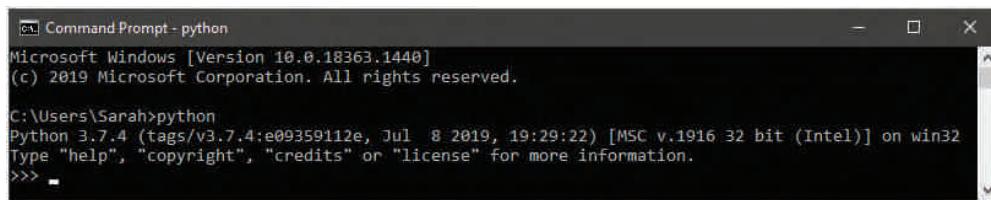
Über das Startmenü können Sie nun überprüfen, ob Python korrekt installiert wurde. Wählen Sie hierzu unter der Liste der installierten Programme den Ordner **Python 3.7** und darin **Python 3.7 (32 Bit)** aus. Der Python-Interpreter öffnet sich in der Eingabeaufforderung, die Versions- und Installationsdetails werden angezeigt. In dieser Menüauswahl befindet sich ebenfalls die zuvor erwähnte Entwicklungsumgebung IDLE.

**Abb. 2.3** Python-Installationsinfo im Python-Interpreter

Alternativ hierzu können Sie den Python-Interpreter auch direkt in der Eingabeaufforderung durch den Befehl `python` öffnen. Zur Eingabeaufforderung gelangen Sie

## 2 Die Vorbereitung

am schnellsten, indem Sie **cmd** in die Windows-Suche eingeben. Die folgende Ausgabe sollte angezeigt werden:



The screenshot shows a Windows Command Prompt window titled "Command Prompt - python". The window displays the following text:  
Microsoft Windows [Version 10.0.18363.1440]  
(c) 2019 Microsoft Corporation. All rights reserved.  
C:\Users\Sarah>python  
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul 8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> -

Abb. 2.4 Python-Installationsinfo

Die drei spitzen Klammern (>>>) bedeuten hier, dass der Python-Interpreter einsatzbereit ist und auf Befehle wartet.

In der Eingabeaufforderung – also nicht im Interpreter – können Sie übrigens jederzeit durch die Eingabe von `python --version` überprüfen, welche Python-Version auf Ihrem PC aktuell installiert ist.

### 2.2.2 Unter Linux installieren

Sehr wahrscheinlich ist auf Ihrem System bereits mindestens eine Version von Python vorhanden. Um zu überprüfen, ob es sich dabei um Python 2 oder um Python 3 handelt, können Sie in einem Terminal-Fenster (**Ctrl + Alt + T**) die Befehle `python -V` (für Python 2) bzw. `python3 -V` (für Python 3) eingeben. Die entsprechenden Installationsinformationen werden angezeigt. Um die derzeit aktuellste Python-Version, Python 3.7, zu verwenden, können Sie deren Installation durch den Befehl `sudo apt-get install python3.7` veranlassen. Mit dem Befehl `python3.7` lässt sich anschließend kontrollieren, ob die Installation richtig durchgeführt wurde.

### 2.2.3 Unter macOS installieren

Standardmäßig ist auf macOS Python 2.7 vorinstalliert. Um die aktuelle Version zu erhalten, müssen Sie die entsprechende Datei wie oben in 2.2. beschrieben herunterladen, ausführen und sodann den Schritten auf dem Bildschirm folgen. War die Installation erfolgreich, sollten auf die Eingabe von `python3` in einem Terminal-Fenster (zu finden unter **Applications → Utilities → Terminal**) hin die Installationsinformationen erscheinen.

## 2.3 PyCharm – Eigenschaften und Installation

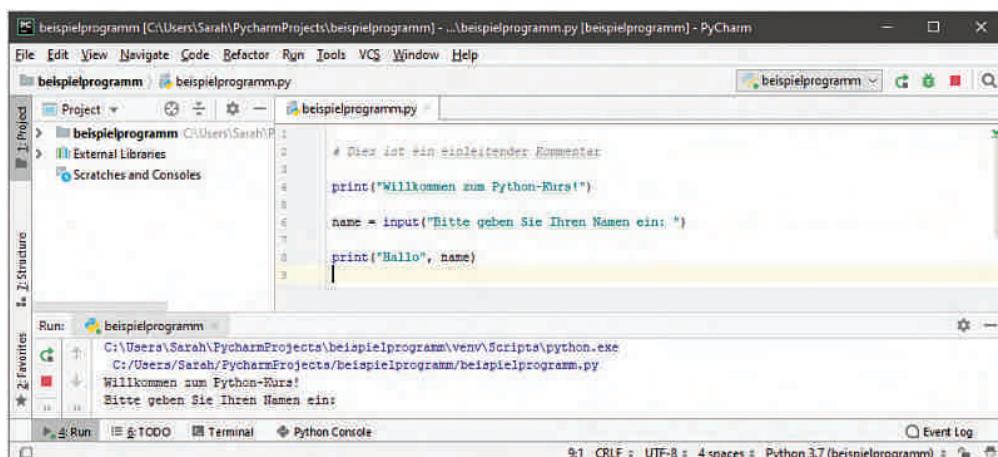
Widmen wir uns nun dem Tool, mit dem wir im Rahmen dieses Buches hauptsächlich arbeiten werden: PyCharm. Zusätzlich zum Texteditor beinhaltet PyCharm weitere

## 2.3 PyCharm – Eigenschaften und Installation

Komponenten wie eine Quellcodeverwaltung, einen Debugger sowie Test- und Analysewerkzeuge. Hier sind einige der Vorteile, die der PyCharm-Texteditor gegenüber einem gewöhnlichen Texteditor wie Notepad bietet:

- ▶ Syntaxhervorhebung: Verschiedene Bestandteile des Codes werden automatisch erkannt und farblich markiert, sodass sie leichter voneinander zu unterscheiden sind.
- ▶ Zeilenummerierung: Durch die Nummerierung der Zeilen wird der geschriebene Programmcode übersichtlicher und man kann leichter zwischen verschiedenen Stellen navigieren. Zudem lassen sich Fehlermeldungen einfacher nachvollziehen, da diese normalerweise zusammen mit der Zeilenummer angegeben werden.
- ▶ Code-Faltung: Einzelne zusammengehörende Blöcke lassen sich durch einen Klick auf das seitliche Minuszeichen ein- und wieder ausklappen. Das erhöht die Übersichtlichkeit beim Arbeiten mit dem Text.
- ▶ Suchen und Ersetzen-Funktion: Bestimmte Elemente im Code lassen sich gezielt suchen und durch einen anderen Befehl automatisch ersetzen. Dadurch lassen sich Korrekturen deutlich schneller ausführen.
- ▶ Automatische Vervollständigung von Befehlen: Sobald Sie die ersten Buchstaben eines Befehls eingetippt haben, öffnet sich ein Fenster, in dem Sie den richtigen Befehl unter verschiedenen Möglichkeiten auswählen können.
- ▶ Automatische Einrückung: Zusammengehörende Blöcke werden automatisch eingerückt. Da Einrückungen bei Python bestimmte Funktionen haben, ist diese Eigenschaft besonders praktisch.
- ▶ Kompilieren und ausführen: Die Programme können direkt vom Texteditor aus kompiliert und ausgeführt werden.

2



**Abb. 2.5** Erste Ansicht der PyCharm-IDE

## 2 Die Vorbereitung

Sie können PyCharm auf der Seite <http://www.jetbrains.com/pycharm/> für Ihr Betriebssystem herunterladen. Hier haben Sie die Auswahl zwischen zwei verschiedenen Versionen: der *Professional*-Version (zahlungspflichtig) und der quelloffenen *Community*-Version (kostenlos). Die Community-Version wird für unsere Zwecke vollkommen ausreichend sein.

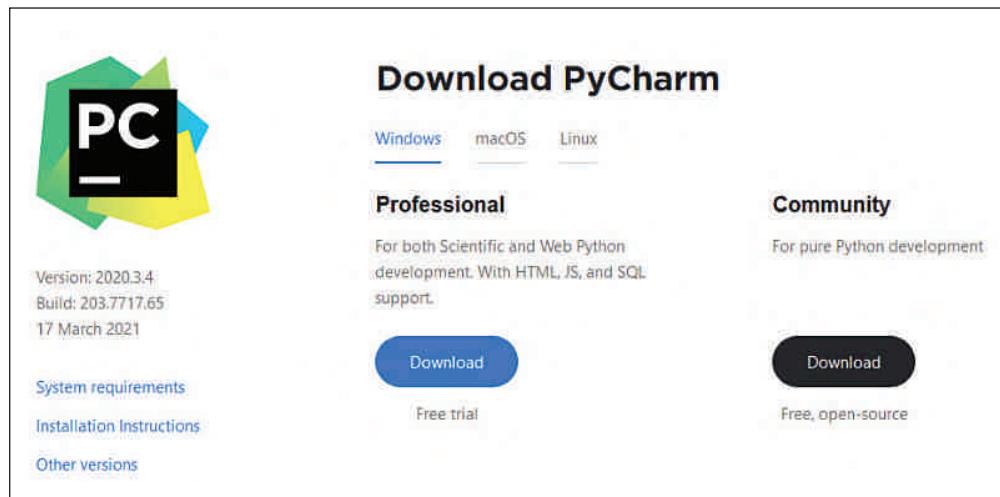


Abb. 2.6 Erhältliche PyCharm-Versionen

Die Wahl der Entwicklungsumgebung ist nicht zwingend. Sollten Sie feststellen, dass Sie mit einer anderen IDE besser zureckkommen, können Sie für die Aufgaben und Beispiele in diesem Buch gerne auch diese verwenden. Eine Liste über einige allgemeine sowie speziell auf Python zugeschnittene IDEs samt Vor- und Nachteilen gibt es unter <https://realpython.com/python-ides-code-editors-guide/>. Eine vollständige Liste finden Sie unter <https://wiki.python.org/moin/PythonEditors>.

### 2.3.1 PyCharm: Installation unter Windows

Nachdem Sie die Installationsdatei heruntergeladen haben, folgen Sie den Schritten des Installationsprogramms. Wählen Sie zunächst den Ordner aus, in dem Sie PyCharm installieren möchten. Im darauffolgenden Fenster müssen Sie sodann angeben, ob Ihr Betriebssystem auf einer 32-Bit- oder einer 64-Bit-Architektur beruht. Dies können Sie in der Systemsteuerung unter **System → Systemtyp** herausfinden. Verwenden Sie tatsächlich ein 32-Bit-System, so müssen Sie zusätzlich unten im Fenster das Feld *Download and install JRE x86 by JetBrains* durch ein Häkchen aktivieren. Achten Sie in jedem Fall zudem darauf, dass Sie unter *Create Associations* einen Haken bei *.py* setzen. Dadurch werden Python-Quelltexte (also alle *.py*-Dateien) automatisch direkt mit PyCharm geöffnet. Wählen Sie zudem die Option *Add launchers dir to the PATH* aus.

## 2.3 PyCharm – Eigenschaften und Installation

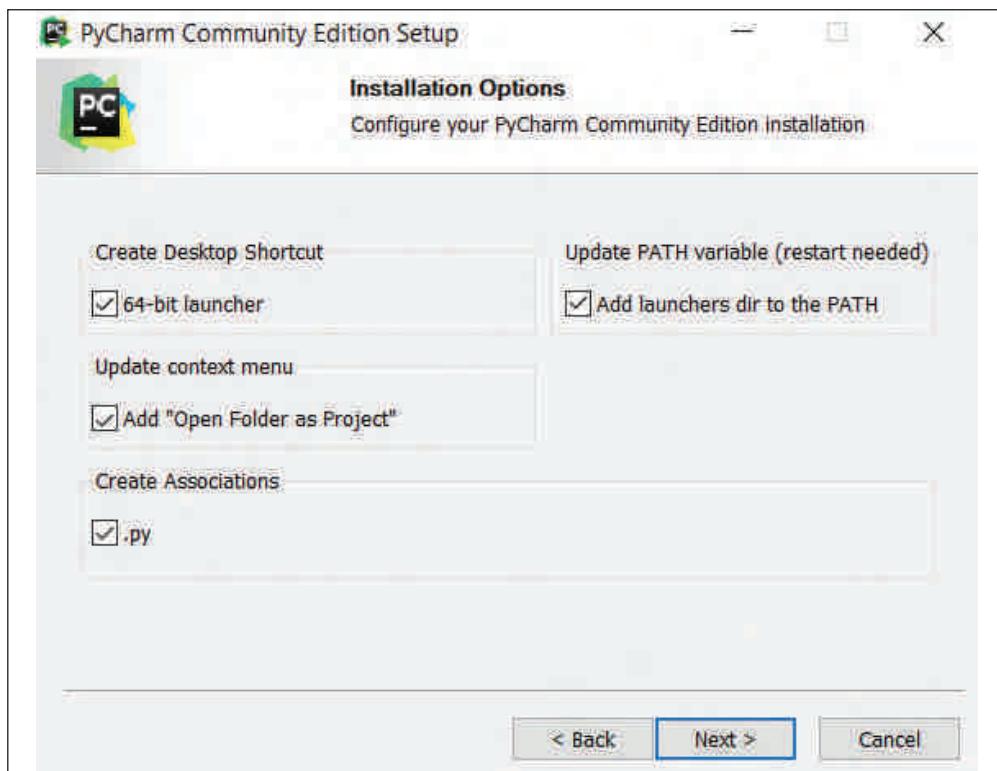


Abb. 2.7 PyCharm unter Windows installieren

### 2.3.2 PyCharm: Installation unter Linux

Ab Ubuntu 16.04 lässt sich PyCharm über ein Snap-Paket installieren. Durch den Kommandozeilenbefehl `sudo snap install pycharm-professional --classic` für die Professional-Version bzw. `sudo snap install pycharm-community --classic` für die kostenlose Community-Version können Sie die Installation veranlassen.

Unter Ubuntu 18.04 befindet sich PyCharm bereits in den offiziellen Paketquellen, was die Installation sehr einfach gestaltet. Klicken Sie im Dock auf **Ubuntu-Software** oder alternativ dazu auf **Anwendungen Anzeigen → Ubuntu-Software**. Geben Sie im Suchfeld oben rechts *pycharm* ein. Nun können Sie eine der angezeigten Versionen auswählen, beispielsweise die kostenfreie Version *PyCharm CE*. Bestätigen Sie Ihre Auswahl durch **Installieren** und starten Sie PyCharm.

## 2 Die Vorbereitung

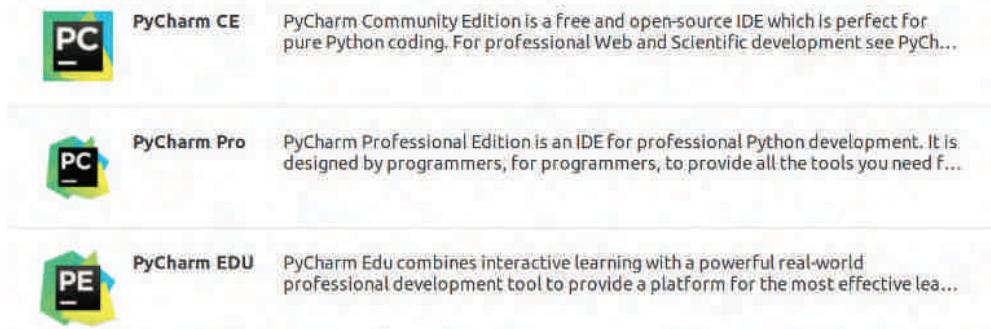


Abb. 2.8 Installation unter Ubuntu 18.04

Unter einer anderen Linux-Distribution sollten Sie den Anweisungen auf <https://www.jetbrains.com/help/pycharm/installation-guide.html?section=Linux> zur Installation als .tar.gz-Archiv folgen.

### 2.3.3 PyCharm: Installation unter macOS

Laden Sie die Installationsdatei unter der oben angegebenen Adresse herunter. Auch hier können Sie zwischen der Professional- und der Community-Version wählen. Öffnen Sie die .dmg-Datei nach dem Download. Anschließend öffnet sich ein Fenster. Nun müssen Sie das Icon *PyCharm CE* in den Ordner *Applications* ziehen. Anschließend können Sie das Image auswerfen und PyCharm aus Ihrem Applications-Ordner heraus starten.

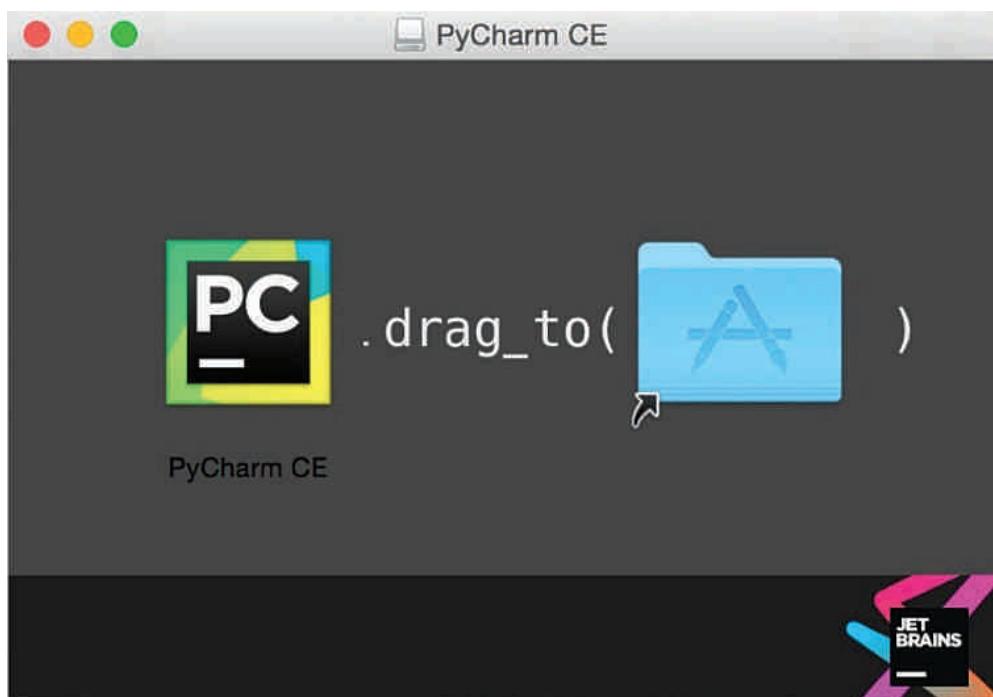


Abb. 2.9 PyCharm unter macOS installieren

## 2.4 PyCharm anpassen

Wenn Sie PyCharm zum ersten Mal starten, werden Sie gefragt, ob Sie bereits vorhandene Einstellungen einer alten Installation verwenden möchten. Klicken Sie **Do not import settings** und bestätigen Sie mit **OK**. MacOS-Nutzer können anschließend zwischen verschiedenen Tastaturlayouts wählen. Im nächsten Schritt können Sie ein Farbschema auswählen. Mit **Skip Remaining and Set Defaults** überspringen Sie die weiteren Einstellungen, die für uns nicht relevant sind. Ihre Einstellungen können Sie auch im Nachhinein jederzeit über das Einstellungsmenü einsehen und ändern. In das Einstellungsmenü gelangen Sie über das Hauptmenü (über **File → Settings...** bzw. macOS: **PyCharm → Preferences**). Hier müssen Sie nun den gewünschten Interpreter auswählen, den Sie zuvor installiert haben. Das geht über den Eintrag **Project → Project Interpreter** in der Auswahl auf der linken Seite. Anschließend sollte eine neue Auswahlbox mit dem installierten Interpreter erscheinen. Ist dieser nicht in der Liste enthalten, wählen Sie **Show All...** und die Schaltfläche mit dem Plus-Zeichen. Gehen Sie im neuen Fenster links auf **System Interpreter** und schließlich auf **Python 3.7**.

## 2 Die Vorbereitung

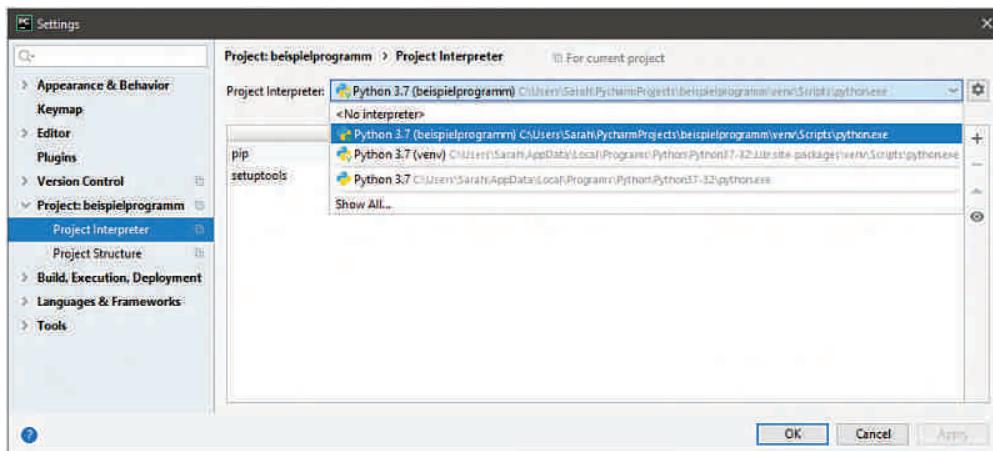


Abb. 2.10 Angabe des gewünschten Interpreters

Nun wird standardmäßig bei jedem neuen Projekt automatisch der zuvor installierte Interpreter verwendet. Sollten Sie für ein Projekt einen anderen Interpreter (beispielsweise Python 2.7) nutzen wollen, so können Sie dies in den Projekteinstellungen wieder ändern.

Über **File → Settings...** können Sie viele weitere Einstellungen vornehmen. Möchten Sie beispielsweise die Rechtschreibprüfung, die standardmäßig eingeschaltet ist, wieder deaktivieren, müssen Sie hierfür in der linken Auswahl unter **Editor → Inspections** den Haken rechts bei **Spelling → Typo** herausnehmen.

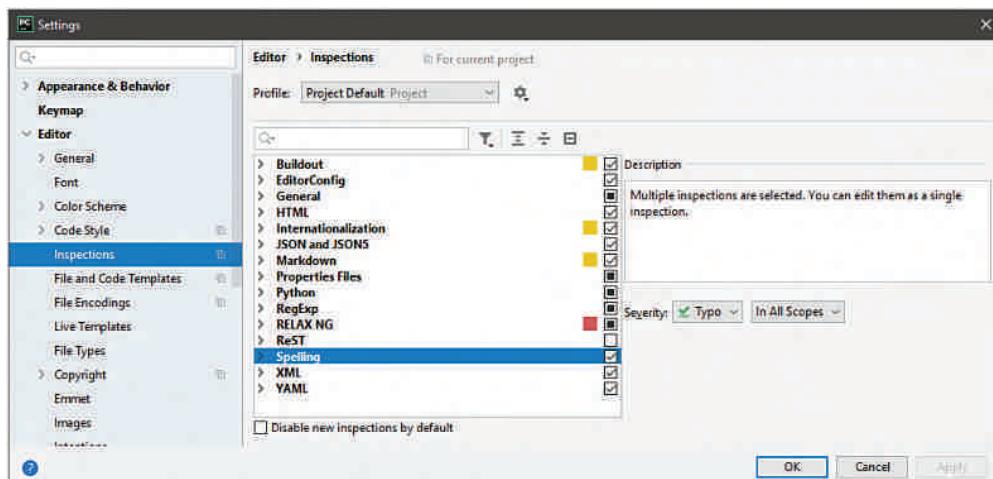


Abb. 2.11 Editor-Einstellungen ändern

Die Anzeige von Zeilennummern ist im Grunde eine sehr hilfreiche Option. Wenn Sie sie dennoch ausschalten möchten, können Sie sie unter **Editor → General → Ap-**

## 2.4 PyCharm anpassen

**pearance → Show line numbers** deaktivieren. Unter dem **Editor**-Eintrag lassen sich viele weitere Einstellungen, etwa zur Schrift und Farbdarstellung, anpassen.

Nun steht dem eigentlichen Programmieren nichts mehr im Wege. Bevor wir in die faszinierenden Möglichkeiten des Programmierens mit Python eintauchen und unsere ersten längeren Programme in der IDE schreiben, ausführen und überarbeiten lernen, werden wir im nächsten Kapitel einige grundlegende Befehle in einem Python-Prompt ausprobieren. Ab Kapitel 4 werden wir zur IDE wechseln und uns mit ihren unterschiedlichen Funktionen vertraut machen.

## Kapitel 3

# Der interaktive Modus: ideal für den ersten Kontakt mit Python

Python ist eine der wenigen Programmiersprachen, die zusätzlich zur Standard-Programmausführung durch einen Compiler oder einen Interpreter eine weitere Möglichkeit anbieten: die interaktive Interpretation. Hierbei schreibt man den Programmcode direkt in den Interpreter (auch Python-Shell genannt) und führt ihn mit der **Enter**-Taste sofort aus. Diese Eingabemethode ist in vielen Situationen von Vorteil, zum Beispiel, wenn man:

- ▶ bestimmte Programmteile separat vom übrigen Programmcode kurz testen möchte. Dabei genügt es, den gewünschten Ausschnitt des Gesamtprogramms in den Interpreter einzugeben, sodass man nicht das gesamte Programm ausführen muss. Auch professionelle Entwickler und Entwicklerinnen machen gerne von dieser zeit- und ressourcensparenden Möglichkeit Gebrauch, etwa um verschiedene Alternativen und deren Funktionsweise auszuprobieren. Gerade bei komplexen Strukturen, bei denen die Lösung nicht unmittelbar ersichtlich ist, kann dies den Entwicklungsprozess beschleunigen.
- ▶ als Anfänger verschiedene Befehle spielerisch kennen lernen möchte, ohne dafür eine Datei anlegen zu müssen. Die Befehle kann man so ganz einfach direkt in den Interpreter eingeben; der Output wird anschließend sofort angezeigt. Diese experimentelle Herangehensweise hat zudem den Vorteil, dass Fehler in Programmteilen leichter zu erkennen sind.

Da sich das Erlernen einer neuen Programmiersprache im interaktiven Modus in der Regel schneller und einfacher gestaltet, werden wir unsere ersten Python-Befehle auf diese Weise eingeben. Sie sollten sich jedoch dessen bewusst sein, dass man mit dieser Programmiermethode keine dauerhaft abrufbaren Programme erstellen kann, da die Befehle nach ihrer Ausführung nicht gespeichert werden. Möchten Sie diese auch zu einem späteren Zeitpunkt wiederholen können, müssen Sie sie in den Texteditor von PyCharm oder einer anderen von Ihnen gewählten IDE schreiben und abspeichern.

### 3.1 Den Python-Prompt aufrufen

Um mit der interaktiven Programmierung anfangen zu können, müssen Sie den Python-Prompt öffnen. In Kapitel 2.2. hatten wir dies bereits getan, als wir nach der Installation des Python-Interpreters überprüften, ob dieser korrekt installiert wurde.

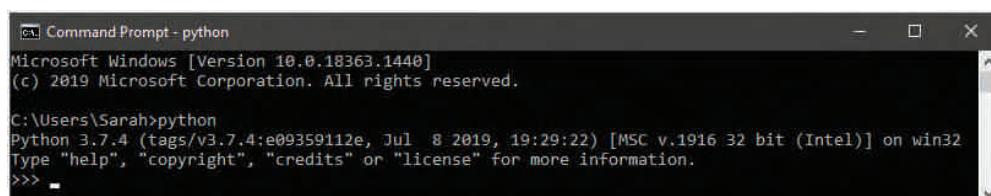
## 3.2 Erste Befehle im Python-Prompt

**Zur Wiederholung:** Wir öffnen zunächst einen Kommandozeileninterpreter, auch *Eingabeaufforderung* genannt, um darin den Python-Prompt aufzurufen.

- ▶ Unter Windows gelangen Sie am schnellsten zur Eingabeaufforderung, indem Sie in der Windows-Suche im Startmenü die Buchstaben **cmd** eingeben.
- ▶ Unter Linux müssen Sie ein Terminal-Fenster öffnen. Eine der Möglichkeiten, dies zu tun, ist durch die Tastenkombination **Ctrl + Alt + T**.
- ▶ Unter macOS klicken Sie im **Applications**-Ordner den Ordner **Utilities** an. In diesem befindet sich das Programm **Terminal**.

In der Eingabeaufforderung werden ganz oben die Versionsangabe des Betriebssystems und Copyright-Informationen angezeigt. Darunter erscheint das Verzeichnis, in dem sich das Programm befindet. Geben Sie nun den Befehl `python` (Windows) bzw. `python3` (Linux, macOS) ein und bestätigen Sie Ihre Eingabe durch **Enter**. Je nach Betriebssystem und Python-Version sollte daraufhin eine ähnliche Ausgabe wie die folgende erscheinen:

3



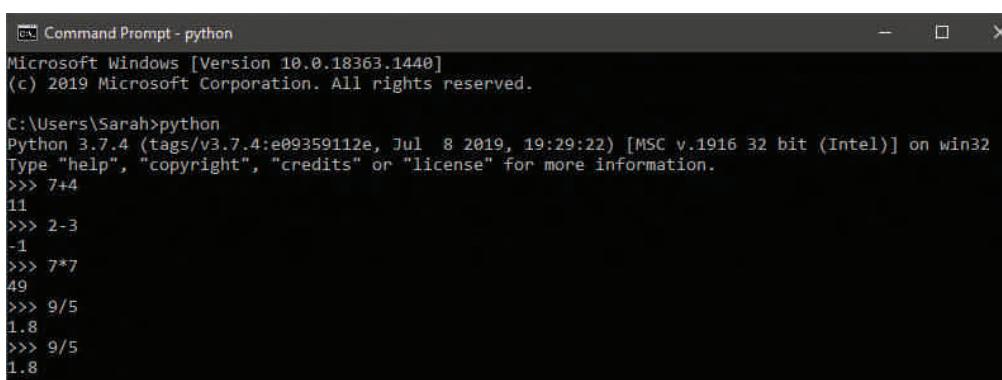
```
Command Prompt - python
Microsoft Windows [Version 10.0.18363.1440]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Sarah>python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Abb. 3.1 Der Python-Prompt für die interaktive Interpretation

## 3.2 Erste Befehle im Python-Prompt

Lassen Sie uns nun verschiedene Eingaben im Python-Prompt ausprobieren. Da Ihnen womöglich noch keinerlei Befehle bekannt sind, beschränken wir uns zunächst auf einfache Rechenoperationen: Geben Sie einfache Berechnungen wie  $7+4$ ,  $2-3$ ,  $7 \cdot 7$  oder  $9 / 5$  ein, jeweils gefolgt von **Enter**.



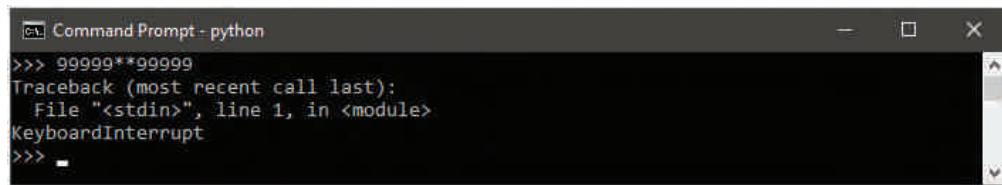
```
Command Prompt - python
Microsoft Windows [Version 10.0.18363.1440]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Sarah>python
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 7+4
11
>>> 2-3
-1
>>> 7*7
49
>>> 9/5
1.8
>>> 9/5
1.8
```

Abb. 3.2 Einfache Rechenaufgaben im Python-Prompt

### 3 Der interaktive Modus: ideal für den ersten Kontakt mit Python

Nach jeder Eingabe wird nun in der nächsten Zeile direkt das Ergebnis der Rechenaufgabe ausgegeben. Wie Sie sehen, können Sie Python sehr gut anstelle eines Taschenrechners verwenden. Hierbei lassen sich alle Grundrechenarten durchführen, wobei wie bei den meisten Programmiersprachen das Sternsymbol (\*) für die Multiplikation und der Schrägstrich (/) für Division stehen. Sollte eine Berechnung einmal zu lange benötigen, können Sie diese mit der Tastenkombination **Strg + C** unterbrechen. Beispielsweise führt bereits die Berechnung des Ergebnisses von  $99999^{99999}$ , die in Python durch den Befehl `99999**99999` ausgedrückt wird, zu einer Berechnung, die mehrere Sekunden benötigt.



```
Command Prompt - python
>>> 99999**99999
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
>>> -
```

**Abb. 3.3** Längere Berechnungen durch Strg + C unterbrechen

Auch die gängigen mathematischen Berechnungen mit Klammern lassen sich in der Python-Shell ausführen. Eine praktische Option ist zudem die Verwendung des Unterstrichs. Darin wird jeweils der letzte berechnete Ausgabewert gespeichert, sodass dieser nicht händisch eingetippt werden muss:



```
Command Prompt - python
>>> 2*(5 + 7)
24
>>> _
27
>>> -
```

**Abb. 3.4** Klammern und Unterstrich in Berechnungen

Wir gehen noch einen Schritt weiter und führen sogleich Variablen ein, da diese sich gut für den spielerischen Einstieg in die interaktive Interpretation eignen. In Kapitel 5 werden wir das Thema Variablen ausführlicher behandeln.

Angenommen, wir möchten der Variablen `x` den Wert 7 zuweisen. Hierfür geben wir `x = 7` ein und bestätigen mit **Enter**. Python gibt hiernach in der nächsten Zeile nichts aus, sondern speichert nur ab, dass es eine Variable `x` gibt, die den Wert 7 hat. Um uns den Wert von `x` erneut anzeigen zu lassen, müssen wir lediglich erneut den Variablennamen `x` eingeben und mit der Entertaste bestätigen. Diese besondere Methode, eine Variable sofort anzuzeigen, funktioniert übrigens nur im interaktiven Modus. In der IDE wäre hierfür ein eigener Befehl, der `print`-Befehl, nötig.

## 3.2 Erste Befehle im Python-Prompt

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

**Abb. 3.5** Die Verwendung einer Variablen

In Python lassen sich mit Variablen Rechenaufgaben aller Art durchführen. Geben Sie für das nächste Rechenbeispiel folgende Befehle hintereinander ein:

```
1 x = 11
2 x+6
3 y = x*7
4 y
5 y = y+11
6 y
```

3

```
>>> x = 11
>>> x+6
17
>>> y = x*7
>>> y
77
>>> y = y+11
>>> y
88
```

**Abb. 3.6** Verschiedene Operationen mit Variablen

Wie Sie sehen, wird die Variable `y` in der dritten Eingabezeile zunächst durch die Multiplikation von `x` mit der Zahl 7 definiert. Im Anschluss an den nächsten Befehl, `y`, wird sodann der für `y` ermittelte Wert 77 ausgegeben. Interessant sind hier auch die beiden letzten Eingabezeilen, die – streng genommen – einer mathematisch unkorrekten Logik folgen: Die Variable wird durch eine Rechenoperation mit sich selbst neu definiert, woraufhin sich ihr Wert ändert.

Sollte Ihnen die Funktionsweise der Variablenoperationen an dieser Stelle noch unklar sein, ist das gar kein Problem. Im weiteren Verlauf dieses Buchs werden wir uns eingehender mit der Handhabung von Variablen befassen und diese genauer erklären.

**Tipp:** In der Regel muss man den Programmcode beim Programmieren sehr oft ausprobieren, abändern und erneut ausführen. Damit man dabei den Befehl nicht jedes Mal erneut händisch eintippen muss, kann man sich im Kommandozeileninterpretier durch einmaliges Drücken auf die Pfeiltaste nach oben ( $\uparrow$ ) den zuletzt verwendeten Befehl anzeigen lassen. Durch erneutes Betätigen der Pfeiltaste nach

### 3 Der interaktive Modus: ideal für den ersten Kontakt mit Python

oben wird der vorletzte Befehl angezeigt, usw. Zusammen mit der Pfeiltaste nach unten (**↓**) kann man so leichter zwischen bereits verwendeten Eingaben navigieren oder Tippfehler schnell korrigieren, was eine erhebliche Arbeitserleichterung bedeutet.

Mit den Befehlen `exit()` oder `quit()` gefolgt von **Enter** können Sie den Python-Prompt wieder verlassen und zur Kommandozeile zurückkehren.

## Kapitel 4

# Python-Programme in eine Datei schreiben

Der interaktive Modus ist sehr hilfreich, um erste Erfahrungen mit Python zu sammeln. Im Python-Prompt können Sie spielerisch grundlegende Befehle kennen lernen und bei Bedarf verschiedene Bestandteile des Programms testen. Sobald Sie jedoch den Python-Prompt schließen, gehen alle Informationen und eingegebenen Befehle verloren. Wenn Sie also ein längeres Programm schreiben möchten, das auch hinterher noch jederzeit beliebig oft aufrufbar sein soll, ist die interaktive Interpretation nicht mehr ausreichend.

Bei der klassischen Form des Programmierens wird der Quellcode daher in eine separate Datei geschrieben. Wir werden diese Methode im weiteren Verlauf des Buchs für alle Beispiele und Übungsaufgaben verwenden. Sie sollten trotzdem nicht vergessen, welche Vorteile Ihnen die interaktive Interpretation bietet und dass Sie diese beispielsweise später bei komplexeren Programmen geschickt einsetzen können, um einzelne Bestandteile interaktiv auszuprobieren.

4

### 4.1 Ein Programm für eine einfache Textausgabe erstellen

Wir werden nun unser erstes Programm in Python schreiben. Hierfür müssen Sie zunächst PyCharm öffnen und links oben im Fenster auf **File → New Project...** klicken. Es öffnet sich ein neues Fenster, in dem der Pfad für Ihre PyCharm-Projekte angezeigt wird. Geben Sie den Dateienamen für das neue Projekt ein. Dieser ist frei wählbar. Wir wählen hier *Hallo Welt*. Klicken Sie sodann rechts unten auf **Create**.

Im linken Navigationsbereich sehen Sie nun die Informationen zum neuen Hallo-Welt-Projekt. Es besteht aus mehreren Ordner und Dateien, die unter anderem die Einstellungen des Projekts beinhalten. Das eigentliche Python-Programm ist eine Datei mit der Endung *.py*, die noch angelegt werden muss. Hierfür klicken Sie im linken Navigationsbereich mit der rechten Maustaste auf den gewählten Projektnamen *Hallo Welt* und wählen im Aufklappmenü unter dem ersten Eintrag **New → Python File**. Hier geben Sie den Namen des Programms ein. Der Programmname kann dabei vom Projektnamen abweichen. Dies ist beispielsweise sinnvoll, wenn Sie innerhalb eines Projekts mehrere Programme anlegen möchten. Der Einfachheit halber wählen wir hier denselben Namen. Nachdem Sie mit **OK** bestätigt haben, wird die *.py*-Datei angelegt. Der Editor öffnet sich rechts in einem neuen Fenster.

## 4 Python-Programme in eine Datei schreiben

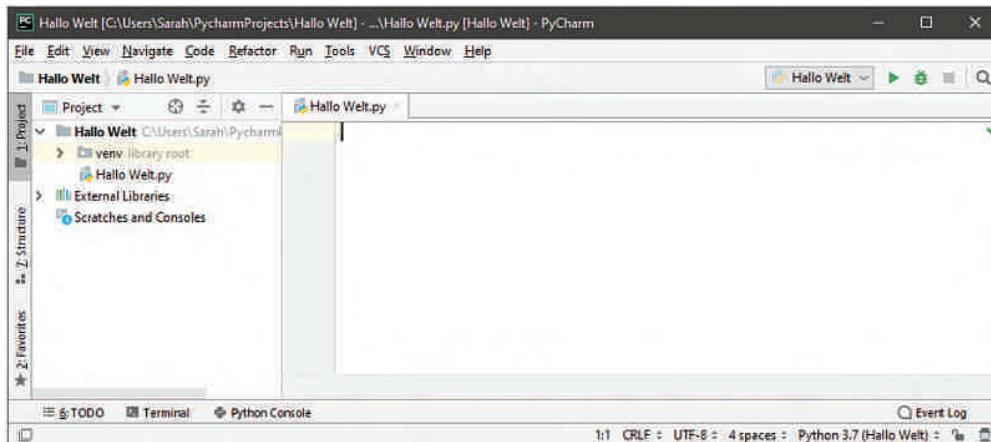


Abb. 4.1 Eine Programmdatei in PyCharm anlegen

Nun tragen wir in das Editorfenster die folgende Zeile ein: `print ("Hallo Welt!")`. Mit dieser einen Zeile haben Sie bereits Ihr erstes Python-Programm geschrieben! Der `print`-Befehl bewirkt hierbei, dass der entsprechende Inhalt in den Klammern in der Programmausgabe angezeigt wird. Um sich das Ergebnis anzeigen zu lassen, gehen Sie oben in der Menüzeile auf **Run → Run** und bestätigen Sie den Programmnamen, indem Sie ihn anklicken. Nun öffnet sich im unteren Fensterbereich von PyCharm das Ausgabefenster, in dem der gewünschte Text angezeigt wird:



Abb. 4.2 Die Textausgabe des Hallo-Welt-Programms im unteren Fensterbereich

Beim *Hallo-Welt*-Programm (bzw. *Hello-World*-Programm) handelt es sich um ein klassisches Programmbeispiel, das erstmals 1974 in einem Handbuch von Bell Laboratories erschien. Es wurde ursprünglich in der Sprache *C* geschrieben und existiert mittlerweile in gut über 100 Programmiersprachen. Anhand des *Hallo-Welt*-Programms lässt sich auf einfache Art und Weise veranschaulichen, aus welchen Befehlen oder

## 4.1 Ein Programm für eine einfache Textausgabe erstellen

Komponenten ein fertiges Programm in unterschiedlichen Sprachen bestehen muss. Es gewährt somit einen grundlegenden Einblick in die programmspezifische Syntax.

Wie Sie sehen, benötigt Python lediglich eine einzige Zeile, um ein vollständiges Hallo-Welt-Programm zu erzeugen, was erneut die einfache Struktur von Python verdeutlicht. Die meisten anderen Programmiersprachen würden für diese einfache Textausgabe mehrere Zeilen Code benötigen, was insbesondere Anfängerinnen die erste Anwendung unnötig verkomplizieren würde. Python verzichtet jedoch auf diesen Ballast – der Befehl lässt sich hier ohne weitere Zusätze verwenden.

### 4.1.1 PEP – Guidelines zum Layout und Styling von Code

Obwohl wir einleitend bemerkten, dass Python der Einfachheit und Übersichtlichkeit der Syntax halber weitestgehend auf Klammern verzichtet, sind Klammern bei einigen Befehlen und Funktionen dennoch notwendig – zum Beispiel beim oben verwendeten `print`-Befehl. Das ist darin begründet, dass dieser Befehl aus mehreren Bestandteilen bestehen kann. Später in diesem Buch wollen wir der Ausgabe beispielsweise Variablen und weitere Textblöcke hinzufügen. Das Programm muss hier stets erkennen können, an welchem Punkt die Ausgabe anfängt und wo sie endet. Hierfür kommt die runde Klammer zum Einsatz.

4

Der Text innerhalb der Klammern, der die Ausgabe darstellt, muss in Anführungszeichen gesetzt werden. Sie können dabei entweder einfache oder doppelte Anführungszeichen verwenden – die Funktionsweise ist gleich. Enthält der Ausgabetext jedoch selbst Anführungszeichen, so müssen Sie hierfür die Anführungszeichen vom jeweils anderen Typ benutzen. Setzt man hier beide Male die gleichen Anführungszeichen, so wird das Programm diese als das Ende des Texts interpretieren, was zu einer Fehlermeldung oder einer unkorrekten Darstellung führt. Folgende Schreibweise würde also beispielsweise einen Fehler produzieren:

```
1 print("Hallo "Welt!" ")
```

Verwendet man für den gesamten Textteil einfache Anführungszeichen, stimmt die Ausgabe:

```
1 print("Hallo 'Welt!' ")
```

Probieren Sie dies einmal selbst in PyCharm aus!

Python erkennt Leerzeichen nur innerhalb einer Zeichenkette und gibt diese im Output ihrer Anzahl gemäß als solche aus. Im restlichen Programm werden Leerzeichen ignoriert und führen zu keinerlei Fehlermeldungen, sofern sie nicht am Anfang einer Zeile eine Einrückung kennzeichnen. Für derartige und viele weitere designspezifi-

## 4 Python-Programme in eine Datei schreiben

sche Zwecke beruht Python auf unterschiedlichen Design-Dokumenten, die unter dem Namen *PEP* zusammengefasst sind. „PEP“ steht dabei für *Python Enhancement Proposal*. In der PEP-Dokumentation ist unter anderem genau vorgegeben, wie der Code und seine Komponenten geschrieben, eingerückt und formatiert sein sollen.

So ist *PEP 8* beispielsweise ein Styleguide, der das spezifische Layout- und Stilformat von Code vorgibt. Zwar handelt es sich dabei nicht um ein striktes Regelwerk, dessen Nichteinhaltung zu fehlerhaftem Code führt. PEP 8 dient jedoch der besseren Lesbarkeit und einheitlichen Formatierung von Code und sollte daher in jedem Fall eingehalten werden. Ist Ihr Code getreu PEP 8 formatiert, erkennen Sie dies in PyCharm beispielsweise daran, dass im oberen rechten Rand des Code-Eingabefensters ein grünes Häkchen (✓) zu sehen ist. Ist dies nicht der Fall, erscheinen dort ein Ausrufezeichen oder ein Kasten sowie entlang des Scrollbalkens Hinweise zu denjenigen Stellen, die laut PEP 8 einer Änderung bedürfen. Mehr Informationen über PEP 8 finden Sie unter <https://www.python.org/dev/peps/pep-0008/>.

### 4.2 Die Ausführung im Python-Interpreter

Wir haben nun unser erstes Programm geschrieben und in der IDE ausgeführt. Es gibt jedoch ebenfalls die Möglichkeit eine *.py*-Datei ohne IDE direkt aus der Kommandozeile auszuführen. Diese Option ist besonders dann hilfreich, wenn Sie ein Programm zur Ausführung an eine andere Person weitergeben wollen, auf deren Computer keine Entwicklungsumgebung installiert ist. Diese Person benötigt dann lediglich einen installierten Python-Interpreter, um das Programm starten zu können. Obwohl die Programmausführung aus der Kommandozeile eher unpraktisch ist, wenn Ihr Computer über eine IDE verfügt, soll sie hier der Vollständigkeit halber erwähnt sein. In den Folgekapiteln werden wir jedoch den Weg der Programmausführung über die IDE wählen.

Nachdem Sie die Kommandozeile (etwa durch den Befehl **cmd**) geöffnet haben, zeigt diese zunächst das Stammverzeichnis an. Um unser Python-Programm *Hallo Welt.py* auszuführen, müssen wir in das Verzeichnis wechseln, in dem wir das Programm abgespeichert haben. Das gelingt mit dem Befehl **cd** (engl. für *change directory*) gefolgt vom Namen des Pfads, in dem sich die gewünschte Datei befindet. In der Pfadangabe werden die einzelnen Verzeichnisse durch einen Backslash (\) voneinander getrennt. Wenn Sie als Betriebssystem Windows verwenden, müssen Sie hierbei zudem darauf achten, Standardverzeichnisse stets mit ihrer englischen Bezeichnung anzugeben. Das Verzeichnis „Dokumente“ im Windows-Explorer wird beispielsweise „Documents“ im Kommandozeileninterpreter heißen.

Wenn Sie sich nicht mehr an den genauen Ort erinnern, in dem Sie die Datei *Hallo Welt.py* abgespeichert haben, können Sie innerhalb eines Ordners den Befehl **dir** eingeben, woraufhin der gesamte Ordnerinhalt angezeigt wird. In diesem Bei-

### 4.3 Kommentare: hilfreich für das Verständnis des Programms

spiel befindet sich das Python-Programm im Projektverzeichnis *PycharmProjects\Hallo Welt*, wo wir es im vorigen Abschnitt angelegt hatten. Wir empfehlen Ihnen, Ihre künftigen Programme ebenfalls in diesem Ordner oder darin erstellten Unterverzeichnissen abzuspeichern. Um zum entsprechenden PycharmProjects-Ordner zu wechseln, müssen Sie folgenden Befehl eingeben:

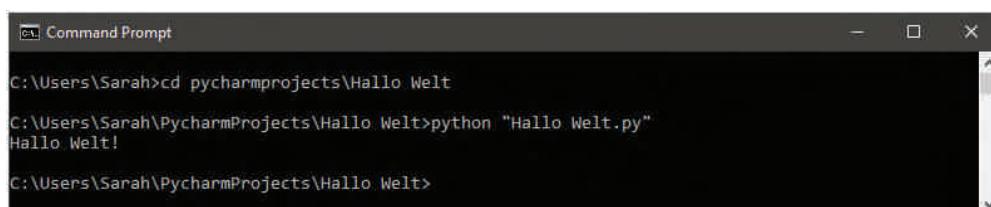
```
1 cd pycharmprojects\Hallo Welt
```

Jetzt müssen Sie nur noch die entsprechende Programmdatei starten. Dazu geben Sie ganz einfach `python` gefolgt vom Dateinamen ein:

```
1 python "Hallo Welt.py"
```

Der Dateiname muss hier in Anführungszeichen gesetzt werden, da er aus mehreren Wörtern besteht und sonst nicht als Datei erkannt werden kann. Aus diesem Grund ist es ratsamer, stets Programmnamen zu wählen, die lediglich aus einem Wort bestehen. Nachdem Sie den Befehl bestätigt haben, wird der Text im selben Fenster ausgegeben. Im Anschluss wird das Programm beendet. Der Interpreter wartet auf neue Befehle.

4



```
C:\Users\Sarah>cd pycharmprojects\Hallo Welt
C:\Users\Sarah\PycharmProjects\Hallo Welt>python "Hallo Welt.py"
Hallo Welt!
C:\Users\Sarah\PycharmProjects\Hallo Welt>
```

**Abb. 4.3** Die Ausführung des ersten Programms aus der Kommandozeile

Sie haben nun gesehen, wie Sie ein Python-Programm entweder direkt in der IDE oder in der Kommandozeile ausführen können. Während es praktischer ist, das komplette Programm direkt in der IDE zu schreiben und auszuführen, ist es dennoch sinnvoll, über diese zusätzliche Möglichkeit der Programmausführung Bescheid zu wissen. Wir werden unsere Programme fortan innerhalb der IDE ausführen.

### 4.3 Kommentare: hilfreich für das Verständnis des Programms

Oftmals möchte man bestehende Programme nach einiger Zeit nochmals überarbeiten. So ist es etwa denkbar, dass ein Unternehmen aufgrund eines aktualisierten Wissensstands einen Algorithmus im Programm anpassen oder bestimmte Prozessabläufe darin verändern will. Hierfür muss meist kein komplett neues Programm geschrieben werden – es reicht, Teilbereiche des Codes abzuändern, womit viel Zeit und Mühe gespart ist.

## 4 Python-Programme in eine Datei schreiben

Im Nachhinein die oftmals abstrakten Funktionen und genauen Details des geschriebenen Codes nachzuvollziehen, kann dabei mitunter schwierig sein – insbesondere, wenn es sich bei dem Code um Programmcode einer fremden Person handelt, die eventuell einen anderen Programmierstil hat.

Hier kommen Kommentare ins Spiel. Kommentare bestehen aus gewöhnlichem Text, der in den Quellcode eingefügt wird, jedoch keinerlei Auswirkung auf die Ausführung des Programms hat. Sie werden durch das Raute-Zeichen (#) eingeleitet. Man kann sie überall dort einfügen, wo einzelne Programmteile und deren Bedeutung im Code genauer erklärt werden sollen. Kommentare helfen dabei, bestimmte Stellen im Code leichter zu finden, zu verstehen oder anzupassen. Man sollte jedoch immer darauf achten, sie knapp und klar verständlich zu halten, und nach Änderungen im Code auch die Kommentare dementsprechend zu aktualisieren.

Allgemein gilt es als gute Praxis, am Anfang jedes komplexeren Programms einen Kommentar einzufügen, der die Aufgabe und Funktionsweise des Programms kurz beschreibt. Auch während des Schreibens von Code sind Kommentare nützlich, beispielsweise, wenn Sie eine Stelle zum nachträglichen Bearbeiten markieren möchten. Kommentare ermöglichen zudem das Auskommentieren: Wenn Sie einen Programmteil umgeschrieben haben, sich aber noch nicht sicher sind, ob er wirklich so funktioniert wie gewünscht, so können Sie die alte Version vorübergehend zu einem Kommentar machen und die neue Version testen, bis Sie sicher sind, dass die alte Version endgültig gelöscht werden kann.

Anfänger sollten sich so früh wie möglich daran gewöhnen, Kommentare in ihren Code einzubauen, sodass dieser leicht verständlich bleibt. In diesem Buch werden Sie an einigen Stellen Kommentare im Quellcode finden, die die Aufgaben der einzelnen Programmteile direkt bei deren Einführung erklären. Diese Kommentare müssen Sie nicht unbedingt mit abtippen. Spätestens dann, wenn Sie Ihr eigenes Programm schreiben, sollten Sie jedoch mit Kommentaren arbeiten. Auch, wenn Sie nach dem Lesen dieses Buches Ihre Programmierkenntnisse mithilfe von Lernmaterial aus dem Internet erweitern möchten, werden Sie feststellen, dass diese Online-Beispiele meist ebenfalls mit Kommentaren versehen sind.

Wie bereits erwähnt, werden Kommentare in Python durch das Raute-Zeichen (#) gekennzeichnet. Es gilt jeweils für den Text, der hinter dem Raute-Zeichen steht und bezieht sich dabei nur auf die entsprechende Zeile. Die Ausführung des Programms ändert sich durch Kommentare nicht. Die einzige Ausnahme ist hier, wenn das Raute-Zeichen innerhalb von Anführungszeichen steht: In diesem Fall wird es als Text interpretiert und in der Ausgabe mit angezeigt.

Unser erstes Programm mit einem eingefügten Kommentar könnte etwa wie folgt aussehen:

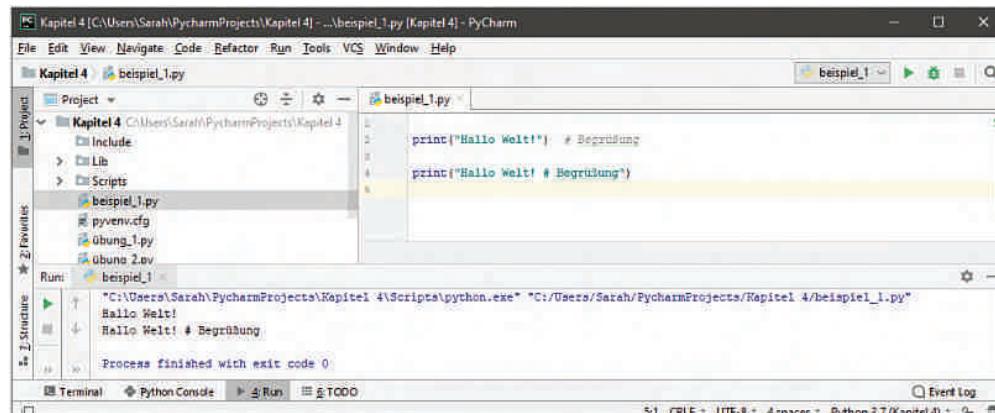
### 4.3 Kommentare: hilfreich für das Verständnis des Programms

#### Codebeispiel #1

```
1 print("Hallo Welt!") # Begrüßung
```

Wie Sie sehen, ändert der Kommentar nichts an der Ausführung des `print`-Befehls. Setzt man den Kommentar jedoch mit in die Anführungszeichen, erscheint er ebenfalls im ausgegebenen Text:

```
1 print("Hallo Welt! # Begrüßung")
```



4

**Abb. 4.4** Die Verwendung des Kommentar-Zeichens innerhalb der Anführungszeichen

Kommentare müssen also nach dem eigentlichen Programmteil am Ende der Zeile stehen. Längere Kommentare, etwa am Anfang oder am Ende des Programms, können mitunter auch eine oder mehrere eigene Zeilen einnehmen.

Achten Sie stets darauf, nur sinnvolle Kommentare in den Quellcode einzufügen. Der obige Begrüßungs-Kommentar wäre beispielsweise in Wirklichkeit überflüssig, da man bereits beim Lesen des `print`-Befehls versteht, was in dieser Zeile geschieht.

Möchten Sie Kommentare einfügen, die über mehrere Zeilen gehen, oder möchten Sie mehrere Zeilen Code ausklammern, so müssen Sie nicht jede einzelne Zeile mit dem `#`-Zeichen als Kommentar auszeichnen. Stattdessen können Sie drei einfache ('''') oder doppelte Anführungszeichen ("""") jeweils vor und hinter den zu kommentierenden Teil setzen:

```
1 """
2 Dies ist ein
3 mehrzeiliger Kommentar.
4 """
5 """
6 Dies ist auch ein
7 mehrzeiliger Kommentar.
8 """
```

## 4 Python-Programme in eine Datei schreiben

Kommentare, die mit drei Anführungszeichen ausgewiesen werden, nennt man auch *Docstrings*. In Kapitel 11.2.1. werden wir nochmals genauer auf dieses Thema eingehen.

### 4.4 Übung: Eigene Inhalte zum Programm hinzufügen

Von nun an werden Sie am Ende jedes Kapitels einige Übungsaufgaben vorfinden. Die Aufgaben dienen dazu, das Gelernte zu vertiefen und selbst anwenden zu können. Sie werden darin schwerpunktmäßig diejenigen Themen vorfinden, die im jeweiligen Kapitel behandelt wurden. Selbstverständlich bauen die Übungen inhaltlich ebenfalls auf dem Wissen aus früheren Kapiteln auf. Sollte für die erfolgreiche Bearbeitung einer Aufgabe einmal Zusatzwissen vorgenötigt sein, das bislang nicht vorkam, werden wir dies in einer Anmerkung an Ort und Stelle kurz erklären.

Im Anschluss an die Aufgaben folgt stets eine Musterlösung. Wir empfehlen Ihnen, zunächst zu versuchen, jede Aufgabe selbstständig zu lösen und die Musterlösungen nur dann zurate zu ziehen, wenn Sie wirklich nicht selbst auf die Lösung kommen. Haben Sie die Übung eigenständig gelöst, können Sie Ihre Lösung anschließend mit der Musterlösung vergleichen. Beachten Sie, dass es mitunter mehrere mögliche Lösungswege geben kann und dass die Musterlösung nur eine dieser verschiedenen Möglichkeiten darstellt. Wenn Sie mit Ihrer Lösung die Aufgabe korrekt umgesetzt und ein funktionsfähiges Programm geschrieben haben, das das gewünschte Ergebnis liefert, so ist dieser Weg ebenfalls richtig, selbst wenn er von der Musterlösung abweicht.

#### 4.4 Übung: Eigene Inhalte zum Programm hinzufügen

### Übungsaufgaben

1. Stellen Sie sich vor, Sie sind Programmiererin bzw. Programmierer und wurden von einer großen Bibliothek damit beauftragt, ein Programm für die Verwaltung ihres Literaturbestands zu schreiben. Erstellen Sie eine Überschrift, eine Begrüßungs-Anrede und einen kurzen Überblick über die Aufgaben, die das Programm übernimmt. Diese sollen jeweils einen eigenen `print`-Befehl verwenden.

**Tipp:** Wenn der Inhalt zu lang für eine einzelne Zeile ist, können Sie diesen in zwei Abschnitte unterteilen. Jeder Abschnitt muss in Anführungszeichen stehen. Verbinden Sie die Abschnitte mit einem Plus-Zeichen.

2. Fügen Sie einen Kommentar in das Verwaltungs-Programm ein, der angibt, welche Aufgaben hierfür noch zu erledigen sind.
3. Erstellen Sie ein neues Programm, das aus zwei `print`-Befehlen besteht. Der Inhalt dieser Befehle soll jeweils  $7 + 4$  lauten – allerdings einmal in Anführungszeichen und einmal ohne Anführungszeichen. Führen Sie das Programm aus und überlegen Sie sich, worauf die unterschiedlichen Ausgaben zurückzuführen sind.

## 4 Python-Programme in eine Datei schreiben

### Lösungen

1.

```
1 print("Literatur-Verwaltungsprogramm")
2 print("Herzlich Willkommen zur Verwaltung Ihres Literaturbestands")
3 print("Mit diesem Programm können Sie neue Werke hinzufügen, " +
4      "Werke aus dem Bestand entfernen und Werkinformationen ändern")
```

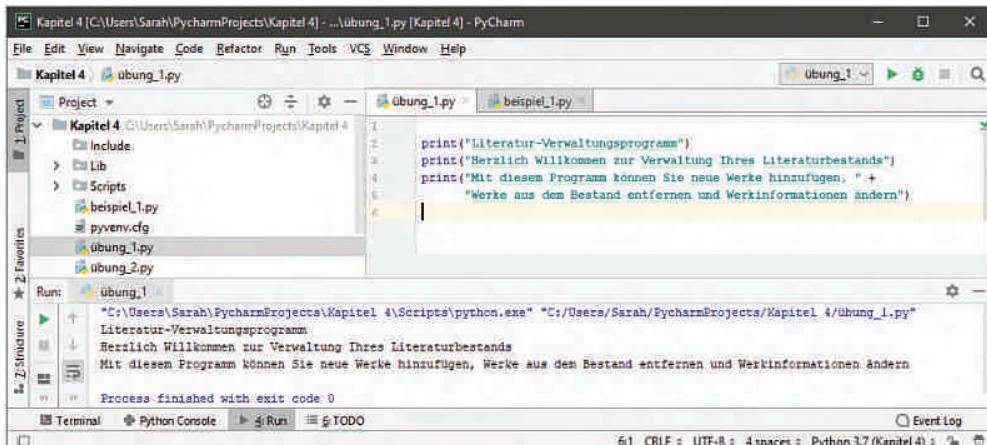


Abb. 4.5 print-Befehle des Literatur-Verwaltungsprogramms

2.

```
1 print("Literatur-Verwaltungsprogramm")
2 print("Herzlich Willkommen zur Verwaltung Ihres Literaturbestands")
3 print("Mit diesem Programm können Sie neue Werke hinzufügen, " +
4      "Werke aus dem Bestand entfernen und Werkinformationen ändern")
5
6 # To-do: Funktionen für das Entfernen & Hinzufügen von Werken und für die
7 Eingabe von Werkinformationen
```

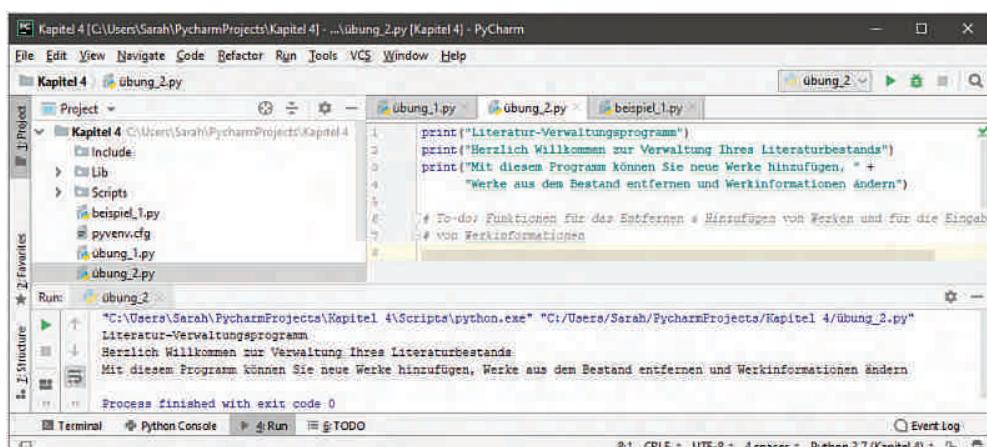


Abb. 4.6 Der Kommentar ändert nichts an der Ausgabe

#### 4.4 Übung: Eigene Inhalte zum Programm hinzufügen

3.

```
1 print("7+4")
2 print(7+4)
```



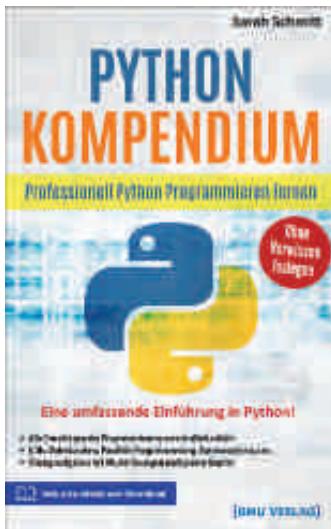
```
Command Prompt - python
>>> print("7+4")
7+4
>>> print(7+4)
11
>>>
```

Abb. 4.7 Die Verwendung von Anführungszeichen verändert die Ausgabe

Zahlen oder Buchstaben, die in Anführungszeichen stehen, gibt der Interpreter unverändert als Zeichenkette wieder. Werden keine Anführungszeichen gesetzt, so geht er davon aus, dass es sich hierbei um eine Rechenaufgabe handelt. In diesem Fall wird das Ergebnis aus  $7+4$  berechnet und ausgegeben. Wenn man im obigen Fall anstatt der Rechenaufgabe eine zuvor definierte Variable in die Klammer setzt, wird der Wert der Variablen ausgegeben. Schreibt man einen Funktionsnamen (Kapitel 9) in die Klammer, berechnet der Interpreter das Ergebnis der Funktion. Ist der Inhalt des `print`-Befehls weder eine mathematische Operation, noch eine Variable, Zeichenkette oder eine Funktion, erscheint eine Fehlermeldung.

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

# Kapitel 5

## Variablen: unverzichtbar für die Programmierung mit Python

In Kapitel 3.2. verwendeten wir im Rahmen der interaktiven Anwendung des Python-Prompts erstmalig Variablen, ohne jedoch deren genaue Funktionsweise zu verstehen. Da Variablen einen wesentlichen Aspekt des Programmierens ausmachen, wollen wir uns in diesem Kapitel dem Thema Variablen widmen.

Variablen werden durch einen *Namen* (auch Bezeichner genannt) angegeben und verweisen auf einen bestimmten Wert, dem sie durch ein Gleichheitszeichen – dem sogenannten *Zuweisungsoperator* – zugewiesen werden. Durch Variablen lassen sich somit Werte erfassen, auf die man später im Programm mittels des Variablenamens zugreifen kann. So lassen sich mithilfe von Variablen etwa feste Rechenvorschriften für größere Berechnungen definieren, spezifische Ergebnisse speichern und anschließend für weitere Prozesse verwenden.

5

### 5.1 Die Aufgabe von Variablen

Sie können sich den Arbeitsspeicher in einem Computer wie eine große Kommode vorstellen. Die Kommode besteht aus vielen Schubladen, in die Sie unterschiedliche Objekte legen können – bildlich gesprochen sind dies die Werte, die das Programm abspeichern wird. Da die Gegenstände in der Kommode verschieden groß sein können, ist die Kommode zudem mit Schubladen unterschiedlicher Größen ausgestattet, die jeweils für Objekte einer bestimmten Größe vorgesehen sind.

Möchten Sie später einen bestimmten Gegenstand aus der Kommode entnehmen, so müssen Sie hierfür wissen, in welcher Schublade sich dieser befindet. Je größer die Kommode ist und je mehr Gegenstände Sie darin abgelegt haben, umso schwieriger wird es sein, sich für jeden Gegenstand an dessen genauen Aufbewahrungsort zu erinnern. Um nicht lange nach dem gewünschten Objekt suchen zu müssen, ist es daher eine gute Idee, die einzelnen Schubladen zu beschriften.

Variablen sind ähnlich organisiert. Der Arbeitsspeicher eines Computers bietet Platz für viele Tausend Variablen – jede der verwendeten Variablen muss dabei einen eindeutigen, festen Namen erhalten. Der zugehörige Speicherort wird vom Programm bei dessen Ausführung automatisch hinterlegt, sodass die benötigten Informationen stets sofort aufgerufen werden können. Mithilfe des Variablenamens lässt sich der entsprechende Speicherort sodann identifizieren. Dafür ist es wichtig, dass der Varia-

## 5 Variablen: unverzichtbar für die Programmierung mit Python

blenname eindeutig und nicht etwa doppelt vergeben wurde, da dies zu Fehlern im Programm führen kann.

Variablen können Speicherplatz unterschiedlicher Größe beanspruchen – so kann eine Variable etwa eine einstellige Zahl, ein einziges Wort oder einen ganzen Roman aufnehmen. Aus diesem Grund wird in vielen Programmiersprachen verlangt, dass die Variablentypen schon zu Beginn fest vorgegeben werden, damit das Programm den passenden Speicherplatz reservieren kann. Python funktioniert in diesem Punkt anders: Der Variablentyp wird hier je nach Bedarf während der Ausführung angegeben. Die Variablen sind also nicht von vornherein an einen speziellen Typ gebunden, sondern können unterschiedliche Größen annehmen. Dies nennt man *dynamische Typisierung*.

### 5.2 Variablen in Python verwenden

In Kapitel 3 machten wir bei unserer ersten Verwendung von Variablen im Python-Prompt folgende Eingabe: `x = 7`. Wir wiesen der Variablen `x` somit den Wert 7 zu. Auch Programme, die wir in eine eigene Datei schreiben, setzen Variablen nach demselben Schema ein.

Dabei wählt man zunächst einen Namen für die Variable. Der Name kann aus beliebigen Buchstaben, Zahlen sowie dem Unterstrich bestehen. Die einzige Einschränkung hierbei ist, dass am Anfang des Variablenamens keine Zahl stehen darf: `7_variable` ist somit kein gültiger Ausdruck für eine Variable, `variable_7` jedoch schon. Man wählt hier gerne sogenannte *sprechende Variablennamen* – dabei wird aus dem Namen bereits die Bedeutung der Variablen ersichtlich, sodass man beim Lesen des Codes leichter verstehen kann, was die Variable bezeichnet.

Ein weiterer wichtiger Aspekt ist die Groß- und Kleinschreibung. Die gängige Konvention ist hier – wie auch bei vielen anderen Namensgebungen in Python – ausschließlich kleine Buchstaben als Variablennamen zu verwenden. In Python stellen die drei Ausdrücke `variable`, `Variable` und `VARIABLE` außerdem drei verschiedene Bezeichnungen dar, die als separate Variablen abgespeichert werden. Ein neues Feature ab Python-Version 3 ist zudem die Unterstützung von Unicode-Zeichen. Dazu gehören etwa Umlaute, mathematische Symbole, nichtlateinische Buchstaben verschiedener Sprachen sowie eine große Anzahl weiterer Sonderzeichen. Die Verwendung solcher Zeichen in Variablennamen ist jedoch eher selten und sollte wohlüberlegt sein, insbesondere dann, wenn weitere Personen und Rechner am Programmierprozess beteiligt sind.

## 5.2 Variablen in Python verwenden

Nachdem Sie einen Variablenamen gewählt und eingegeben haben, muss im Anschluss ein Gleichheitszeichen (=) stehen. Das Gleichheitszeichen funktioniert in diesem Fall als Zuweisungsoperator, der der Variablen einen spezifischen Wert zuweist. Im nächsten Schritt wird sodann dieser Wert eingegeben. Dieser kann aus einer ganzen oder einer Kommazahl, einem Buchstaben oder sogar Wörtern und Sätzen bestehen. Auch Wahrheitswerte (True und False) sind möglich. Buchstaben, Wörter und Sätze müssen stets in Anführungszeichen stehen, da Python diese sonst für einen anderen Variablen- oder Funktionsnamen hält. Ist kein solcher definiert, gibt das Programm eine Fehlermeldung aus.

Den Datentyp müssen Sie nicht händisch vorgeben. Python erkennt anhand der Zuweisung selbst, um welche Art von Information es sich dabei handelt und reserviert dafür einen passenden Platz im Speicher.

Erinnern wir uns erneut an die interaktive Ausführung von Python: Nachdem wir eine Variable samt deren Wert definiert hatten, war es ausreichend, den Variablenamen einzutippen und **Enter** zu drücken, um sich vom Interpreter den Wert der Variablen ausgeben zu lassen. Wenn Sie ein Programm in eine eigene Datei schreiben, können Sie den Wert der Variablen jedoch nicht auf diese Weise abfragen. Hier müssen Sie den Namen in einen print-Befehl setzen. Achten Sie darauf, hier keine Anführungszeichen zu verwenden, da Python den Inhalt des print-Befehls sonst nicht als Variable, sondern als Zeichenkette interpretiert und lediglich die Buchstaben in den Anführungszeichen anzeigt.

5

Ein zweizeiliges Programm, in dem zunächst die Variable x definiert und mit einem Wert versehen wird, der sodann abgefragt wird, würde demnach die folgende Form haben:

### *Codebeispiel #1*

```
1 x = 7
2 print(x)
```

Eine praktische Darstellungsoption ist darin gegeben, in der Ausgabe einen Text mit dem Wert der Variablen zu verbinden. Hierfür müssen Sie diese beiden Bestandteile in der Klammer bloß durch ein Komma trennen:

```
1 x = 8
2 print("Wert der Variablen x: ", x)
```

## 5 Variablen: unverzichtbar für die Programmierung mit Python

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top indicates 'Kapitel 5 [C:\Users\Sarah\PycharmProjects\Kapitel 5] - ..\beispiel\_1.py [Kapitel 5] - PyCharm'. Below the menu bar, the code editor window is open with the file 'beispiel\_1.py' containing the following code:

```
x = 7
print(x)

x = 8
print("Wert der Variablen x: ", x)
```

The 'Run' tab in the bottom left shows the command 'C:\Users\Sarah\PycharmProjects\Kapitel 5\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 5/beispiel\_1.py"'. The run output window displays the results:

```
7
Wert der Variablen x: 8
Process finished with exit code 0
```

**Abb. 5.1** Die Ausgabe der Variablenwerte

Wie Sie anhand des obigen Beispiels womöglich erkannt haben, sind Variablen innerhalb des Programms nicht an feste Werte gebunden. Ihr Wert kann sich im Laufe des Programmverlaufs beliebig ändern. Diese Änderung lässt sich entweder durch eine Neuzuweisung, eine Rechenaufgabe oder weitere Variablen realisieren:

### Codebeispiel #2

```
1 x = 7
2 print("Wert der Variablen x: ", x)
3 x = 11
4 print("Wert der Variablen x: ", x)
5 x = 5 * 5
6 print("Wert der Variablen x: ", x)
7 y = 19
8 x = y
9 print("Wert der Variablen x: ", x)
10 x = x + 4
11 print("Wert der Variablen x: ", x)
```

## 5.2 Variablen in Python verwenden

```

Kapitel 5 [C:\Users\Sarah\PycharmProjects\Kapitel 5] - ...\\beispiel_2.py [Kapitel 5] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 5 beispiel_2.py
Project: Kapitel 5 C:\Users\Sarah\PycharmProjects\Kapitel 5
  venv library root
    beispiel_1.py
    beispiel_2.py
    beispiel_3.py
    beispiel_4.py
    beispiel_5.py
    beispiel_6.py
    Kapitel 7.mn
Run: beispiel_2
  "C:\Users\Sarah\PycharmProjects\Kapitel 5\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 5/beispiel_2.py"
  Wert der Variablen x: 7
  Wert der Variablen x: 11
  Wert der Variablen x: 25
  Wert der Variablen x: 19
  Wert der Variablen x: 23
  Wert der Variablen x: 23
  Terminal Python Console Run TODO Event Log
  12:1 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 5) 7m

```

**Abb. 5.2** Die Variable `x` nimmt verschiedene Werte an

5

Wie bereits in 3.2. bemerkt, ist die letzte Zuweisung `x = x + 4` aus mathematischer Sicht betrachtet nicht korrekt, da der Wert der Variablen `x` nicht gleichzeitig derselbe Wert plus 4 sein kann. Das Gleichheitszeichen (`=`) ist in diesem Fall also nicht als Gleichheitszeichen im mathematischen Sinn zu verstehen. Vielmehr handelt es sich dabei um einen Zuweisungsoperator, der der Variablen auf der linken Seite den Wert auf der rechten Seite zuordnet. Derselbe Variablenname darf dabei auf beiden Seiten des Gleichheitszeichens vorkommen. Der Ausdruck `x = x + 4` bedeutet also, dass die Variable `x` unter der Neuzuweisung ihren bisherigen Wert plus vier erhalten soll.

Python verfügt neben dem Gleichheitszeichen über eine Reihe zusammengesetzter Zuweisungsoperatoren, mithilfe derer sich bestimmte Variablenrechnungen in reduzierter Form schreiben lassen:

Operator	Beispiel	Äquivalent zu
<code>=</code>	<code>x = 5</code>	<code>x = 5</code>
<code>+=</code>	<code>x += 5</code>	<code>x = x + 5</code>
<code>-=</code>	<code>x -= 5</code>	<code>x = x - 5</code>
<code>*=</code>	<code>x *= 5</code>	<code>x = x * 5</code>
<code>/=</code>	<code>x /= 5</code>	<code>x = x / 5</code>
<code>%=</code>	<code>x %= 5</code>	<code>x = x % 5</code>
<code>//=</code>	<code>x //= 5</code>	<code>x = x // 5</code>
<code>**=</code>	<code>x **= 5</code>	<code>x = x ** 5</code>

**Tab. 5.1** Zusammengesetzte Zuweisungsoperatoren

## 5 Variablen: unverzichtbar für die Programmierung mit Python

Die Division mit `/` ergibt immer eine Fließkommazahl als Ergebnis. Was das ist, werden wir in Abschnitt 5.4. erklären. Die Operanden `%` und `//` dürften Ihnen bislang unbekannt sein. Das `%`-Zeichen steht für die Modulo-Rechnung: Das Ergebnis ist der Rest der Division des linken Operanden durch den rechten. So ergibt beispielsweise der Ausdruck `5 % 2` als Ergebnis `1`. Das `//`-Zeichen bezeichnet die Abrundungsdivision: Nach der Division von zwei Zahlen wird auf diejenige ganze Zahl abgerundet, die dem Ergebnis auf der linken Seite des Zahlenstrahls am nächsten ist. So ergibt zum Beispiel `7 // 2` die Zahl `3`. Durch `**` wird die Potenz zweier Zahlen berechnet. Das Ergebnis des Potenzierens von `3**2` ist beispielsweise `9`.

### 5.3 Den Wert einer Variablen durch eine Nutzereingabe festlegen

Unsere bisherigen Programmbeispiele berechneten Werte, die wir ganz zu Anfang beim Erstellen des Programms vorgaben. Einer der großen Vorteile eines Computerprogramms besteht jedoch darin, dass es mit der Nutzerin oder dem Nutzer interagieren und die Berechnungen neu an deren Anforderungen anpassen kann.

Nun wollen wir untersuchen, wie sich Anwender interaktiv in das Programm einbeziehen lassen. Hierzu werden wir ein Programm schreiben, das den Nutzer oder die Nutzerin dazu auffordert, eine beliebige Zahl einzugeben. Daraufhin wird der doppelte Wert dieser Zahl berechnet und auf dem Bildschirm ausgegeben.

Wie zuvor werden wir dazu in der Eingabeaufforderung einen kurzen Text eingeben. Folgende Zeile beschreibt dabei die Eingabe der Zahl:

```
1 inhalt = input()
```

Hier wird zunächst die Variable `inhalt` definiert, die als Wert die Funktion `input()` enthält. Diese Funktion hat die Aufgabe, eine Eingabe des Anwenders zurückzugeben. Im Anschluss wollen wir den Wert verdoppeln um ihn danach auszugeben:

#### Codebeispiel #3

```
1 print("Geben Sie bitte eine Zahl ein: ")
2 inhalt = input()
3 inhalt = inhalt*2
4 print("Doppelter Wert: ", inhalt)
```

### 5.3 Den Wert einer Variablen durch eine Nutzereingabe festlegen

The screenshot shows a PyCharm interface with the following details:

- Project:** Kapitel 5 [C:\Users\Sarah\PycharmProjects\Kapitel 5] - beispiel\_3.py [Kapitel 5] - PyCharm
- Code Editor:** Content of 'beispiel\_3.py':
 

```
1 print("Geben Sie bitte eine Zahl ein: ")
2 inhalt = input()
3 inhalt = inhalt*2
4 print("Doppelter Wert: ", inhalt)
```
- Run Tab:** Shows the command run: "C:/Users/Sarah/PycharmProjects/Kapitel 5/venv/Scripts/python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 5/beispiel\_3.py". The output window shows:
 

```
Geben Sie bitte eine Zahl ein:
7
Doppelter Wert: 77
```
- Bottom Status Bar:** Shows the status: Process finished with exit code 0.

Abb. 5.3 Die vermeintliche Verdopplung des Werts

Offensichtlich macht diese Ausgabe keinen Sinn, da der doppelte Wert von 7 die Zahl 14 und nicht 77 ergeben sollte. Das fehlerhafte Ergebnis ist darin begründet, dass der `input`-Befehl alle Werte als Zeichen anstatt als Zahlen liest und einen dementsprechenden Output liefert. Verdoppelt man ein Zeichen, wird dieses daher einfach zwei Mal hintereinander ausgegeben: Die Verdopplung des Zeichens x ist xx, und 7 verdoppelt sich zu 77.

5

Damit die `input`-Funktion ein sinnvolles Ergebnis produziert, muss man den Befehl `eval()` verwenden, mithilfe dessen das Ergebnis automatisch nicht als Zeichen, sondern als Zahl abgespeichert wird. Der `input`-Befehl steht dabei innerhalb der Klammer des `eval`-Befehls. Die notwendige Abänderung sieht demnach wie folgt aus:

#### Codebeispiel #4

```
1 print("Geben Sie bitte eine Zahl ein: ")
2 inhalt = eval(input())
3 inhalt = inhalt*2
4 print("Doppelter Wert: ", inhalt)
```

## 5 Variablen: unverzichtbar für die Programmierung mit Python

```

print("Geben Sie bitte eine Zahl ein:")
inhalt = eval(input())
inhalt = inhalt*2
print("Doppelter Wert: ", inhalt)

print("Geben Sie bitte eine Zahl ein:")
inhalt = eval(input())
inhalt = inhalt*2

```

C:\Users\Sarah\PycharmProjects\Kapitel 5\venv\Scripts\python.exe "C:/Users/Sarah/PycharmProjects/Kapitel 5/beispiel\_4.py"  
 Geben Sie bitte eine Zahl ein:  
 7  
 Doppelter Wert: 14  
 Geben Sie bitte eine Zahl ein:  
 X  
 Traceback (most recent call last):  
 File "C:/Users/Sarah/PycharmProjects/Kapitel 5/beispiel\_4.py", line 7, in <module>  
 inhalt = eval(input())  
 File "<string>", line 1, in <module>  
 NameError: name 'X' is not defined

**Abb. 5.4** Die richtige Verdopplung und die Fehlermeldung bei Eingabe eines Buchstabens

**Achtung:** Bei der Verwendung des `eval`-Befehls ist Vorsicht geboten. Allgemein ist die Aufgabe dieses Befehls, beliebige Strings als separaten Python-Code auszuwerten. Die Gefahr hierbei ist, dass bestimmter Quellcode unerwünschte Auswirkungen auf Ihr gesamtes System haben könnte – bis hin zur Löschung all Ihrer Daten. Verwenden Sie diesen Befehl also stets mit Vorsicht und geben Sie in die Klammer des `eval`-Befehls nur Inhalte ein, über deren Auswirkungen Sie genau Bescheid wissen.

Eine bevorzugte Alternative zum `eval`-Befehl ist, die Eingabe direkt in einen anderen Datentyp zu überführen. Wie Sie dies bewerkstelligen, erklären wir Ihnen in Abschnitt 5.5. Bei dieser letzten Methode müssen Sie den Datentyp gleich von Anfang an definieren, wodurch er nicht mehr dynamisch ist. Das bedeutet, dass eingegebene Werte, die einen anderen Datentyp verwenden, zu einer Fehlermeldung führen.

Die Verwendung von `eval` ist hier sehr flexibel und für viele verschiedene Fälle geeignet. Die Funktion erkennt selbstständig, welche Eingabe die passende ist. Lediglich bei Buchstaben, Wörtern und anderen Zeichenketten ist Vorsicht geboten. Damit diese richtig abgespeichert werden, müssen Sie sie in Anführungszeichen setzen, sonst erscheint eine Fehlermeldung (Abbildung 5.4.).

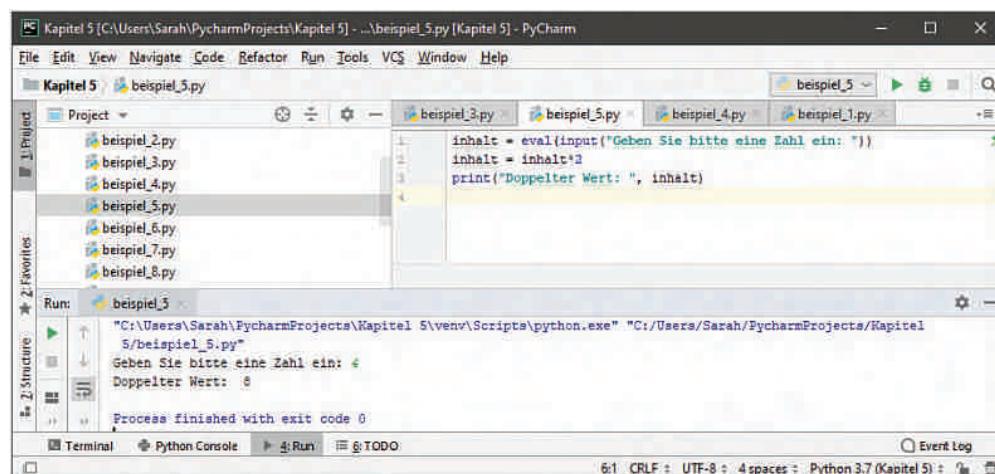
Wie Sie womöglich bemerkt haben, verlangt der `input`-Befehl eine leere Klammer. Das liegt daran, dass es sich hierbei um eine Funktion handelt, der durch die Eingabe des Nutzers bzw. der Nutzerin ein Wert übergeben werden kann. Im obigen Beispiel

## 5.4 Dynamische Typisierung: viele Freiheiten bei der Nutzung von Variablen

wurden Sie durch den separaten `print`-Befehl dazu aufgefordert, eine Zahl einzugeben. Man kann diese Eingabeaufforderung jedoch auch direkt in die Klammer der `eval`-Funktion einfügen, was einerseits den Code reduziert und andererseits eine sinnvollere Formulierung darstellt. Die Eingabe erfolgt nun in derselben Zeile wie die Eingabeaufforderung:

### Codebeispiel #5

```
1 inhalt = eval(input("Geben Sie bitte eine Zahl ein: "))
2 inhalt = inhalt*2
3 print("Doppelter Wert: ", inhalt)
```



5

Abb. 5.5 Das Programm mit der Eingabeaufforderung im `input`-Befehl

## 5.4 Dynamische Typisierung: viele Freiheiten bei der Nutzung von Variablen

Wie wir bereits gesehen haben, verzichtet Python auf die in vielen anderen Programmiersprachen obligatorische Angabe des Variablentyps. Python zeichnet sich hier durch die sogenannte Eigenschaft der *dynamischen Typisierung* aus. Das bedeutet, dass der Variablentyp und damit auch der benötigte Speicherplatz in Python erst während der Ausführung des Codes festgelegt werden, und dass sich der Variablentyp auch noch nachträglich innerhalb des Programms ändern lässt.

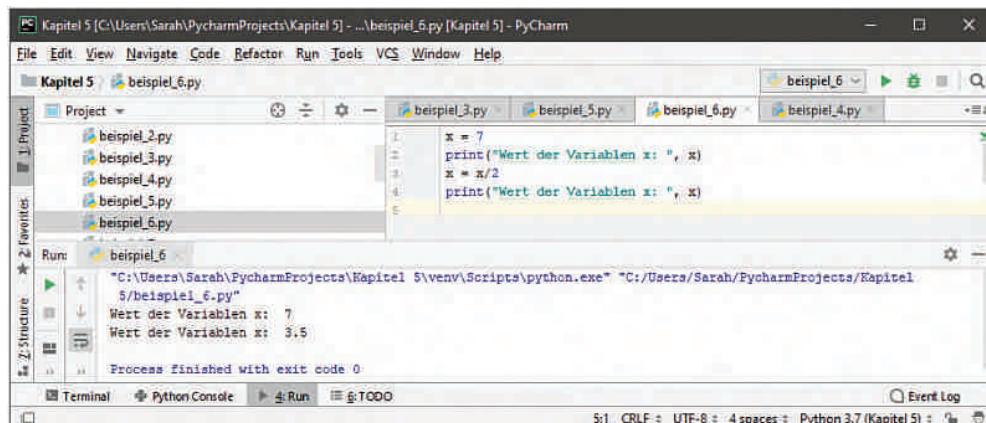
Beispiele für unterschiedliche Variablentypen sind etwa ganze Zahlen und Fließkommazahlen. Letztere (auch *Gleitkommazahlen* genannt, engl. *floating-point numbers*) enthalten per Definition einen Dezimalanteil. So handelt es sich etwa bei den Zahlen 5.77 oder  $1.3 \cdot 10^2$  oder auch bei 1.0 um Fließkommazahlen. Beachten Sie hier, dass das Dezimaltrennzeichen in Python gemäß der englischen Schreibweise stets ein Punkt (.) und kein Komma wie im Deutschen ist.

## 5 Variablen: unverzichtbar für die Programmierung mit Python

Computerprogramme machen einen Unterschied zwischen ganzen Zahlen und Fließkommazahlen, da sich der benötigte Speicherplatz für diese beiden Zahlentypen stark unterscheidet. Programmiersprachen mit statischer Typisierung können Variablenarten im Nachhinein nicht mehr ändern. Eine zunächst als ganze Zahl definierte Variable kann daher nachträglich keine Nachkommastellen erhalten. In Python ist dies jedoch möglich:

### Codebeispiel #6

```
1 x = 7
2 print("Wert der Variablen x: ", x)
3 x = x/2
4 print("Wert der Variablen x: ", x)
```



**Abb. 5.6** Die Variable verändert ihren Typ

Man könnte nun vermuten, dass die Zahl 7 von Anfang an in Wahrheit eine Fließkommazahl war, da die Zahlen 7 und 7.0 eigentlich denselben Wert haben. Das ist jedoch nicht der Fall. Betrachten Sie hierzu folgendes Beispiel:

### Codebeispiel #7

```
1 x = 7
2 print("Wert der Variablen x: ", x)
3 x = x + 0.0
4 print("Wert der Variablen x: ", x)
```

## 5.4 Dynamische Typisierung: viele Freiheiten bei der Nutzung von Variablen

```

Kapitel 5 [C:\Users\Sarah\PycharmProjects\Kapitel 5] - ..\beispiel_7.py [Kapitel 5] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 5 beispiel_7.py
Project beispiel_5.py beispiel_6.py beispiel_7.py beispiel_8.py beispiel_9.py
Run 5/beispiel_7.py
Wert der Variablen x: 7
Wert der Variablen x: 7.0
Process finished with exit code 0

```

**Abb. 5.7** Die Ausgabe als Ganzzahl und als Fließkommazahl

Hier wird der Wert 0.0 zur Variablen x addiert. Das ändert aus mathematischer Sicht zwar nicht ihren Wert, die Ausgabe jedoch ändert sich: Die Nachkommastelle, in diesem Fall eine Null, wird nun angezeigt. Das ist darin begründet, dass Rechenoperationen mit Fließkommazahlen in Python ebenfalls wieder Fließkommazahlen als Ergebnis liefern. Testen Sie dies einmal durch die Eingabe von  $4 * 1.5$  – die Rechnung ergibt den Wert 6.0. Python wandelt auch hier den Variablentyp automatisch um, obwohl das Ergebnis streng genommen eine ganze Zahl ist.

5

Doch nicht nur ganze Zahlen können in Fließkommazahlen umgewandelt werden und umgekehrt. Auch Variablen, die zuvor einen Zahlenwert abgespeichert hatten, können einer Zeichenkette (engl. *string*) zugewiesen werden. Python bewältigt dabei automatisch alle nötigen Änderungen:

### Codebeispiel #8

```

1 x = 7
2 print("Wert der Variablen x: ", x)
3 x = "Guten Tag"
4 print("Wert der Variablen x: ", x)

```

## 5 Variablen: unverzichtbar für die Programmierung mit Python

```

Kapitel 5 [C:\Users\Sarah\PycharmProjects\Kapitel 5] - ..\beispiel_8.py [Kapitel 5] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 5 beispiel_8.py
Project beispiel_3.py beispiel_5.py beispiel_6.py beispiel_7.py beispiel_8.py beispiel_9.py
1 x = 7
2 print("Wert der Variablen x: ", x)
3 x = "Guten Tag"
4 print("Wert der Variablen x: ", x)
5
Run beispiel_8
5/beispiel_8.py"
Wert der Variablen x: 7
Wert der Variablen x: Guten Tag
Process finished with exit code 0
Terminal Python Console Run TODO Event Log
5:1 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 5):

```

Abb. 5.8 Dieselbe Variable kann Zahlen und Zeichenketten aufnehmen

### 5.5 Datentypen sind auch in Python von Bedeutung

Wie wir gesehen haben, macht die dynamische Typisierung Programmervorgänge um einiges einfacher, da die Datentypen dabei nicht berücksichtigt werden müssen. Es gibt hier allerdings auch einige Ausnahmen. Im Codebeispiel aus Kapitel 5.3. erhielten wir etwa ein fehlerhaftes Ergebnis bei der Verdoppelung einer Zahl, da diese zunächst automatisch als Zeichenkette abgespeichert wurde. Mithilfe der `eval`-Funktion erreichten wir schließlich, dass das Programm die Nutzereingabe als Zahlenwert erkannte und die Rechnung wie gewünscht ausführte.

Der Datentyp lässt sich jedoch auch direkt umwandeln. Hierfür müssen Sie im Programmcode zunächst den gewünschten Datentyp explizit angeben und die bezeichnende Variable dahinter in Klammern hinzufügen. In der folgenden Tabelle finden Sie die häufigsten Datentypen:

Python-Abkürzung	Englische Bezeichnung	Datentyp
int	integer	ganze Zahlen
float	floating-point number	Fließkommazahlen
str	string	String bzw. Zeichenkette (Buchstaben, Wörter, Sätze oder Texte)
bool	boolean	Boolesche Variablen/ Wahrheitswerte (True oder False)

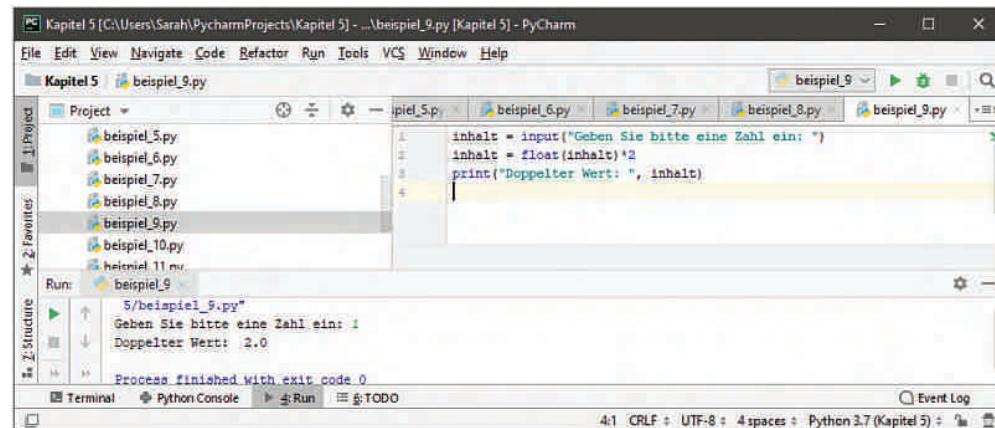
Tab. 5.2 Die häufigsten Datentypen in Python

## 5.5 Datentypen sind auch in Python von Bedeutung

Womöglich können Sie sich anhand dieser einfachen Tabelle bereits vorstellen, wie man den String in unserem Zahlen-Verdoppelungsprogramm ohne Verwendung der eval-Funktion in eine Zahl umwandeln kann:

### Codebeispiel #9

```
1 inhalt = input("Geben Sie bitte eine Zahl ein: ")
2 inhalt = float(inhalt)*2
3 print("Doppelter Wert: ", inhalt)
```



5

**Abb. 5.9** Die direkte Umwandlung des Datentyps

Der Output ist hier also ein Datentyp vom Typ float, was eine Fließkommazahl darstellt. Wenn man annehmen kann, dass die Anwendereingabe stets eine ganze Zahl sein wird, lässt sich die zweite Zeile hier genauso gut durch `inhalt = int(inhalt)*2` ersetzen. Versuchen Sie einmal aus, was passiert, wenn Sie eine float-Variable – etwa 1.5 – mittels des `int`-Befehls in eine ganze Zahl umwandeln. Ganz richtig! Python entfernt hier einfach die Nachkommastellen und gibt die ganze Zahl vor dem Komma aus: 1.

In Python gibt es zudem einige Funktionen, die nur mit einem bestimmten Datentyp arbeiten können. Eine dieser Funktionen ist beispielsweise `upper()`. Mit diesem Befehl werden alle klein geschriebenen Buchstaben innerhalb einer Zeichenkette in Großbuchstaben umgewandelt:

### Codebeispiel #10

```
1 text = "hallo welt!"
2 print("Wert der Variablen text vor der Änderung: ", text)
3 text = text.upper()
4 print("Wert der Variablen text nach der Änderung: ", text)
```

## 5 Variablen: unverzichtbar für die Programmierung mit Python

Der `upper`-Befehl funktioniert nur bei Zeichenketten, also Variablen des Typs `str`. Versuchen Sie einmal, als `text`-Variable stattdessen eine Zahl einzugeben – sowohl mit als auch ohne Anführungszeichen.

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top indicates 'Kapitel 5' and 'beispiel\_10.py'. The code editor window contains the following Python script:

```

text = "hallo welt!"
print("Wert der Variablen text vor der Änderung: ", text)
text = text.upper()
print("Wert der Variablen text nach der Änderung: ", text)

```

The 'Run' tab in the bottom navigation bar is selected, showing the command: "C:\Users\Sarah\PycharmProjects\Kapitel 5\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 5/beispiel\_10.py". The run output pane displays the results of the print statements:

```

Wert der Variablen text vor der Änderung: hallo welt!
Wert der Variablen text nach der Änderung: HALLO WELT!

```

**Abb. 5.10** Die Verwendung des `upper`-Befehls mit einer Zeichenkette

Python behandelt eine mit Anführungszeichen angegebene Zahl ebenfalls als `str`-Variable, sodass der `upper`-Befehl auch in diesem Fall ausgeführt werden kann, ohne dass es eine sichtbare Veränderung gibt. Wenn Sie die Zahl jedoch ohne Anführungszeichen eingeben, wird diese als `int`- oder `float`-Variable abgespeichert, was zu einem Fehler führt, da die `upper`-Funktion nur für String-Variablen zulässig ist. Eine vom Schema her ähnliche Funktion ist übrigens `lower()`. Probieren Sie sie mit verschiedenen Zeichenketten aus und sehen Sie, was passiert.

In Python gibt es also dennoch Funktionen, bei denen der Datentyp nicht außer Acht gelassen werden darf. Sie sollten sich daher beim Programmieren stets bewusst sein, mit welchen Datentypen Sie hantieren und ob alle gewünschten Befehle und Operationen mit diesen vereinbar sind.

Um den Datentyp einer Variablen zu überprüfen, können Sie den Befehl `type()` verwenden. Wie Sie sehen, lässt sich mit diesem praktischen Befehl der Datentyp der entsprechenden Variablen anzeigen:

### Codebeispiel #11

```

1 var1 = 7
2 print("Datentyp Variable var1: ", type(var1))
3 var2 = 3.141592653
4 print("Datentyp Variable var2: ", type(var2))
5 var3 = True
6 print("Datentyp Variable var3: ", type(var3))
7 var4 = "Hallo"

```

## 5.5 Datentypen sind auch in Python von Bedeutung

```

8 print("Datentyp Variable var4: ", type(var4))
9 var5 = (2, 4)
10 print("Datentyp Variable var5: ", type(var5))

```

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists several files: Beispiel\_6.py, Beispiel\_7.py, Beispiel\_8.py, Beispiel\_9.py, Beispiel\_10.py, Beispiel\_11.py, Übung\_1.py, and Übung\_2.py. Below this is the main editor window containing the code for Beispiel\_11.py. The code defines variables var1 through var5 and prints their types. The terminal below the editor shows the execution of the script and the resulting output, which correctly identifies each variable's type. The status bar at the bottom indicates the current file is 'Python 3.7 (Kapitel 5)'.

```

print("Datentyp Variable var1: ", type(var1))
var2 = 3.141592653
print("Datentyp Variable var2: ", type(var2))
var3 = True
print("Datentyp Variable var3: ", type(var3))
var4 = "Hallo"
print("Datentyp Variable var4: ", type(var4))
var5 = (2, 4)
print("Datentyp Variable var5: ", type(var5))

```

**Abb. 5.11** Die Ausgabe der verschiedenen Datentypen

Mittlerweile sollten Sie eine Idee davon bekommen haben, worum es sich bei den ersten vier Datentypen handelt. Was der tuple-Datentyp von var5 bezeichnet und welche weiteren Datentypen bzw. Datenstrukturen es gibt, werden wir im nächsten Kapitel untersuchen.

## 5 Variablen: unverzichtbar für die Programmierung mit Python

### 5.6 Übung: Mit Variablen arbeiten

#### Übungsaufgaben

1. Verfassen Sie ein Programm, das den Anwender oder die Anwenderin dazu auffordert, zwei beliebige Zahlen einzugeben. Verwenden Sie hierfür nicht die eval-Funktion, sondern die direkte Umwandlung in einen float-Datentyp. Speichern Sie die beiden Zahlen in zwei unterschiedlichen Variablen ab. Das Programm soll danach die beiden Werte addieren und ausgeben.
2. Schreiben Sie ein Programm, das den Anwender oder die Anwenderin dazu auffordert, einen beliebigen Inhalt einzugeben. Speichern Sie den Wert in einer Variablen und geben Sie anschließend ihren Datentyp aus.

## 5.6 Übung: Mit Variablen arbeiten

## Lösungen

1.

```
1 x = float(input("Geben Sie die erste Zahl ein: "))
2 y = float(input("Geben Sie die zweite Zahl ein: "))
3 print("Summe: ", x+y)
```

The screenshot shows the PyCharm interface with the project 'Kapitel 5' open. In the 'Run' tool window, the script 'Übung\_1.py' is selected. The 'Run' dropdown menu is open, showing the command: "C:\Users\Sarah\PycharmProjects\Kapitel 5\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 5/Übung\_1.py". The run log displays the following output:

```
Geben Sie die erste Zahl ein: 2
Geben Sie die zweite Zahl ein: 3.5
Summe: 5.5
```

The status bar at the bottom right indicates: 7:1 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 5) : 2m.

5

Abb. 5.12 Die Berechnung der Summe zweier eingegebener Zahlen

2.

```
1 inhalt = eval(input("Geben Sie einen Wert ein: "))
2 print("Datentyp: ", type(inhalt))
```

The screenshot shows the PyCharm interface with the project 'Kapitel 5' open. In the 'Run' tool window, the script 'Übung\_2.py' is selected. The 'Run' dropdown menu is open, showing the command: "C:\Users\Sarah\PycharmProjects\Kapitel 5\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 5/Übung\_2.py". The run log displays the following output for different inputs:

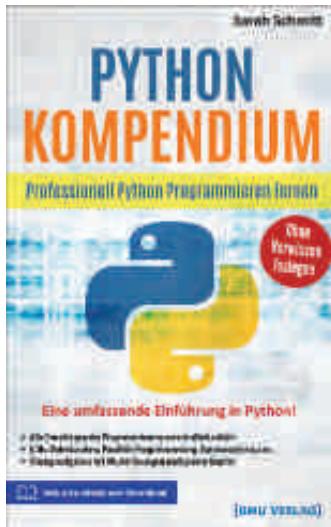
```
Geben Sie einen Wert ein: 1
Datentyp: <class 'int'>
Geben Sie einen Wert ein: 1.01
Datentyp: <class 'float'>
Geben Sie einen Wert ein: True
Datentyp: <class 'bool'>
Geben Sie einen Wert ein: "xy"
Datentyp: <class 'str'>
```

The status bar at the bottom right indicates: 12:1 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 5) : 2m.

Abb. 5.13 Die Anzeige unterschiedlicher Datentypen

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

# Kapitel 6

## Datenstrukturen in Python

Im vorigen Kapitel haben Sie die vier elementaren *Datentypen* `int`, `float`, `bool` und `str` kennen gelernt, in denen unterschiedliche Wertebereiche für Daten abgespeichert werden können, auf denen jeweils bestimmte Operationen möglich sind. Eine weitere Möglichkeit, Daten zu organisieren, sind sogenannte *Datenstrukturen*. Obwohl die beiden Begriffe *Datentyp* und *Datenstruktur* in der Literatur oftmals nicht klar voneinander abgegrenzt sind, kann man im Allgemeinen sagen, dass es sich bei Datenstrukturen um spezifische, komplexere Konstruktionen handelt, in denen gleiche oder verschiedene Datentypen zusammengefasst werden können. Datenstrukturen können zudem weiter unterteilt werden. Wir wollen uns nun mit Datenstrukturen befassen und untersuchen, wie sich diese vorteilhaft für unsere bisherigen Beispiele einsetzen lassen.

### 6.1 Datenstrukturen: praktische Methoden zur Datenerfassung

6

In den Übungsaufgaben zu Kapitel 4 hatten wir das Beispiel eines Literatur-Verwaltungsprogramms einer Bibliothek betrachtet. Die Hauptaufgabe eines solchen Programms besteht offensichtlich darin, die Informationen zu den einzelnen Werken der Bibliothek zu erfassen und abzuspeichern, damit sie später leicht abrufbar sind. Die wesentlichen Werkinformationen würden dann beispielsweise Titel, Autor oder Autorin, Erscheinungsjahr und Verlag umfassen. Obwohl ein reales Programm vermutlich mehr Details beinhalten würde, arbeiten wir in diesem Beispiel der Übersichtlichkeit halber nur mit diesen vier Angaben.

Um die unterschiedlichen Informationen erfassen zu können, ist es nun denkbar, nach der Vorgehensweise des vorigen Kapitels für jeden Wert jeweils eine eigene Variable anzugeben. Die Angaben für ein bestimmtes Werk könnten dann beispielsweise folgendermaßen aussehen:

```

1 titel = "Kritik der reinen Vernunft"
2 autor_in = "Kant, Immanuel"
3 erscheinungsjahr = 1998
4 verlag = "Akademie-Verlag Berlin"
```

Diese Strategie würde genügen, wenn man im Rahmen eines vereinfachten Programms lediglich die Werte für ein einziges Werk erfassen will. Bibliotheken verfügen meistens jedoch über Tausende von Werken. All diese Werke anhand des oben gezeigten Schemas abzuspeichern, wäre mit einem erheblichen Zeitaufwand verbunden. So wären bereits für drei Werke die folgenden Angaben nötig:

## 6 Datenstrukturen in Python

```
1 titell = "Kritik der reinen Vernunft"
2 autor_in1 = "Kant, Immanuel"
3 erscheinungsjahr1 = 1998
4 verlag1 = "Akademie-Verlag Berlin"
5 titel2 = "Verbrechen und Strafe"
6 autor_in2 = "Dostojewskij, Fjodor M."
7 erscheinungsjahr2 = 1996
8 verlag2 = "FISCHER Taschenbuch"
9 titel3 = "Meine geniale Freundin"
10 autor_in3 = "Ferrante, Elena"
11 erscheinungsjahr3 = 2018
12 verlag3 = "Suhrkamp Verlag"
```

Wie Sie sehen, würde diese Form der Datenerfassung bereits bei drei Werken sehr viel Arbeit bedeuten. Zudem wäre es schwierig, hinterher auf die Daten zuzugreifen, da man den Namen einer Variable bereits kennen müsste, um ihren Wert abfragen zu können. Der eigentliche Vorteil der Automatisierung, den man von einem Verwaltungsprogramm erwartet, wäre hier also nur sehr bedingt gegeben.

Wenn die Bibliotheksangestellten die Werkinformationen auf Papier festhalten wollten, würden sie dafür wahrscheinlich ein Blatt pro Werk verwenden, auf dem sie die entsprechenden Daten in einer vorgefertigten Tabelle oder Liste eintragen würden. Diese Ordnungsstruktur würde auch weitere Kategorisierungen der Werke (z. B. in Sachgebiete) vereinfachen, da die einzelnen Blätter wiederum in verschiedenen Ordern zusammengefasst werden könnten.

In Python gibt es ähnliche Möglichkeiten, Daten zu strukturieren, mithilfe derer sich auch größere Datensätze übersichtlich erfassen und einfach handhaben lassen. So mit bleibt Ihnen die Mühe erspart, die Daten wie oben dargestellt anzulegen. Python verfügt sogar über eine vielfältige Auswahl an Datenstrukturen, die sich beim Programmieren je nach Anforderung flexibel einsetzen lassen.

### 6.2 Listen: mehrere Informationen zusammenfassen

Sie kennen analoge Listen aus dem Alltag: das Menü Ihres Lieblingsrestaurants, das Telefonbuch Ihres Wohnorts, die Top20-Musikcharts eines Radiosenders – all dies sind Listen, in denen Informationen in gesammelter Form erfasst werden. In der digitalen Welt der Python-Programme funktionieren Listen ganz ähnlich. Sie beinhalten unterschiedliche Informationen, die jeweils als getrennte Elemente, jedoch innerhalb eines Speicherortes abgelegt und aufgerufen werden können. Der einzige Unterschied hierbei ist, dass Listen in Python – genauso wie Variablen – einen bestimmten Namen benötigen, durch den die Daten referenziert werden.

Ein Listeneintrag für unser erstes Beispielwerk würde folgendermaßen aussehen:

## 6.2 Listen: mehrere Informationen zusammenfassen

### Codebeispiel #1

```
1 werk1 = ["Kritik der reinen Vernunft", "Kant, Immanuel", 1998, "Akademie-
2 Verlag Berlin"]
```

Wichtig ist hierbei, die einzelnen Listeneinträge in eckige Klammern zu setzen und durch Kommas voneinander zu trennen. Wie Sie womöglich bemerkt haben, findet sich in der obigen Liste neben den drei String-Elementen ebenfalls der Integer-Wert 1998. Im Gegensatz zu vielen anderen Programmiersprachen, die sogenannte Arrays für jeweils nur einen bestimmten Datentyp verwenden, können Listen in Python hingegen ganz unterschiedliche Datentypen wie Zeichenketten, ganze Zahlen, Fließkomma-Zahlen oder boolesche Variablen aufnehmen.

Wir wollen jetzt einige Operationen und Bearbeitungsmöglichkeiten von Listen testen. Geben Sie dazu zunächst die oben erstellte Liste in den Python-Interpreter ein. Nun können Sie nacheinander die folgenden Befehle ausprobieren:

```
1 werk1
```

Ähnlich wie bei Variablen wird die vollständige Liste samt aller Einträge angezeigt, wenn man ihren Namen eingibt.

6

Nun könnte es sein, dass Sie lediglich auf bestimmte Elemente innerhalb der Liste zugreifen möchten. Dafür müssen Sie hinter den Listennamen eckige Klammern setzen, in denen die gewünschten Index-Nummern stehen.

Geben Sie hierfür ein:

```
1 werk1[0]
2 werk1[2]
```

Um einzelne Elemente innerhalb der Liste anzuzeigen, müssen Sie die gewünschten Elemente also anhand ihrer Position in Klammern nach dem Listennamen angeben. Dabei ist zu beachten, dass die Listeneinträge stets mit dem Index 0 beginnen. Möchte man sich beispielsweise das dritte Element einer Liste anzeigen lassen, so müsste man hierfür den Index 2 wählen. In Klammerschreibweise wäre dies: `werk1[2]`.

Machen Sie nun die folgende Eingabe:

```
1 werk1[-1]
2 werk1[-2]
```

Wie Sie sehen, können Indizes auch negative Werte haben. In diesem Fall wird die Zählung innerhalb der Liste von hinten begonnen. Das letzte Feld der Liste hat den Index  $-1$ , das vorletzte  $-2$ , und so weiter. Der negative Index fängt also nicht mit  $-0$  an, wie man zunächst vermuten könnte. Die Eingabe  $-0$  wird von Python wie eine  $0$  interpretiert und bezeichnet damit das erste Element in der Liste.

## 6 Datenstrukturen in Python

Testen Sie nun die folgende Eingabe:

```
1 werk1[0:2]
2 werk1[1:3]
```

Um einen Teilbereich sequenziell aufeinanderfolgender Elemente innerhalb der Liste einzusehen, gibt man in der eckigen Klammer den Start- und End-Index getrennt durch einen Doppelpunkt an. Das Element, das den End-Index bezeichnet, wird dabei nicht mit ausgegeben. So gibt `werk1[0:2]` beispielsweise das erste und das zweite Element aus, obwohl der Index 2 in der Klammer das dritte Element bezeichnet. Möchte man etwa das zweite bis siebte Element einer Liste anzeigen lassen, so muss man den Index in der Klammer um 1 erhöhen: `[1:8]`.

Im Anschluss können Sie die folgenden Befehle eingeben:

```
1 werk1[:3]
2 werk1[2:]
3 werk1[:]
```

Wird kein Start- bzw. Endpunkt gesetzt, wird die Liste vom ersten bzw. bis zum letzten Element angezeigt. So würde man mit der Eingabe `[:2]` die ersten beiden Elemente und mit `[1:]` alle Elemente bis auf das erste Element ausgewiesen bekommen. Dementsprechend gibt `[:]` die vollständige Liste aus.

Testen Sie die Funktionsweise mit weiteren Zahlenwerten, um besser zu verstehen, wie die einzelnen Listenelemente abgefragt werden können.

```
Command Prompt - python
>>> werk1 = ["Kritik der reinen Vernunft", "Kant, Immanuel", 1998, "Akademie-Verlag Berlin"]
>>> werk1
['Kritik der reinen Vernunft', 'Kant, Immanuel', 1998, 'Akademie-Verlag Berlin']
>>> werk1[0]
'Kritik der reinen Vernunft'
>>> werk1[2]
1998
>>> werk1[-1]
'Akademie-Verlag Berlin'
>>> werk1[-2]
1998
>>> werk1[0:2]
['Kritik der reinen Vernunft', 'Kant, Immanuel']
>>> werk1[1:3]
['Kant, Immanuel', 1998]
>>> werk1[1:2]
['Kritik der reinen Vernunft', 'Kant, Immanuel', 1998]
>>> werk1[2:]
[1998, 'Akademie-Verlag Berlin']
>>> werk1[::]
['Kritik der reinen Vernunft', 'Kant, Immanuel', 1998, 'Akademie-Verlag Berlin']
```

**Abb. 6.1** Die Abfrage verschiedener Listenelemente

Listen können zunächst leer definiert und sodann gefüllt werden. Eine leere Liste namens `werk1` hätte somit die folgende Form:

```
1 werk1 = []
```

## 6.2 Listen: mehrere Informationen zusammenfassen

Das Löschen eines oder mehrerer Listenelemente folgt einem ähnlichen Schema und wird mithilfe des `del`-Befehls bewerkstelligt. Probieren Sie hierzu einmal die folgenden Eingaben aus. Um die geänderte `werk1`-Liste anzuzeigen, müssen Sie nach jeder Eingabe `werk1` eintippen.

### Codebeispiel #2

```
1 del werk1[0]
2 del werk1[-1]
3 del werk1[0:2]
4 del werk1
```

```
Command Prompt - python
>>> werk1 = ["Kritik der reinen Vernunft", "Kant, Immanuel", 1998, "Akademie-Verlag Berlin"]
>>> del werk1[0]
>>> werk1
['Kant, Immanuel', 1998, 'Akademie-Verlag Berlin']
>>> del werk1[-1]
>>> werk1
['Kant, Immanuel', 1998]
>>> del werk1[0:2]
>>> werk1
[]
>>> del werk1
>>> werk1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'werk1' is not defined
>>> -
```

6

**Abb. 6.2** Unterschiedliche Löschvorgänge für die Bearbeitung von Listen

Die Liste enthält nach den ersten drei Befehlen keine weiteren Einträge. Der letzte Befehl löscht die komplette Liste. Wenn Sie nun `werk1` eingeben, erhalten Sie eine Fehlermeldung.

Umgekehrt lassen sich Elemente direkt aus einer Liste entfernen, indem man nicht deren Indexnummer, sondern das konkrete Element angibt. Dies geschieht mithilfe des `remove`-Befehls. Wollen Sie beispielsweise das Erscheinungsjahr aus der `werk1`-Liste entfernen, so würde hierfür der Befehl `werk1.remove(1998)` genügen. Strings müssen hier wie sonst auch in Anführungszeichen stehen.

Der komplette Inhalt einer Liste lässt sich durch den Listennamen gefolgt von einem Punkt und dem Befehl `clear()` löschen. In unserem Fall wäre das also `werk1.clear()`. Die Liste bleibt hierbei definiert, Python liefert als Ergebnis eine leere Liste: `[]`.

Die Elemente innerhalb einer Liste lassen sich auch nachträglich ändern. Möchten Sie beispielsweise das Erscheinungsjahr einer Ausgabe ändern, so geht das ganz einfach durch den Befehl `werk1[2] = 2000`. Darüber hinaus ist es möglich, Listen zu addieren und zu multiplizieren, oder Teilbereiche daraus in eine neue Liste zu schreiben. Probieren Sie dazu einmal die folgenden Befehle im Python-Prompt aus:

## 6 Datenstrukturen in Python

### Codebeispiel #3

```

1 werk1 = ["Kritik der reinen Vernunft", "Kant, Immanuel", 1998, "Akademie-
2 Verlag Berlin"]
3 werk2 = ["Meine geniale Freundin", "Ferrante, Elena", 2018, "Suhrkamp Verlag"]
4 werk1 + werk2
5 werk1*3
6 werk1 + werk2*2
7 neueliste = werk1[1:3]
8 neueliste

```

```

C:\ Command Prompt - python
>>> werk1 = ["Kritik der reinen Vernunft", "Kant, Immanuel", 1998, "Akademie-Verlag Berlin"]
>>> werk2 = ["Meine geniale Freundin", "Ferrante, Elena", 2018, "Suhrkamp Verlag"]
>>> werk1 + werk2
['Kritik der reinen Vernunft', 'Kant, Immanuel', 1998, 'Akademie-Verlag Berlin', 'Meine geniale Freundin', 'Ferrante, Elena', 2018, 'Suhrkamp Verlag']
>>> werk1*3
['Kritik der reinen Vernunft', 'Kant, Immanuel', 1998, 'Akademie-Verlag Berlin', 'Kritik der reinen Vernunft', 'Kant, Immanuel', 1998, 'Akademie-Verlag Berlin', 'Kritik der reinen Vernunft', 'Kant, Immanuel', 1998, 'Akademie-Verlag Berlin']
>>> werk1 + werk2*2
['Kritik der reinen Vernunft', 'Kant, Immanuel', 1998, 'Akademie-Verlag Berlin', 'Meine geniale Freundin', 'Ferrante, Elena', 2018, 'Suhrkamp Verlag', 'Meine geniale Freundin', 'Ferrante, Elena', 2018, 'Suhrkamp Verlag']
>>> neueliste = werk1[1:3]
>>> neueliste
['Kant, Immanuel', 1998]
>>>

```

**Abb. 6.3** Additionen, Multiplikationen und Zuweisungen bei Listen

Hier geschieht Folgendes: Addiert man zwei Listen, werden deren Elemente hintereinander aneinandergefügt. Multipliziert man eine Liste, wird sie in der angegebenen Anzahl wiederholt. Zudem kann man einen Teilbereich einer Liste herausnehmen und ihn in einer neuen Liste mit einem anderen Namen anlegen, wie wir es hier mit `neueliste` getan haben.

Mithilfe der Additionsmethode lassen sich ebenfalls neue Elemente in die Liste aufnehmen. Möchte die Bibliothek beispielsweise nach dem Erfassen der Werkinformationen den zugehörigen Standort eines Buchs ebenfalls in der Liste angeben, muss hierfür keine neue Liste angelegt werden. Die folgenden Befehle ermöglichen dies:

```

1 werk1 = werk1 + ["CF 5015"]
2 werk1 = ["CF 5015"] + werk1

```

Wie Sie sehen, kann der Standort hier entweder am Ende oder am Anfang hinzugefügt werden.

Eine weitere Möglichkeit, Elemente am Ende einer Liste hinzuzufügen, ist die `append`-Funktion. Den Standort könnten wir somit genauso durch `werk1.append("CF 5015")` unserer Liste hinzufügen. Die `append`-Funktion unterscheidet sich von der vorigen Hinzufügungsmethode jedoch darin, dass diese Funktion jeweils nur ein einziges neues Listenelement aufnehmen kann, während bei der Hinzufügung durch Addition auch ganze Listen addiert werden können, sodass

## 6.2 Listen: mehrere Informationen zusammenfassen

etwa auch Befehle wie `werk1 = werk1 + ["2. Auflage", "CF 5015"]` möglich sind.

Listen können jedoch nicht nur viele einzelne Elemente enthalten, sondern auch wiederum ganze Listen als Unterlisten aufnehmen, wodurch Verschachtelungen von Listen entstehen. Somit lassen sich tabellenartige Strukturen erzeugen. Geben Sie dazu den folgenden Code im Python-Interpreter ein!

### Codebeispiel #4

```

1  werk1 = ["Kritik der reinen Vernunft", "Kant, Immanuel", 1998, "Akademie-
2  Verlag Berlin"]
3  werk2 = ["Meine geniale Freundin", "Ferrante, Elena", 2018, "Suhrkamp Verlag"]
4  werk3 = ["Verbrechen und Strafe", "Dostojewskij, Fjodor M.", 1996, "FISCHER
5  Taschenbuch"]
6  bestandsliste = [werk1, werk2, werk3]
7  bestandsliste
8  bestandsliste[0][0]
9  bestandsliste[1][3]
10 bestandsliste[0][1:3]
11 bestandsliste[1]
```

```

6 Command Prompt - python
>>> werk1 = ["Kritik der reinen Vernunft", "Kant, Immanuel", 1998, "Akademie-Verlag Berlin"]
>>> werk2 = ["Meine geniale Freundin", "Ferrante, Elena", 2018, "Suhrkamp Verlag"]
>>> werk3 = ["Verbrechen und Strafe", "Dostojewskij, Fjodor M.", 1996, "FISCHER Taschenbuch"]
>>> bestandsliste = [werk1, werk2, werk3]
>>> bestandsliste
[['Kritik der reinen Vernunft', 'Kant, Immanuel', 1998, 'Akademie-Verlag Berlin'], ['Meine geniale Freundin', 'Ferrante, Elena', 2018, 'Suhrkamp Verlag'], ['Verbrechen und Strafe', 'Dostojewskij, Fjodor M.', 1996, 'FISCHER Taschenbuch']]
>>> bestandsliste[0][0]
'Kritik der reinen Vernunft'
>>> bestandsliste[1][3]
'Suhrkamp Verlag'
>>> bestandsliste[0][1:3]
['Kant, Immanuel', 1998]
>>> bestandsliste[1]
['Meine geniale Freundin', 'Ferrante, Elena', 2018, 'Suhrkamp Verlag']
```

**Abb. 6.4** Die Verwendung zweidimensionaler Listen

Bei Listen, die aus zwei Dimensionen bestehen, müssen Sie zwei Indexzahlen angeben, um ein bestimmtes Element abzurufen. Die erste Zahl gibt dabei den Index der untergeordneten Liste an, die zweite die entsprechende Position innerhalb dieser gewählten Liste. Auch hier beginnt der Index bei 0. Dabei ist es auch in diesem Fall möglich, mithilfe des Doppelpunkts bestimmte Elementsequenzen auszuwählen. Geben Sie hingegen nur eine Indexzahl an, erhalten Sie als Ergebnis die komplette untergeordnete Liste zurück.

Listen können zudem weitere Datenstrukturen wie etwa Dictionaries oder Tupel enthalten, die in den folgenden Abschnitten vorgestellt werden.

## 6 Datenstrukturen in Python

### 6.3 Dictionaries: Zugriff über einen Schlüsselbegriff

Wie wir gesehen haben, erleichtert die Verwendung von Listen die Datenerfassung. Listen lassen sich schnell erstellen und benötigen verhältnismäßig wenig Ressourcen. Ein Nachteil bei Listen ist jedoch, dass man sich stets merken muss, welche Eigenschaften die einzelnen Elemente der Liste bezeichnen. In unseren Beispielen hatten wir es mit einer überschaubaren Anzahl von vier bis fünf Eigenschaften zu tun. Reale Programme könnten hingegen weitaus mehr Eigenschaften beinhalten, wodurch der Überblick leicht verloren gehen kann.

Hier sind *Dictionaries* (engl. für *Wörterbuch*) eine sinnvolle und beliebte Alternative. Mit Dictionaries wird der Zugriff auf einen bestimmten Wert nicht über eine Index-Nummer, sondern über einen bestimmten Schlüsselbegriff hergestellt. Möchte man beispielsweise den Standort von `werk1` in der Bibliothek wissen, so muss man hierfür nicht mit `werk1[4]` den genauen Index kennen, um darüber den Standort herauszufinden. Stattdessen kann man bei einem Dictionary durch die Eingabe von `werk1["Standort"]` direkt den entsprechenden Standort abfragen. Für unser Bibliotheksprogramm wären Dictionaries also sehr gut geeignet.

Obwohl die Anwendungsmöglichkeiten von Dictionaries über die eines tatsächlichen Wörterbuchs hinausgehen, können Sie sich das Grundprinzip von Dictionaries tatsächlich wie ein Wörterbuch vorstellen: Der Schlüssel (engl. *key*) ist dabei ein bestimmtes Wort in einer Sprache, dem jeweils eine Übersetzung bzw. ein Wert (engl. *value*) einer anderen Sprache zugeordnet wird.

Dictionaries verwenden geschweifte Klammern, in denen die Schlüssel-Wert-Paare stehen, die durch einen Doppelpunkt voneinander getrennt sind. Die verschiedenen Paare sind wiederum durch Kommas getrennt.

`werk1`, als Dictionary geschrieben, würde folgendermaßen aussehen:

#### Codebeispiel #5

```
1 werk1 = {"Titel": "Kritik der reinen Vernunft", "Autor_in": "Kant, Immanuel",
2           "Erscheinungsjahr": 1998, "Verlag": "Akademie-Verlag Berlin"}
```

Als Bezeichnung für den Schlüssel verwendet man meist Strings, die in Anführungszeichen stehen müssen. Es ist jedoch genauso gut möglich, wenn auch hinsichtlich der Verständlichkeit und Lesbarkeit des Codes weniger vorteilhaft, als Schlüsselbegriffe Zahlen oder Tupel (Kapitel 6.4.) statt Strings zu verwenden. Die Schlüsselbezeichnungen müssen zudem stets eindeutig sein.

Wie Sie sehen, ist die Erstellung eines Dictionaries etwas aufwändiger als die Erstellung einer Liste. Dictionaries benötigen zudem mehr Speicherplatz. Dafür sind sie

### 6.3 Dictionaries: Zugriff über einen Schlüsselbegriff

selbsterklärend und erhöhen die Lesbarkeit des Programmcodes, sodass sich die zusätzliche Mühe am Anfang meist lohnt.

Um die Funktionsweise von Dictionaries genauer untersuchen zu können, werden wir im interaktiven Python-Interpreter zunächst mit der obigen Zeile das werk1-Dictionary erstellen. Geben Sie anschließend die folgenden Befehle ein:

```
1 werk1
2 werk1["Titel"]
```

Wie bei Listen oder gewöhnlichen Variablen auch kann man sich den kompletten Inhalt eines Dictionaries anzeigen lassen, indem man den Namen des Dictionaries eingibt. Der Schlüssel für jedes Feld wird hier ebenfalls angezeigt. Möchten Sie den Inhalt eines bestimmten Felds sehen, so müssen Sie hierfür den Schlüssel des entsprechenden Felds in eine eckige Klammer hinter den Namen des Dictionaries setzen.

Machen Sie nun die folgende Eingabe:

```
1 werk1["Sprache"] = "Deutsch"
2 werk1
```

Sie können neue Felder im Dictionary hinzufügen, indem Sie ganz einfach den Dictionary-Namen und dahinter den Schlüssel des neuen Felds angeben, und diesem dann den gewünschten Wert zuweisen. Das neue Element wird automatisch hinten an das Dictionary angefügt. Eine weitere Hinzufügungsmöglichkeit ist die folgende:

```
1 werk1.update({"Seitenzahl": 680})
2 werk1
```

Beachten Sie hier die Verwendung der Klammern. Anstatt eines einzelnen Schlüssel-Wert-Paars lassen sich mit der update-Funktion auch mehrere Dictionaries zusammenfügen. Haben Sie beispielsweise in einem Dictionary namens werk1extra weitere Schlüssel-Wert-Paare mit zusätzlichen Werkinformationen zu werk1 definiert, die Sie als Werksinformationen zu werk1 hinzufügen möchten, so können Sie hierfür den Befehl werk1.update(werk1extra) verwenden. Achten Sie dabei stets darauf, dass das Zusammenführen von Dictionaries nach dieser Methode nur funktioniert, wenn die beiden Dictionaries keine gemeinsamen Schlüssel haben. Gibt es identische Schlüssel, so werden die Werte für werk1 lediglich durch die Werte der entsprechenden Schlüssel in werk1extra ersetzt. Probieren Sie es einmal selbst aus!

Machen Sie nun die nächste Eingabe:

```
1 werk1["Erscheinungsjahr"] = 2000
2 werk1
```

## 6 Datenstrukturen in Python

Mithilfe dieses Befehls können Sie ähnlich wie bei Listen vorhandene Einträge in der Liste ändern.

```
1 del werk1["Erscheinungsjahr"]
2 werk1
```

Mithilfe des `del`-Befehls gefolgt vom Dictionary-Namen und dem zu löschenen Schlüssel können Sie ähnlich wie bei Listen Felder aus dem Dictionary löschen. Sie können mit `del` selbstverständlich auch das gesamte Dictionary löschen.

```
1 list(werk1.keys())
2 sorted(werk1.keys())
```

Eine praktische Methode, um sich alle verwendeten Schlüssel oder Werte in einem Dictionary anzeigen zu lassen, ist die Verwendung des `list`-Befehls gefolgt von einer Klammer, die den Dictionary-Namen, dahinter einen Punkt und den Befehl `keys()` bzw. `values()` enthält. Um sich die Schlüssel in alphabetischer Reihenfolge anzeigen zu lassen, können Sie anstatt des `list()`-Befehls den Befehl `sorted` wählen.

```
1 werk1.clear()
2 werk1
```

Analog zum Löschen von Listen können Sie auch bei Dictionaries die `clear()`-Funktion verwenden. Das Ergebnis ist ein leeres Dictionary: {}.

```
PS C:\Users\Public\Documents\GitHub\Python\Kap06> python
>>> werk1 = {"Titel": "Kritik der reinen Vernunft", "Autor_in": "Kant, Immanuel", ...
...     "Erscheinungsjahr": 1998, "Verlag": "Akademie-Verlag Berlin"}
>>> werk1
{'Titel': 'Kritik der reinen Vernunft', 'Autor_in': 'Kant, Immanuel', 'Erscheinungsjahr': 1998, 'Verlag': 'Akademie-Verlag Berlin'}
>>> werk1["Titel"]
'Kritik der reinen Vernunft'
>>> werk1["Sprache"] = "Deutsch"
>>> werk1
{'Titel': 'Kritik der reinen Vernunft', 'Autor_in': 'Kant, Immanuel', 'Erscheinungsjahr': 1998, 'Verlag': 'Akademie-Verlag Berlin', 'Sprache': 'Deutsch'}
>>> werk1.update({'Seitenzahl': 680})
>>> werk1
{'Titel': 'Kritik der reinen Vernunft', 'Autor_in': 'Kant, Immanuel', 'Erscheinungsjahr': 1998, 'Verlag': 'Akademie-Verlag Berlin', 'Sprache': 'Deutsch', 'Seitenzahl': 680}
>>> werk1["Erscheinungsjahr"] = 2000
>>> werk1
{'Titel': 'Kritik der reinen Vernunft', 'Autor_in': 'Kant, Immanuel', 'Erscheinungsjahr': 2000, 'Verlag': 'Akademie-Verlag Berlin', 'Sprache': 'Deutsch', 'Seitenzahl': 680}
>>> del werk1["Erscheinungsjahr"]
>>> werk1
{'Titel': 'Kritik der reinen Vernunft', 'Autor_in': 'Kant, Immanuel', 'Verlag': 'Akademie-Verlag Berlin', 'Sprache': 'Deutsch', 'Seitenzahl': 680}
>>> list(werk1.keys())
['Titel', 'Autor_in', 'Verlag', 'Sprache', 'Seitenzahl']
>>> sorted(werk1.keys())
['Autor_in', 'Seitenzahl', 'Sprache', 'Titel', 'Verlag']
>>> werk1.clear()
>>> werk1
{}
```

**Abb. 6.5** Beispiele zur Verwendung von Dictionaries

Es gibt eine weitere Möglichkeit, wie Sie ein Dictionary erzeugen können: die Funktion `dict`. Um ein Dictionary wie das obige zu erstellen, können Sie die folgenden zwei Schreibweisen verwenden:

## 6.4 Tupel: unveränderliche Daten

```
1 werk1 = dict([("Titel", "Meine geniale Freundin"), ("Autor_in", "Ferrante, Elena"), ("Erscheinungsjahr", 2018), ("Verlag", "Suhrkamp Verlag")])
```

oder

```
1 werk1 = dict(Titel = "Meine geniale Freundin", Autor_in = "Ferrante, Elena", Erscheinungsjahr = 2018, Verlag = "Suhrkamp Verlag")
```

Die letztgenannte Option kommt mit weniger Klammern und Anführungszeichen aus. Die Schlüssel werden automatisch als Strings abgespeichert. Zahlen können hierbei also nicht als Schlüssel eingegeben werden.

## 6.4 Tupel: unveränderliche Daten

Ein weiteres wichtiges Beispiel einer Datenstruktur sind Tupel. Tupel haben große Ähnlichkeit mit Listen, unterscheiden sich von letzteren jedoch darin, dass sie mit festen, unveränderbaren Werten hantieren. In vielen Anwendungen möchte man gerade mit feststehenden Werten arbeiten – so würde man etwa in unserem Bibliotheks-Verwaltungsprogramm nicht plötzlich den Titel oder das Erscheinungsjahr eines Werks ändern wollen. Um unbeabsichtigte Veränderungen zu vermeiden, ist die Verwendung von Tupeln also sinnvoll.

6

Die Tatsache, dass ein Tupel nicht veränderbar ist, bedeutet zudem weniger Rechenaufwand für den Python-Interpreter, da die Größe des Tupels konstant bleibt. Im Vergleich zu Listen haben sie daher eine bessere Performance und eignen sich sehr gut für die Verarbeitung größerer Datenmengen. Darüber hinaus können sie aufgrund ihrer Unveränderbarkeit als Schlüssel für Dictionaries verwendet werden, was bei Listen nicht möglich ist, da diese auch im Nachhinein verändert werden können. Für unser erstes Tupel-Beispiel können Sie folgende Eingaben im interaktiven Python-Interpreter machen:

### Codebeispiel #6

```
1 werk1 = ("Goethe, Johann Wolfgang von", "Faust", 2018, "Reclam")
2 werk1
3 werk1[0]
4 werk1[1:3]
5 werk1[-1]
```

## 6 Datenstrukturen in Python

```
cmd Command Prompt - python
>>> werk1 = ("Goethe, Johann Wolfgang von", "Faust", 2018, "Reclam")
>>> werk1
('Goethe, Johann Wolfgang von', 'Faust', 2018, 'Reclam')
>>> werk1[0]
'Goethe, Johann Wolfgang von'
>>> werk1[1:3]
('Faust', 2018)
>>> werk1[-1]
'Reclam'
>>>
```

**Abb. 6.6** Die Verwendung von Tupeln

Vielleicht haben Sie anhand dieses Beispiels bereits die große Ähnlichkeit von Tupeln und Listen erkannt: Im Gegensatz zu Listen werden Tupel in runden Klammern definiert und ebenfalls mit runden anstatt eckigen Klammern ausgegeben. Bei der Definition ist es sogar möglich, die runden Klammern wegzulassen und die einzelnen Elemente getrennt durch Kommas aufzulisten. Wenn Sie sich diesen kleinen optischen Unterschied merken, ist es nicht schwierig, Listen von Tupeln zu unterscheiden. Auch das Abrufen einzelner Tupelelemente oder eines Teilbereichs eines Tupels geschieht nach einem zu Listen analogen Schema. Wie oben erwähnt, liegt der Hauptunterschied zwischen Listen und Tupeln jedoch in der Unveränderbarkeit der Tupeleinträge. Beim Arbeiten mit Listen konnten wir beispielsweise das Erscheinungsjahr eines Werks mithilfe des Befehls `werk1[2] = 2019` nachträglich ändern. Wenn Sie dies bei einem Tupel versuchen, erhalten Sie hingegen eine Fehlermeldung. Die Unveränderbarkeit von Tupeln bedeutet selbstverständlich, dass sich aus einem Tupel keine einzelnen Elemente mithilfe der `del`-Funktion löschen lassen. Auch hier wird eine Fehlermeldung ausgegeben. Lediglich das gesamte Tupel kann mithilfe der `del`-Funktion gelöscht werden.

### Codebeispiel #7

```
1 werk1[2] = 2019
2 del werk1[0]
3 del werk1
4 werk1
```

```
cmd Command Prompt - python
>>> werk1[2] = 2019
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> del werk1[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
>>> del werk1
>>> werk1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'werk1' is not defined
>>>
```

**Abb. 6.7** Bei Tupeln sind die Inhalte nicht änderbar

## 6.4 Tupel: unveränderliche Daten

Leere Tupel haben die Form () und lassen sich – genauso wie Tupel, die bereits Elemente enthalten – durch das Hinzufügen von Werten erweitern. Dies wird wie bei Listen durch das Pluszeichen bewerkstelligt. Der hinzuzufügende Wert muss dabei jedoch auch ein Tupel sein. Wenn wir nun beispielsweise den Standort eines Werks zum Inhalt des Tupels hinzufügen wollen, müssen wir hierfür zunächst ein separates Tupel mit der Standortangabe erzeugen. Damit der Interpreter dessen Inhalt als Tupel und nicht als String erkennt, muss nach der Standortangabe unbedingt ein Komma stehen:

### *Codebeispiel #8*

```
1 standort_werk1 = "GB 3002",
```

Diese Schreibweise müssen Sie bei allen einelementigen Tupeln einhalten. In dem Fall, dass das ergänzende Tupel aus mehr als einem Eintrag besteht, werden die einzelnen Elemente wie gewohnt durch Kommas voneinander getrennt. Nun kann das ursprüngliche Tupel durch den neuen Wert erweitert werden:

```
1 werk1 = werk1 + standort_werk1
2 werk1
```

Das neue Element kann ebenfalls am Anfang des Tupels hinzugefügt werden:

```
1 werk1 = standort_werk1 + werk1
2 werk1
```

```
Command Prompt - python
>>> werk1 = ("Goethe, Johann Wolfgang von", "Faust", 2018, "Reclam")
>>> standort_werk1 = "GB 3002",
>>> werk1 = werk1 + standort_werk1
>>> werk1
('Goethe, Johann Wolfgang von', 'Faust', 2018, 'Reclam', 'GB 3002')
>>> werk1 = standort_werk1 + werk1
>>> werk1
('GB 3002', 'Goethe, Johann Wolfgang von', 'Faust', 2018, 'Reclam', 'GB 3002')
>>>
```

**Abb. 6.8** Ein Tupel erweitern

Tupel sind zwar im Prinzip unveränderlich, können jedoch änderbare Datenstrukturen aufnehmen. So kann man beispielsweise ein Tupel erzeugen, das eine Liste enthält, und nachträglich ein Feld innerhalb dieser Liste ändern. Möchten Sie etwa den Standort eines Buches in der Bibliothek ändern, so können Sie den Standort-Wert zunächst mittels eckiger Klammern als einelementige Liste definieren. Anschließend können Sie diesen Wert ändern:

## 6 Datenstrukturen in Python

### Codebeispiel #9

```
1 werk1 = ("Goethe, Johann Wolfgang von", "Faust", 2018, "Reclam", ["GB 3002"])
2 werk1[4][0] = "GK 3831"
3 werk1
```

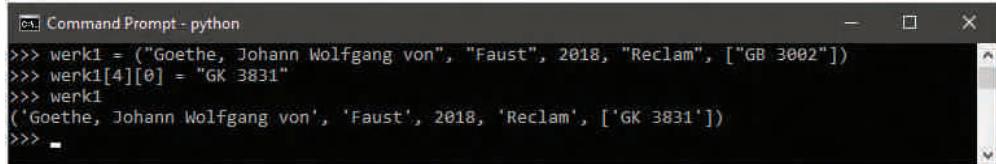


Abb. 6.9 Den Inhalt einer Liste innerhalb eines Tupels verändern

Wie dieses Beispiel zeigt, müssen Sie für den Zugriff auf das gewünschte Feld zunächst den Index innerhalb des Tupels und anschließend den der Liste angeben – auch dann, wenn die Liste nur ein Element enthält. Wenn Sie innerhalb eines Tupels Werte ändern möchten, sollten Sie vorher überlegen, ob der Vorteil unveränderlicher Daten den Mehraufwand für die Verwendung einer zusätzlichen Liste rechtfertigt.

## 6.5 Mit Mengen arbeiten

Mengen (engl. *set*) in Python haben große Ähnlichkeiten mit Mengen in der Mathematik. Auch hier werden die einzelnen Elemente, die die Menge ausmachen, innerhalb geschweifter Klammern durch Kommas voneinander getrennt aufgelistet. Duplikate werden dabei nicht mitgezählt – jedes Element kommt nur ein einziges Mal vor. So würde die Menge `zahlen = {1, 1, 0}` beim späteren Aufrufen von `zahlen` lediglich die Menge `{0, 1}` anzeigen. Wie Sie hieraus bereits erraten können, spielt die Auflistungsreihenfolge der einzelnen Elemente einer Menge dabei keine Rolle.

Da die Schreibweise für Mengen sehr stark an diejenige von Dictionaries (Abschnitt 6.3.) erinnert, bringt sie eine Definitionsschwierigkeit mit sich: Leere Mengen würden auf dieselbe Weise wie Dictionaries angelegt werden müssen. Um ein leeres Dictionary von einer leeren Menge unterscheiden zu können, verwendet man daher die folgende äquivalente Schreibweise: `menge1 = set()`. Der Name der Menge ist dabei wie bei Variablen frei wählbar.

Wir wollen nun die Menge der ersten fünf Primzahlen anlegen:

### Codebeispiel #10

```
1 primzahlen_5 = {5, 7, 3, 11, 2}
2 primzahlen_5
```

Wie Sie sehen, ordnet Python die Zahlen der Menge bei der Ausgabe automatisch der Größe nach an. Dies verdeutlicht zudem, dass die Elemente einer Menge keine be-

## 6.5 Mit Mengen arbeiten

stimmte Reihenfolge haben und somit nicht wie andere Datenstrukturen mittels Indizes referenziert werden können.

Nun erstellen wir eine zweite Menge, die die 6. bis einschließlich 10. Primzahl enthält:

```
1 primzahlen_6bis10 = {13, 17, 19, 23, 29}
```

Diese beiden Mengen lassen sich in Python durch den folgenden Befehl in einer neuen Menge, wir nennen sie `primzahlen_10`, zusammenführen:

```
1 primzahlen_10 = primzahlen_5 | primzahlen_6bis10 # Vereinigung beider Mengen
2 primzahlen_10
```

Zudem besteht die Möglichkeit, sich das jeweils kleinste bzw. größte Element der Menge ausgeben zu lassen:

```
1 min(primzahlen_10)
2 max(primzahlen_10)
```

```
6
C:\> Command Prompt - python
>>> primzahlen_5 = {5, 7, 3, 11, 2}
>>> primzahlen_5
{2, 3, 5, 7, 11}
>>> primzahlen_6bis10 = {13, 17, 19, 23, 29}
>>> primzahlen_10 = primzahlen_5 | primzahlen_6bis10 # Vereinigung beider Mengen
>>> primzahlen_10
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
>>> min(primzahlen_10)
2
>>> max(primzahlen_10)
29
>>>
```

**Abb. 6.10** Zwei Mengen zusammenfügen, kleinstes und größtes Element einer Menge ausgeben

Mathematisch ausgedrückt bildet die Operation mit `|` die Vereinigungsmenge zweier Mengen. Darüber hinaus gibt es die Mengenoperationen der Schnittmenge (`&`) und der Differenz (`-`).

In Python lassen sich zwei verschiedene Mengentypen definieren: `set` und `frozenset`. Die beiden Typen unterscheiden sich insofern, als dass ersterer auch im Nachhinein noch änderbar ist, während beim `frozenset`-Typ – genauso wie bei Strings oder Tupeln – nachträgliche Änderungen nicht mehr möglich sind. In Bezug auf Strings werden wir diese Eigenschaft im nächsten Teilkapitel noch genauer untersuchen.

Eine durch den Befehl `set()` angegebene Menge lässt sich also durch Löschen oder Hinzufügen von Elementen reduzieren bzw. erweitern. Als Argumente innerhalb der Klammern dürfen dabei nur iterierbare Datentypen bzw. -strukturen wie Strings, Lis-

## 6 Datenstrukturen in Python

ten, Dictionaries oder Tupel stehen, die Python sodann in ihre einzelnen Bestandteile zerlegt und zu einer Menge umwandelt.

Wir wollen hierzu zunächst die oben erwähnten Mengenoperationen der Schnittmenge (&) und der Differenz (-) anhand zweier durch `set` definierter Strings untersuchen.

### Codebeispiel #11

```

1  menge1 = set("Sara")
2  menge1
3  menge2 = set("Sahara")
4  menge2
5  menge1 & menge2 # Durchschnitt beider Mengen
6  menge1 - menge2 # Differenz beider Mengen
7  menge2 - menge1 # Umgekehrte Differenz beider Mengen

```

```

C:\ Command Prompt - python
>>> menge1 = set("Sara")
>>> menge1
{'a', 'r', 's'}
>>> menge2 = set("Sahara")
>>> menge2
{'a', 'h', 'r', 's'}
>>> menge1 & menge2 # Durchschnitt beider Mengen
{'a', 'r', 's'}
>>> menge1 - menge2 # Differenz beider Mengen
set()
>>> menge2 - menge1 # Umgekehrte Differenz beider Mengen
{'h'}
>>>

```

**Abb. 6.11** Mathematische Operationen auf Mengen

Beachten Sie hier, dass Python beim Erstellen der Mengen automatisch Duplikate aus den Strings entfernt. Listen, Tupel und Dictionaries lassen sich genauso mit `set()` als Mengen definieren:

### Codebeispiel #12

```

1  menge1 = set([1, 2, 3]) # Liste mit Zahlen
2  menge1
3  menge2 = set(["Alex", "Valeria", "George"]) # Liste mit Strings
4  menge2
5  menge3 = set({"Land": "Kenia", "Sprache": "Swahili"}) # Dictionary
6  menge3
7  menge4 = set(("orange", "lila")) # Tupel
8  menge4

```

## 6.5 Mit Mengen arbeiten

```

Command Prompt - python
>>> menge1 = set([1, 2, 3]) # Liste mit Zahlen
>>> menge1
{1, 2, 3}
>>> menge2 = set(["Alex", "Valeria", "George"]) # Liste mit Strings
>>> menge2
{'George', 'Valeria', 'Alex'}
>>> menge3 = set({"Land": "Kenia", "Sprache": "Swahili"}) # Dictionary
>>> menge3
{'Sprache', 'Land'}
>>> menge4 = set(("orange", "lila")) # Tupel
>>> menge4
{'orange', 'lila'}
>>>

```

Abb. 6.12 Mengen mit `set()` aus unterschiedlichen Datentypen erzeugen

Beachten Sie, dass Sie Zahlenmengen mit `set()` nur erzeugen können, indem Sie diese wie im obigen Beispiel als Liste (alternativ: Tupel) eingeben, da sich Zahlen aufgrund der Iterierbarkeitsbedingung von `set()` nicht direkt in die Klammer schreiben lassen.

Möglicherweise fragen Sie sich, wofür die Implementierung dieses doch sehr mathematisch anmutenden Mengenkonzepts in Python nützlich sein kann. Es gibt jedoch zahlreiche spannende Anwendungsgebiete: Anhand der oben vorgestellten Beispiele lassen sich etwa Textanalysen zu Satzinhalteten durchführen oder Verschlüsselungsverfahren realisieren.

6

Mithilfe der Befehle `remove()`, `add()` und `clear()` können Sie einer Menge Elemente hinzufügen und einzelne oder alle Elemente daraus löschen.

#### Codebeispiel #13

```

1 zweierreihe = set("2467")
2 zweierreihe
3 zweierreihe.remove("7")
4 zweierreihe.add("8")
5 zweierreihe
6 zweierreihe.clear()
7 zweierreihe

```

```

Command Prompt - python
>>> zweierreihe = set("2467")
>>> zweierreihe
{'4', '2', '7', '6'}
>>> zweierreihe.remove("7")
>>> zweierreihe.add("8")
>>> zweierreihe
{'4', '2', '8', '6'}
>>> zweierreihe.clear()
>>> zweierreihe
set()
>>>

```

Abb. 6.13 Elemente hinzufügen und löschen

## 6 Datenstrukturen in Python

Die Zahlen aus zweierreihe werden bei dieser Methode als Strings eingegeben und als veränderbares set abgespeichert. Die anfangs vorgestellte Schreibweise zweierreihe = {2, 4, 6, 7} ist hierzu also nicht äquivalent, da es sich bei den Elementen dabei um Zahlen, nicht um Strings handelt.

Versuchen Sie diese Operationen mit einem frozenset, erscheint aufgrund der Unveränderbarkeit dieses Mengentyps eine Fehlermeldung:

### Codebeispiel #14

```
1 zweierreihe = frozenset("2467")
2 zweierreihe
3 zweierreihe.remove("7")
```

```
Command Prompt - python
>>> zweierreihe = frozenset("2467")
>>> zweierreihe
frozenset(['4', '2', '7', '6'])
>>> zweierreihe.remove("7")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'remove'
>>>
```

**Abb. 6.14** Fehlermeldung bei der Änderung von frozenset

Beachten Sie hier ebenfalls der sich von den vorherigen Definitionsmethoden für Mengen unterscheidende Output von frozenset.

Wie Sie bemerkt haben, sind die Operationen auf und Gestaltungsmöglichkeiten von Mengen sehr vielfältig. Experimentieren Sie selbst mit weiteren Kombinationen, um im Umgang mit Mengen mehr Sicherheit zu erlangen!

## 6.6 Weiterführendes über Strings

In Kapitel 5 haben wir bereits den String-Datentyp kennen gelernt. Hierbei handelt es sich um eine Zeichenkette, die aus einzelnen Buchstaben, Wörtern oder mehreren Sätzen bestehen kann. Die Bezeichnungen String und Zeichenkette werden dabei synonym verwendet. Strings müssen stets in einfachen oder doppelten Anführungszeichen stehen, damit Python sie richtig abspeichern und erkennen kann.

Ähnlich wie sich einzelne Elemente in Listen und Tupeln handhaben lassen (Kapitel 6.2. und 6.4.), kann man auch bei Strings mittels eckiger Klammern und eines bei 0 beginnenden Index auf eines oder mehrere der Zeichen einer Zeichenkette zugreifen. Hat man beispielsweise einen Wochentag-String mit der Zeichenfolge Freitag erstellt, für den man nun eine Abkürzung verwenden möchte, so kann man hierfür folgenden Befehl eingeben:

## 6.6 Weiterführendes über Strings

### Codebeispiel #15

```
1 wochentag = "Freitag"
2 wochentag[0:4]
```

```
>>> wochentag = "Freitag"
>>> wochentag[0:4]
'Frei'
>>>
```

**Abb. 6.15** Zeichen innerhalb eines Strings auswählen

Im Unterschied zu Listen lässt sich der Inhalt eines Strings im Nachhinein jedoch nicht mehr ändern. Die Eingabe `wochentag[0:4] = "Sonn"` liefert daher eine Fehlermeldung. Der Inhalt des Strings ist nur änderbar, wenn man ihn durch die erneute Eingabe von `wochentag = "Sonntag"` überschreibt und als Variable neu definiert:

### Codebeispiel #16

```
1 wochentag[0:4] = "Sonn"
2 wochentag = "Sonntag"
3 wochentag
```

```
>>> wochentag[0:4] = "Sonn"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> wochentag = "Sonntag"
>>> wochentag
'Sonntag'
>>>
```

**Abb. 6.16** Fehlermeldung beim Ändern von Strings

Genauso lassen sich durch den Befehl `del` keine einzelnen Elemente innerhalb des Strings löschen. Lediglich der gesamte String kann mit `del` gelöscht werden:

### Codebeispiel #17

```
1 del wochentag[0:3]
2 del wochentag
3 wochentag
```

## 6 Datenstrukturen in Python

Abb. 6.17 Strings löschen

Strings lassen sich auf vielfältige Art und Weise bearbeiten und kommen in Python daher sehr häufig zum Einsatz. Dass sich Strings durch den Multiplikator \* beliebig oft aneinanderreihen lassen, haben wir bereits in Kapitel 5.3. gesehen. Einige weitere gängige Operationen mit Strings wollen wir hier in Kürze vorstellen:

Zwei oder mehrere Strings lassen sich mittels *Konkatenation* zusammenfügen, wobei die ursprüngliche Reihenfolge der Eingabe erhalten bleibt:

### Codebeispiel #18

```
1 anrede = "Hallo, "
2 name = "Anna"
3 anrede + name
```

Das Gleiche lässt sich übrigens erreichen, indem man zwei oder mehrere Strings hintereinanderstellt:

```
1 "Hallo, " "Anna"
```

Abb. 6.18 Verschiedene Möglichkeiten des Zusammenfügens von Strings

An dieser Stelle sei sogleich eine Möglichkeit erwähnt, wie sich die Grußformel eleganter formatieren lässt. Folgendermaßen erhält man damit einen String, der einen durch ein Ausrufezeichen abgeschlossenen Satz bildet, in den die entsprechenden Namen der Begrüßten eingefügt werden kann:

## 6.6 Weiterführendes über Strings

### Codebeispiel #19

```

1 name = "Anna"
2 name2 = "Ahmed"
3 "Hallo, %s!" % name
4 "Hallo, %s!" % name2

```

Der neue String besteht aus Buchstaben sowie dem Parameter (auch *Argument* genannt) `%s`, für den in der Ausgabe der entsprechende Name eingefügt wird. Das `s` nach dem `%`-Operator gibt dabei an, dass es sich bei dem Parameter um einen String-Datentyp handelt. Um mehrere Argumente innerhalb der Begrüßungsformel zu verwenden, setzt man diese ganz einfach nach dem Referenzieren durch das `%`-Zeichen in Klammern:

```
1 "Hallo, %s und %s!" % (name, name2)
```



6

**Abb. 6.19** Formatierung einer Begrüßungsformel

Auch die sogenannten *Zugehörigkeitsoperatoren* `in` und `not in`, die wir im nächsten Kapitel genauer kennen lernen werden, können auf Strings angewandt werden. Diese Operatoren testen, ob eine bestimmte Zeichenfolge Teil der gesamten Zeichenkette ist oder nicht, und geben dementsprechend `True` oder `False` aus:

### Codebeispiel #20

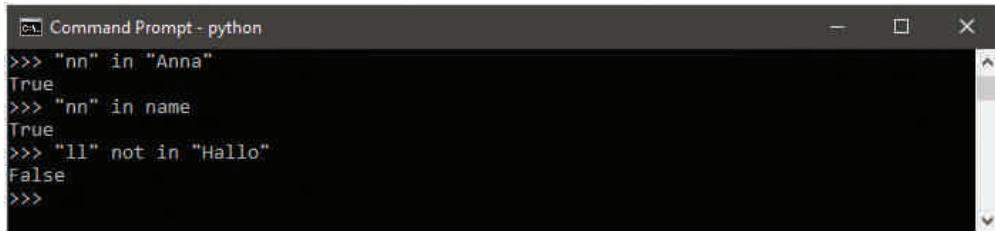
```

1 "nn" in "Anna"
2 "nn" in name
3 "ll" not in "Hallo"

```

Die Buchstabenfolge `nn` befindet sich tatsächlich im `name`-String, der den Inhalt `Anna` hat. Auch `ll` ist in `Hallo` enthalten. Gleichzeitig ist es also falsch, dass `ll` *nicht* im `Hallo`-String enthalten ist:

## 6 Datenstrukturen in Python



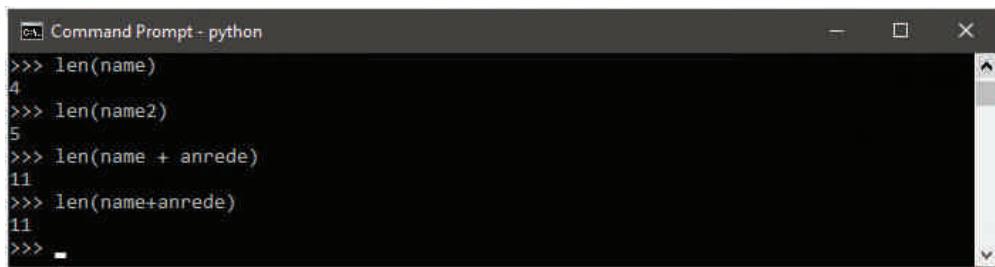
```
Command Prompt - python
>>> "nn" in "Anna"
True
>>> "nn" in name
True
>>> "ll" not in "Hallo"
False
>>>
```

Abb. 6.20 Zugehörigkeitsoperatoren bei Strings

Eine weitere praktische Funktion zum Arbeiten mit Strings ist die `len`-Funktion (von engl. *length*). Diese Funktion gibt die Länge des Strings, das heißt die Anzahl der Zeichen an, aus denen der String besteht:

### Codebeispiel #21

```
1 len(name)
2 len(name2)
3 len(name + anrede)
4 len(name+anrede)
```



```
Command Prompt - python
>>> len(name)
4
>>> len(name2)
5
>>> len(name + anrede)
11
>>> len(name+anrede)
11
>>> -
```

Abb. 6.21 Abzählen der Zeichen; die Leerzeichen in den Klammern werden nicht mitgezählt

Wie Sie sehen, spielt es dabei keine Rolle, ob zwischen der Konkatenation zweier Strings ein Leerzeichen steht oder nicht. Die `len`-Funktion lässt sich übrigens auch auf Listen, Dictionaries und Tupel anwenden. Hierbei gibt sie jeweils die Anzahl der Elemente (bzw. bei Dictionaries die Anzahl der Schlüssel-Wert-Paare) an, aus denen die Datenstruktur besteht. Enthält eine Zeichenkette oder eine Datenstruktur keine Elemente, so liefert `len` den Ausgabewert 0.

Weitere Operationen auf Strings sind im unten angegebenen Code zusammengefasst. Je nach Nutzeranforderungen gibt es weitaus mehr Möglichkeiten, wie sich Strings bearbeiten lassen. Für unsere Zwecke sollten die hier vorgestellten Methoden jedoch ausreichen.

## Codebeispiel #22

```

1 anrede = " Hallo "
2 anrede = anrede.strip() # entfernt Leerzeichen am Anfang oder am Ende des
3 Strings
4 anrede
5 anrede.upper() # setzt den gesamten String in Großbuchstaben
6 anrede.lower() # setzt den gesamten String in Kleinbuchstaben

```

```

C:\ Command Prompt - python
>>> anrede = " Hallo "
>>> anrede = anrede.strip() # entfernt Leerzeichen am Anfang oder am Ende des Strings
>>> anrede
'Hello'
>>> anrede.upper() # setzt den gesamten String in Großbuchstaben
'HALLO'
>>> anrede.lower() # setzt den gesamten String in Kleinbuchstaben
'hallo'
>>>

```

Abb. 6.22 Einige weitere Bearbeitungsmöglichkeiten für Strings

## 6.7 Operatoren in Python

Nachdem Sie nun mit Variablen sowie den grundlegenden Datentypen und -strukturen samt deren Verwendungsmöglichkeiten vertraut sind, möchten wir Ihnen an dieser Stelle einen Überblick über die wichtigsten Operatoren in Python geben. Python verfügt über eine Vielzahl an Operatoren, mithilfe derer sich unterschiedliche Rechenoperationen – beispielsweise arithmetischer oder logischer Art – mit Objekten wie Zahlen oder Variablen durchführen lassen. Der *Operand* ist dabei derjenige Teil des Ausdrucks, auf den ein *Operator* ausgeübt wird. Da Zuweisungsoperatoren bereits in Kapitel 5.2. behandelt wurden, werden wir sie hier nicht erneut anführen.

6

### Arithmetische Operatoren

Die einfachsten Operatoren, die Sie bereits in den Kapiteln 3 und 5 kennen gelernt haben, sind arithmetische Operatoren. Mit ihnen lassen sich komplexe mathematische Berechnungen vornehmen.

Operator	Beschreibung	Beispiel
+	Addition zweier Operanden oder unäres Pluszeichen	x + y +2
-	Subtraktion des rechten Operanden vom linken, oder unäres Minuszeichen	x - y -2
*	Multiplikation von zwei Operanden	x * y

## 6 Datenstrukturen in Python

/	Division des linken Operanden durch den rechten (ergibt immer einen float-Datentyp)	$x / y$
%	Modulo: Rest der Division des linken Operanden durch den rechten	$x \% y$ (Rest aus $x/y$ )
//	Abrundungsdivision: auf die nächste ganze Zahl abgerundeter Quotient zweier Zahlen	$x // y$
**	Linker Operand (Basis) potenziert mit rechtem Operanden (Exponent)	$x^{**}y$ (Potenz: $xy$ )

**Tab. 6.1** Arithmetische Operatoren in Python

Machen Sie nun zum Ausprobieren die folgenden Eingaben im Python-Prompt:

### Codebeispiel #23

```

1 x = 7
2 y = 2
3 x + y
4 x - y
5 x * y
6 x / y
7 x // y
8 x ** y

```

**Abb. 6.23** Output arithmetischer Operationen

## Vergleichsoperatoren

Mithilfe von Vergleichsoperatoren (auch *relationale Operatoren* genannt) lassen sich Werte und längere arithmetische Ausdrücke vergleichen. Es handelt sich dabei um sogenannte zweistellige logische Operatoren. Das sind Operatoren, die sich auf jeweils

zwei Argumente beziehen und als Ergebnis einen Wahrheitswert – True oder False – liefern.

Operator	Beschreibung	Beispiel
>	Größer als – True, wenn der linke Operand größer ist als der rechte	x > y
<	Kleiner als – True, wenn der linke Operand kleiner ist als der rechte	x < y
==	Gleich – True, wenn beide Operanden gleich groß sind	x == y
!=	Ungleich – True, wenn die Operanden nicht gleich groß sind	x != y
>=	Größer gleich – True, wenn der linke Operand größer als der rechte oder gleich groß ist	x >= y
<=	Kleiner gleich – True, wenn der linke Operand kleiner als der rechte oder gleich groß ist	x <= y

Tab. 6.2 Vergleichsoperatoren in Python

Nehmen Sie nun die folgenden Eingaben vor:

6

#### Codebeispiel #24

```

1 x = 3
2 y = 7
3 x > y
4 "Python" > "Java"
5 x < y
6 [1, 4, 5] < [0, 1]
7 x == y
8 {1, 2, 3} == {1, 3, 2}
9 (1, 2, 3) == (1, 3, 2)
10 x != y
11 "Sarah" != "Sahara"
12 x >= y
13 x <= y

```

## 6 Datenstrukturen in Python

```
>>> x = 3
>>> y = 7
>>> x > y
False
>>> "Python" > "Java"
True
>>> x < y
True
>>> [1, 4, 5] < [0, 1]
False
>>> x == y
False
>>> {1, 2, 3} == {1, 3, 2}
True
>>> (1, 2, 3) == (1, 3, 2)
False
>>> x != y
True
>>> "Sarah" != "Saraha"
True
>>> x >= y
False
>>> x <= y
True
```

**Abb. 6.24** Output unterschiedlicher Vergleichsoperationen

Auch Verkettungen von mehreren Werten wie beispielsweise `7 < x <= 199` sind in Python möglich. Beim Vergleichen von Strings wird die lexikographische Ordnung ihrer Zeichen betrachtet: Im obigen Beispiel kommt das P in Python im Alphabet nach dem J in Java und macht den Python-String daher größer. Sind die ersten beiden Zeichen zweier Strings identisch, wird überprüft, welches der zweiten Zeichen der jeweiligen Strings im Alphabet zuerst kommt, usw.

### Boolesche Operatoren

Boolesche Operatoren sind logische Operatoren, die aus einer Verknüpfung aus der sogenannten booleschen Algebra bestehen. In Python sind dies die Operatoren `and`, `or` und `not`.

Für die *Konjunktion* `and` müssen beide Operanden wahr sein, damit das Ergebnis ebenfalls wahr wird. Bei der *Disjunktion* `or` ist es ausreichend, wenn einer der beiden Operanden wahr ist. Die *Negation* `not` kehrt den Wahrheitswert einer Aussage um.

Die Wahrheitstabellen der drei booleschen Operatoren sind die folgenden:

Wahrheitstabelle für and			Wahrheitstabelle für or			Wahrheitstabelle für not	
A	B	A and B	A	B	A or B	A	B
True	True	True	True	True	True	True	False
True	False	False	True	False	True	False	True
False	True	False	False	True	True	False	True
False	False	False	False	False	False	True	False

**Tab. 6.3** Wahrheitstabellen für and, or und not

In Python ist jedem Ausdruck und jedem Objekt übrigens automatisch ein bestimmter Wahrheitswert zugeordnet. So sind alle Zahlen ungleich Null wahr, während numerische Ausdrücke, die Null ergeben, falsch sind. Das Gleiche gilt für leere Strings und Listen. Enthalten sie Elemente, ist ihr Wahrheitswert wahr.

Hier einige Verwendungsbeispiele:

#### Codebeispiel #25

```

1 x = True
2 y = False
3 x and y
4 x or y
5 not x
6 z = 7
7 z > 0 and z >= 7
8 (1 < 2) and (3 < 4)
9 not (7 > 1)
```

6

**Abb. 6.25** Verwendung boolescher Operatoren

Die Klammern in den obigen Beispielen sind nicht notwendig, wurden jedoch aufgrund der Übersichtlichkeit der Ausdrücke gesetzt. Die Variablen x und y sind vom sogenannten *booleschen Datentyp*: Dieser Datentyp nimmt stets einen von zwei

## 6 Datenstrukturen in Python

Wahrheitswerten – „wahr“ (`True`) oder „falsch“ (`False`) – an. Beachten Sie, dass die beiden Werte stets mit Großbuchstaben geschrieben werden müssen.

### Zugehörigkeitsoperatoren

Die Zugehörigkeitsoperatoren in Python heißen `in` und `not in`. Mit ihnen lässt sich überprüfen, ob eine Variable oder ein Zeichen in einem String, einer Liste, einem Tupel, einer Menge oder einem Dictionary enthalten ist. Im Falle von Dictionaries lässt sich lediglich testen, ob das Zeichen als Schlüssel vorkommt, nicht als Wert.

Operator	Beschreibung	Beispiel
<code>in</code>	True, wenn der Wert vorhanden ist	<code>5 in x</code>
<code>not in</code>	True, wenn der Wert nicht enthalten ist	<code>5 not in x</code>

**Tab. 6.4** Zugehörigkeitsoperatoren in Python

Wir werden die Funktionsweise der Zugehörigkeitsoperatoren anhand der folgenden Befehle für eine Zeichenkette, ein Dictionary und ein Tupel testen:

#### Codebeispiel #26

```

1 x = "Hello universe"
2 y = {"Obst": "Banane", "Gemüse": "Broccoli"}
3 z = (11, 77)
4 "H" in x
5 "Universe" in x
6 "Obst" in y
7 "Broccoli" in y
8 1 in z

```

```

Command Prompt - python
>>> x = "Hello universe"
>>> y = {"Obst": "Banane", "Gemüse": "Broccoli"}
>>> z = (11, 77)
>>> "H" in x
True
>>> "Universe" in x
False
>>> "Obst" in y
True
>>> "Broccoli" in y
False
>>> 1 in z

```

**Abb. 6.26** Verwendungsbeispiele für Zugehörigkeitsoperatoren

Hier befindet sich `H` in `x`, wo hingegen `Universe` nicht in `x` enthalten ist, da Python die unterschiedliche Groß- und Kleinschreibung registriert. Während im Dictionary der Schlüssel `Obst` gefunden werden kann, ist dies beim Wert `Broccoli` nicht der Fall.

## 6.8 Übung: Mit unterschiedlichen Datenstrukturen arbeiten

### 6.8 Übung: Mit unterschiedlichen Datenstrukturen arbeiten

#### Übungsaufgaben

1. Stellen Sie sich vor, Sie schreiben ein Programm, das den Bestand eines Supermarkts verwaltet. Wählen Sie drei Produkte, zu denen Sie jeweils die Produktbezeichnung, den Barcode, Preis und Regalstandort abspeichern. Legen Sie diese Informationen jeweils in Listen, Dictionaries und Tupeln an.
  - ▶ Ändern Sie in den Listen im Nachhinein den Preis eines Artikels und fügen Sie einen weiteren Eintrag über die vorrätige Anzahl des Artikels hinzu. Lassen Sie sich diesen abgeänderten Artikel ausgeben.
  - ▶ Lassen Sie sich bei den Dictionaries die jeweiligen Standorte der Produkte anzeigen.
  - ▶ Geben Sie für die Tupel alle drei Produkte auf dem Bildschirm aus. Dabei soll die Ausgabe nicht die kompletten Tupel anzeigen, sondern jeden Eintrag in einer einzelnen Zeile auflisten. Die drei Produkte sollen jeweils durch eine Leerzeile voneinander getrennt werden. Hierfür dient das Zeichen \n.
2. Sie wollen drei Ihrer Lieblingslieder samt Interpreten tabellarisch erfassen. Wählen Sie eine Datenstruktur aus, die Ihnen hierfür besonders geeignet erscheint. Speichern Sie die Angaben wiederum in einer übergeordneten Datenstruktur ab. Lassen Sie sich die einzelnen Einträge anschließend separat ausgeben.
3. Erstellen Sie eine Menge, die die ersten vier Vielfachen der Zahl 7 enthält und lassen Sie sich diese ausgeben. Fügen Sie die fünfte Zahl in der Reihe hinzu, geben Sie die neue Menge aus und speichern Sie sie als frozenset ab.  
Überprüfen Sie nun,
  - ▶ ob sich die Zahl 35 in der Menge befindet,
  - ▶ ob sich sowohl 7 als auch 12 in der Menge befinden.

## 6 Datenstrukturen in Python

### Lösungen

1.

Mit Listen:

```

1 produkt1 = ["Butter", 84687875453, 1.99, "A33"]
2 produkt2 = ["Blumenkohl", 49781152003, 3.10, "X01"]
3 produkt3 = ["Spaghetti", 55698875210, 1.49, "C99"]
4 produkt3[2] = 1.99
5 produkt3 = produkt3 + [20]
6 print(produkt3)

```

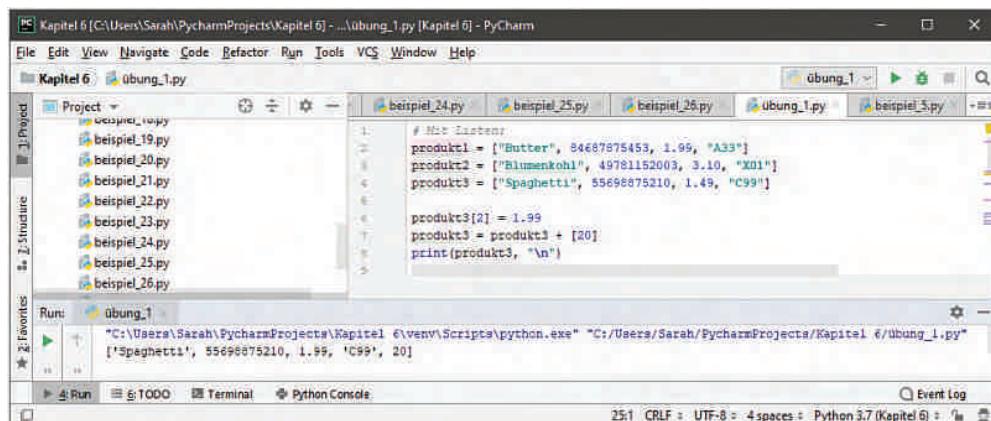


Abb. 6.27 Die Ausgabe der abgeänderten Liste

Mit Dictionaries:

```

1 produkt1 = {"Bezeichnung": "Butter", "Barcode": 84687875453, "Preis": 1.99,
2 "Regalstandort": "A33"}
3 produkt2 = {"Bezeichnung": "Blumenkohl", "Barcode": 49781152003, "Preis":
4 3.1, "Regalstandort": "X01"}
5 produkt3 = {"Bezeichnung": "Spaghetti", "Barcode": 55698875210, "Preis": 1.49,
6 "Regalstandort": "C99"}
7 print("Der Regalstandort von %s ist:" % produkt1["Bezeichnung"],
8 produkt1["Regalstandort"])
9 print("Der Regalstandort von %s ist:" % produkt2["Bezeichnung"],
10 produkt2["Regalstandort"])
11 print("Der Regalstandort von %s ist:" % produkt3["Bezeichnung"],
12 produkt3["Regalstandort"])

```

## 6.8 Übung: Mit unterschiedlichen Datenstrukturen arbeiten

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_19.py' through 'beispiel\_26.py'. Below it, the code editor displays a script named 'übung\_1.py' containing Python code that uses dictionaries to store product information. The output window shows the results of running the script, which prints the shelf locations for three products: Butter (A33), Blumenkohl (X01), and Spaghetti (C99). The status bar at the bottom indicates the file is 25:1, encoding is CRLF, encoding is UTF-8, 4 spaces are used, and the Python version is 3.7 (Kapitel 6).

```

# Mit Dictionaries
produkt1 = {"Bezeichnung": "Butter", "Barcode": 84687875453, "Preis": 1.99, "Regalstandort": "A33"}
produkt2 = {"Bezeichnung": "Blumenkohl", "Barcode": 49781152003, "Preis": 3.10, "Regalstandort": "X01"}
produkt3 = {"Bezeichnung": "Spaghetti", "Barcode": 55698875210, "Preis": 1.49, "Regalstandort": "C99"}

print("Der Regalstandort von %s ist: %s" % (produkt1["Bezeichnung"], produkt1["Regalstandort"]))
print("Der Regalstandort von %s ist: %s" % (produkt2["Bezeichnung"], produkt2["Regalstandort"]))
print("Der Regalstandort von %s ist: %s" % (produkt3["Bezeichnung"], produkt3["Regalstandort"]))

```

**Abb. 6.28** Die Ausgabe der Standorte mittels Dictionaries

Mit Tupeln:

```

1 produkt1 = ("Butter", 84687875453, 1.99, "A33")
2 produkt2 = ("Blumenkohl", 49781152003, 3.10, "X01")
3 produkt3 = ("Spaghetti", 55698875210, 1.49, "C99")
4 print(produkt1[0])
5 print(produkt1[1])
6 print(produkt1[2])
7 print(produkt1[3], "\n")
8 print(produkt2[0])
9 print(produkt2[1])
10 print(produkt2[2])
11 print(produkt2[3], "\n")
12 print(produkt3[0])
13 print(produkt3[1])
14 print(produkt3[2])
15 print(produkt3[3], "\n")

```

## 6 Datenstrukturen in Python

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists several Python files: 'beispiel\_18.py', 'beispiel\_19.py', 'beispiel\_20.py', 'beispiel\_21.py', 'beispiel\_22.py', 'beispiel\_23.py', 'beispiel\_24.py', 'beispiel\_25.py', 'beispiel\_26.py', 'Übung\_1.py', and 'Übung\_2.py'. The main code editor window displays the following Python code:

```
# Mit Tupeln:  
produkt1 = ("Butter", 84687875453, 1.99, "A33")  
produkt2 = ("Blumenkohl", 49781152003, 3.10, "X01")  
produkt3 = ("Spaghetti", 5568875210, 1.49, "C99")  
  
print(produkt1[0])  
print(produkt1[1])  
print(produkt1[2])  
print(produkt1[3], "\n")  
print(produkt2[0])  
print(produkt2[1])  
print(produkt2[2])
```

The 'Run' tool window below the editor shows the output of the 'Übung\_1' run. It lists two tuples: 'Butter' and 'Blumenkohl'. For 'Butter', it shows the values: 84687875453, 1.99, and 'A33'. For 'Blumenkohl', it shows the values: 49781152003, 3.1, and 'X01'. The bottom status bar indicates the Python version is 3.7 (Kapitel 6).

Abb. 6.29 Die Ausgabe der Tupeleinträge

2.

```
1 lieder = [{"Interpret_in": "Cyndi Lauper", "Titel": "Girls just want to have  
2 fun"}, {"Interpret_in": "A Tribe Called Quest", "Titel": "The Space Program"},  
3 {"Interpret_in": "John Lennon", "Titel": "Imagine"}]  
4 print(lieder[0]["Interpret_in"])  
5 print(lieder[0]["Titel"], "\n")  
6 print(lieder[1]["Interpret_in"])  
7 print(lieder[1]["Titel"], "\n")  
8 print(lieder[2]["Interpret_in"])  
9 print(lieder[2]["Titel"], "\n")
```

## 6.8 Übung: Mit unterschiedlichen Datenstrukturen arbeiten

```

Kapitel 6 [C:\Users\Sarah\PycharmProjects\Kapitel 6] - ...\\Übung_2.py (Kapitel 6) - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 6 Übung_2.py
Project External Libraries Run: übung_2
External Libraries
Run: übung_2
"C:\Users\Sarah\PycharmProjects\Kapitel 6\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 6/Übung_2.py"
Cyndi Lauper
Girls just want to have fun.
A Tribe Called Quest
The Space Program.
John Lennon
Imagine.

```

**Abb. 6.30** Die Ausgabe der abgeänderten Liste

6

Dieses Programm besteht aus einer übergeordneten Liste, die die einzelnen Dictionaries enthält. Sie können hierfür jedoch auch jede beliebige andere Kombination aus den bekannten Datenstrukturen wählen.

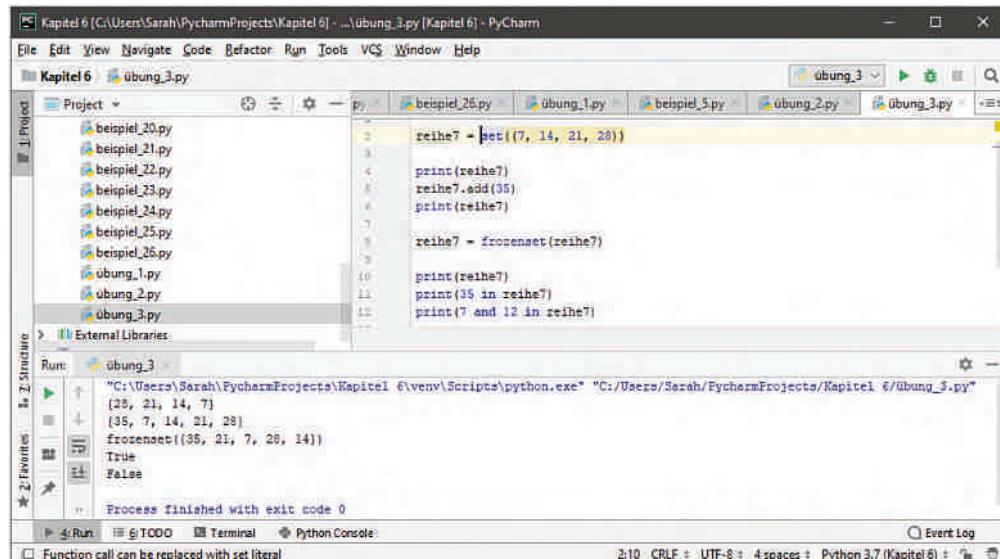
3.

```

1 reihe7 = set((7, 14, 21, 28))
2 print(reihe7)
3 reihe7.add(35)
4 print(reihe7)
5 reihe7 = frozenset(reihe7)
6 print(reihe7)
7 print(35 in reihe7)
8 print(7 and 12 in reihe7)

```

## 6 Datenstrukturen in Python



```
reihe7 = set([7, 14, 21, 28])
print(reihe7)
reihe7.add(35)
print(reihe7)

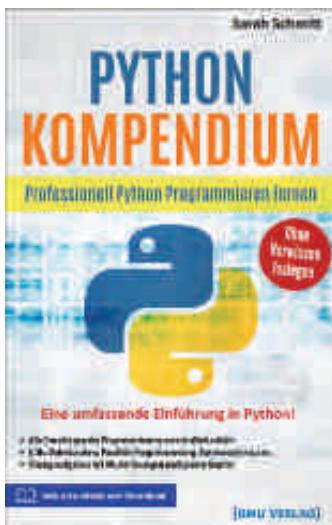
reihe7 = frozenset(reihe7)
print(reihe7)
print(35 in reihe7)
print(7 and 12 in reihe7)
```

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_20.py' through 'beispiel\_26.py', 'übung\_1.py', 'übung\_3.py', 'übung\_2.py', and 'übung\_3.py'. The main code editor window contains the provided Python code. Below the editor, the 'Run' tool window displays the execution results: the creation of a set 'reihe7' with elements 7, 14, 21, 28, adding 35 to it, creating a frozen set 'reihe7', and finally printing the set and checking for elements 35, 7, and 12. The status bar at the bottom indicates the run was successful with exit code 0.

Abb. 6.31 Die Ausgabe der Menge samt Operationen

Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 7

# Entscheidungen treffen – Programmabläufe steuern

Bisher haben wir grundlegende Konzepte und Werkzeuge kennen gelernt, die die Voraussetzung für das Erstellen von Programmen sind. Computerprogramme werden jedoch erst spannend und wirklich nützlich, wenn sie interaktiv und dynamisch auf unterschiedliche Anwendereingaben oder Ergebnisse reagieren können, die vor der Ausführung des Programms nicht bekannt waren. Das bedeutet: Ist im Laufe des Programms eine bestimmte Bedingung erfüllt, wird ein entsprechender Programmteil ausgeführt – sonst nicht.

In Kapitel 6.7 hatten wir bereits unterschiedliche Operatoren kennen gelernt. In diesem Kapitel werden wir sehen, dass einige dieser Operatoren äußerst nützlich sein können, um innerhalb des Programms Entscheidungen zu treffen. So lassen sich etwa mittels < und > Unterscheidungen für Variablen definieren, die sich innerhalb bestimmter Zahlenbereiche befinden, die wiederum unterschiedliche Programm-ausführungen zur Folge haben. Doch mehr dazu später. Zunächst werden wir den Schlüsselbegriff `if` untersuchen, der die verschiedenen Verzweigungen innerhalb des Programms ermöglicht.

### 7.1 Der Schlüsselbegriff `if`

Mittels `if`-Abfragen wird überprüft, ob eine bestimmte Bedingung erfüllt ist. Ist dies der Fall, wird daraufhin eine Verzweigung innerhalb des Programms ausgeführt. Ein `if`-Ausdruck hat allgemein die folgende Form:

```
1 if Bedingung: Anweisungen
```

Die Entscheidungsformel wird also stets durch den Schlüsselbegriff `if` (engl. für *wenn*) eingeleitet. Daraufhin folgen die Bedingung, ein Doppelpunkt und zuletzt eine oder mehrere Anweisungen, die ausgeführt werden, wenn die Bedingung erfüllt ist.

Auf den Begriff `if` können weitere vielfältige Verzweigungen folgen, die wir im Laufe dieses Kapitels kennen lernen werden. Der einfachste Fall ist jedoch die sogenannte *einseitige Verzweigung*: Trifft eine Bedingung zu, wird ein aus weiteren Anweisungen bestehender Programmteil ausgeführt. Trifft sie nicht zu, passiert nichts. Sie kennen ähnliche Situationen aus dem Alltag: „Wenn der Wecker klingelt, dann stehen Sie auf“.

## 7.2 Vergleiche: wichtig für das Aufstellen der Bedingung

In Python hat der Doppelpunkt nach der Bedingung die Aufgabe des *dann*-Begriffs im obigen Satz: Er leitet die auszuführenden Anweisungen ein. Der Anweisungsblock steht dabei meist eine Zeile unter dem `if`-Zweig mit der Bedingung und muss stets einheitlich eingerückt sein, damit Python alle einzelnen Anweisungen als solche erkennt. Lediglich dann, wenn es sich nur um eine einzige Anweisung handelt, ist es möglich, den gesamten Ausdruck in eine Zeile zu schreiben. In der Regel hat man es jedoch mit mehreren Anweisungen zu tun. Es empfiehlt sich also, für die einzelnen Befehle eine neue Zeile nach dem folgenden Schema zu verwenden:

```

1 if Bedingung:
2     Befehl 1
3     Befehl 2
4     ...
5     Befehl n

```

Die PyCharm-IDE erkennt den `if`-Begriff und nimmt die Einrückungen für die Anweisungen automatisch vor, nachdem Sie den Doppelpunkt eingegeben und **Enter** gedrückt haben. Standardmäßig ist in PyCharm eine Einrückung von vier Leerzeichen eingestellt. Sie können dies jedoch in den Einstellungen ändern. Gehen Sie hierzu auf **File → Settings... → Editor → Code Style → Python → Tabs and Indents** und wählen Sie die gewünschte Tabulator- und Leerzeichenanzahl.

Sollte Ihre IDE die Einrückungen nicht von selbst vornehmen, können Sie diese entweder durch die Tabulatortaste oder durch das Einfügen mehrerer Leerzeichen angeben. Achten Sie bloß darauf, die Einrückungen einheitlich vorzunehmen, damit der Interpreter keine Fehlermeldung ausgibt.

7

Folgen auf den Anweisungsblock weitere Zeilen ohne Einrückung, geht der Python-Interpreter davon aus, dass der Anweisungsblock an dieser Stelle endet.

## 7.2 Vergleiche: wichtig für das Aufstellen der Bedingung

Nun wollen wir genauer untersuchen, wie sich einzelne Bedingungen formulieren lassen. Hierfür kommen die im vorigen Kapitel vorgestellten Operatoren zum Einsatz. So könnten wir die Gleichheit zweier Werte beispielsweise durch den Einsatz des Vergleichsoperators `==` testen wollen. Wenn von Nutzerseite die erwartete Zahl eingegeben wurde, wird eine entsprechende Nachricht ausgegeben.

Ein komplettes Programm, das eine `if`-Abfrage implementiert, könnte demnach folgendermaßen aussehen:

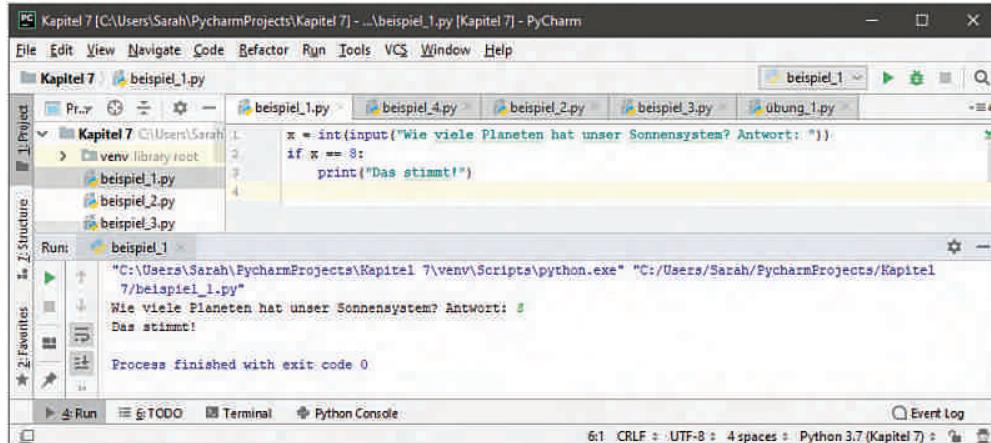
## 7 Entscheidungen treffen – Programmabläufe steuern

### Codebeispiel #1

```

1 x = int(input("Wie viele Planeten hat unser Sonnensystem? Antwort: "))
2 if x == 8:
3     print("Das stimmt!")

```



**Abb. 7.1** Die Ausführung des Programms nach der richtigen Eingabe

Hier wurde sogleich der `int`-Datentyp verlangt, da davon auszugehen ist, dass die Nutzereingabe eine ganze Zahl sein wird. Wird ein anderer Datentyp angegeben, erscheint eine Fehlermeldung. Versuchen Sie einmal, in diesem Programm eine andere ganze Zahl als 8 einzugeben! Sie sehen, dass das Programm aufgrund der einseitigen `if`-Verzweigung daraufhin keine weitere Ausgabe tätigt.

Beachten Sie hier auch den Unterschied zwischen dem Zuweisungsoperator (`=`) und dem Gleichheitsoperator (`==`). Diese beiden werden häufig verwechselt, was zu Fehlermeldungen führt. Der Zuweisungsoperator setzt eine Variable einem Wert gleich, während der Gleichheitsoperator überprüft, ob die Variable tatsächlich einen bestimmten Wert hat.

Stellen Sie sich nun vor, dass ein Supermarkt ein Programm über dessen Warenbestand erstellen möchte, in dem die Anzahl der einzelnen Produkte abgefragt wird. Liegt der Vorrat für ein bestimmtes Produkt unterhalb einer vorgegebenen Anzahl, wird eine Meldung ausgegeben, dass das jeweilige Produkt nachbestellt werden muss. Das Programm drückt somit etwa den folgenden Algorithmus aus: „Wenn die Anzahl der vorrätigen Spaghetti-Packungen weniger als 50 Stück beträgt, müssen neue nachbestellt werden“.

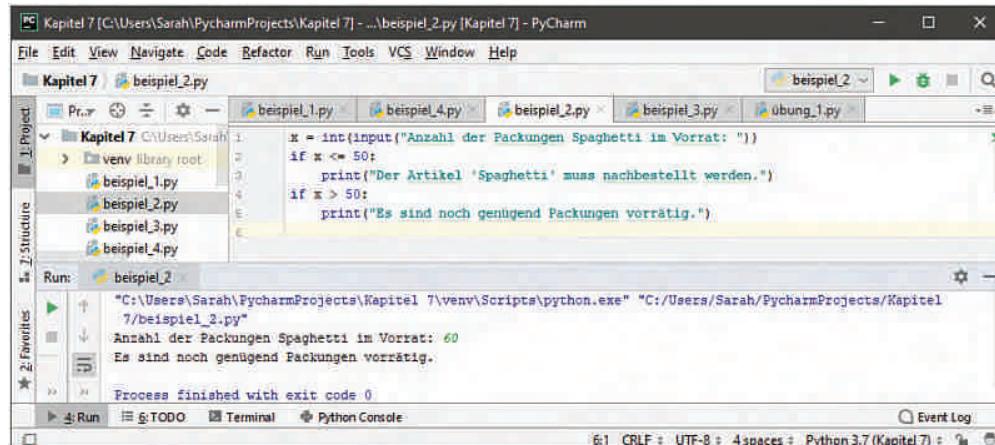
## 7.2 Vergleiche: wichtig für das Aufstellen der Bedingung

### Codebeispiel #2

```

1 x = int(input("Anzahl der Packungen Spaghetti im Vorrat: "))
2 if x <= 50:
3     print("Der Artikel 'Spaghetti' muss nachbestellt werden.")
4 if x > 50:
5     print("Es sind noch genügend Packungen vorrätig.")

```



**Abb. 7.2** Zwei Verzweigungen im Programm

7

Wie Sie sehen, lassen sich mithilfe der Vergleichsoperatoren `==`, `<`, `>`, `<=` und `>=` Verzweigungen mit mehr als nur einer `if`-Abfrage angeben. Um allzu viele Wiederholungen von `if`-Abfragen innerhalb eines Anweisungsblocks zu vermeiden, lassen sich jedoch oftmals geschickte Formulierungen vornehmen, die durch geeignete Operatoren mit möglichst wenigen `if`-Abfragen möglichst viele Fälle abdecken.

So könnte man für die Frage „Was ergibt  $11 * 11$ ?“ etwa drei verschiedene Fälle definieren: Eine Ausgabe würde die Nutzereingabe von 121 (Operator: `==`) als richtig erkennen, die zweite Ausgabe würde eingegebene Zahlen unter 121 (Operator: `<`) als falsch und die dritte würde Zahlen oberhalb von 121 (Operator: `>`) ebenfalls als falsch ausweisen.

Eine elegantere Formulierungsmöglichkeit ist hier jedoch durch die Verwendung des Vergleichsoperators `!=` gegeben. Dieser würde alle Eingaben, die nicht die richtige Antwort darstellen, unter einer `if`-Bedingung mit der entsprechenden Ausgabe zusammenfassen:

```

1 x = int(input("Was ergibt 11*11? Antwort: "))
2 if x == 121:
3     print("Ihre Antwort ist richtig.")
4 if x != 121:
5     print("Diese Eingabe ist nicht richtig.")

```

## 7 Entscheidungen treffen – Programmabläufe steuern

### 7.3 Mehrere Bedingungen verknüpfen

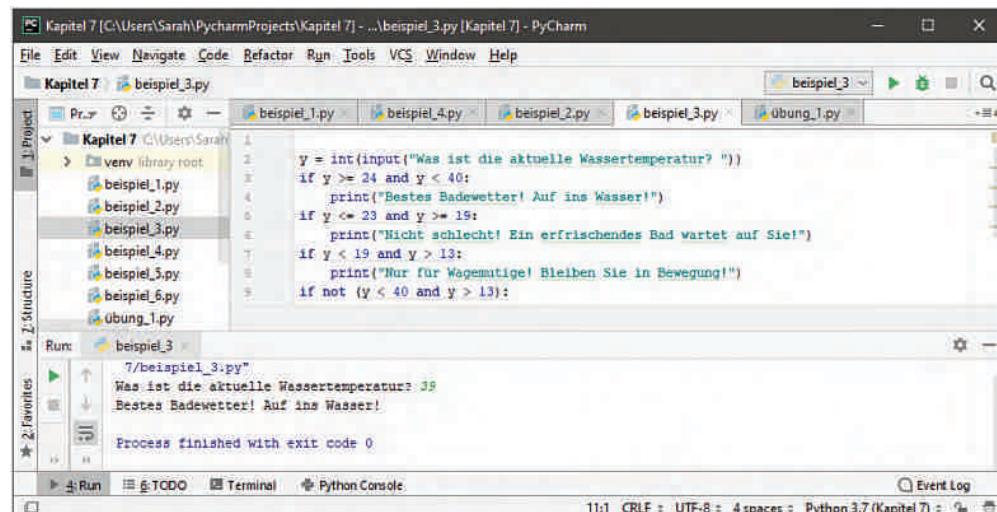
Nun wollen wir untersuchen, wie sich Verknüpfungen mehrerer Bedingungen durch die in Kapitel 6.7 vorgestellten Operatoren vornehmen lassen. Betrachten Sie dazu das folgende Beispiel:

#### Codebeispiel #3

```

1 y = int(input("Was ist die aktuelle Wassertemperatur? "))
2 if y >= 24 and y < 40:
3     print("Bestes Badewetter! Auf ins Wasser!")
4 if y <= 23 and y >= 19:
5     print("Nicht schlecht! Ein erfrischendes Bad wartet auf Sie!")
6 if y < 19 and y > 13:
7     print("Nur für Wagemutige! Bleiben Sie in Bewegung!")
8 if not (y < 40 and y > 13):
9     print("Schwimmen nicht ratsam!")

```



**Abb. 7.3** Verknüpfungen mehrerer Bedingungen

In der letzten `if`-Anweisung muss die Klammer gesetzt werden, da sich `not` ansonsten nur auf den ersten Term und nicht auf beide Terme bezieht. Dieses Beispiel sollte veranschaulichen, wie der `and`-Operator in einem logischen Ausdruck mit `if` verwendet werden kann. Eine praktikablere Variante wäre hier, anstelle von Ausdrücken wie `if y >= 24 and y < 40` die Formulierung `if 24 <= y < 40` zu wählen.

Mit dem Gleichheitsoperator `==` lassen sich übrigens nicht nur Zahlen, sondern auch Strings überprüfen. Da der `input`-Befehl standardmäßig eine Zeichenkette erwartet und davon auszugehen ist, dass die Nutzer im folgenden Beispiel eine Zeichenkette eingeben werden, muss man den Datentyp nicht im Voraus definieren:

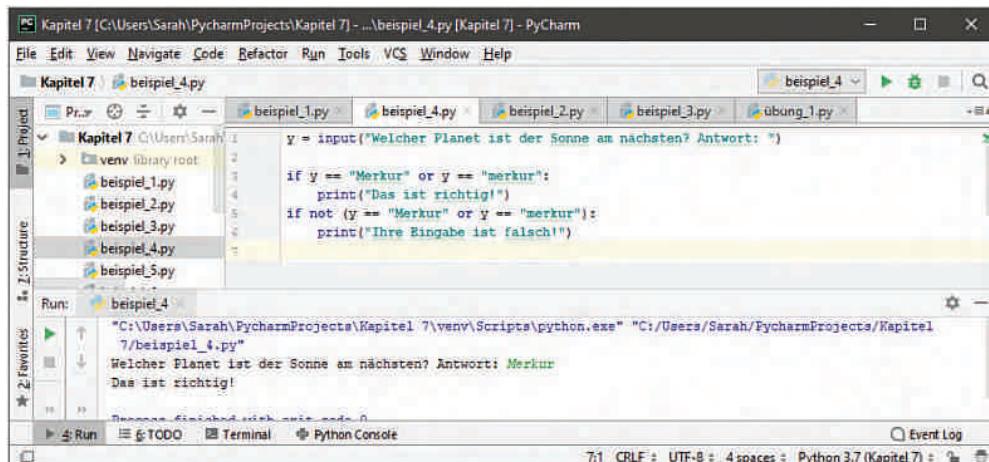
## 7.4 else und elif: weitere Alternativen hinzufügen

### Codebeispiel #4

```

1 y = input("Welcher Planet ist der Sonne am nächsten? Antwort: ")
2 if y == "Merkur" or y == "merkur":
3     print("Das ist richtig!")
4 if not (y == "Merkur" or y == "merkur"):
5     print("Ihre Eingabe ist falsch!")

```



**Abb. 7.4** Die Eingabe von Strings mit dem Gleichheitsoperator ==

7

Hier wird der boolesche Operator `or` verwendet, um weder die Klein- noch die Großschreibung der richtigen Antwort auszuschließen. Wird eine der beiden möglichen Antworten eingegeben, erfolgt eine Nachricht, dass die Eingabe richtig war. Im Zweifelsfall sollten Sie in Python-Code immer Klammern setzen, um Zugehörigkeiten klar auszuweisen und den Code übersichtlicher zu gestalten.

## 7.4 else und elif: weitere Alternativen hinzufügen

In den vorigen Abschnitten haben wir bereits einige Möglichkeiten kennen gelernt, wie sich das Gegenteil einer nicht erfüllten Bedingung formulieren lässt: Je nach Fall und Geeignetheit kann man hierzu beispielsweise den Vergleichsoperator `!=` oder den booleschen Operator `not` einsetzen, um darin alle nicht zutreffenden Fälle zusammen zu fassen. Bei Aufgabenstellungen mit einem höheren Komplexitätsgrad kann es jedoch oftmals recht kompliziert werden, das Gegenteil der `if`-Klausel auszuformulieren. Im besten Fall erzeugen die bislang bekannten Möglichkeiten dabei einen langen, unpraktischen Code. Hier verfügt Python über zwei weitere nützliche Schlüsselwörter: `else` und `elif`.

## 7 Entscheidungen treffen – Programmabläufe steuern

### 7.4.1 else

Die `else`-Verzweigung (engl. für *andernfalls*) wird in all denjenigen Fällen ausgeführt, in denen die davor genannte `if`-Bedingung nicht zutrifft. Hierbei handelt es sich um eine *zweiseitige Verzweigung*. Der Vorteil von `else` liegt darin, dass hierfür keine weitere Bedingung wie in den bisher behandelten Beispielen angegeben und ausformuliert werden muss. Innerhalb eines `else`-Blocks können wiederum weitere `if`- oder `else`-Pfade stehen, was unnötige Wiederholungen eliminiert und den gesamten Code eleganter und übersichtlicher macht. Innerhalb desselben Codes dürfen jedoch nur mehrere `else`-Klauseln vorkommen, wenn sich diese durch unterschiedliche Einrückungen auf verschiedenen Ausführungsebenen befinden und nicht auf dieselbe `if`-Klausel beziehen. Die `else`-Klausel muss stets mit einem Doppelpunkt abgeschlossen werden, genauso wie auch nach der `if`-Klausel samt ihrer Bedingung stets ein Doppelpunkt folgt.

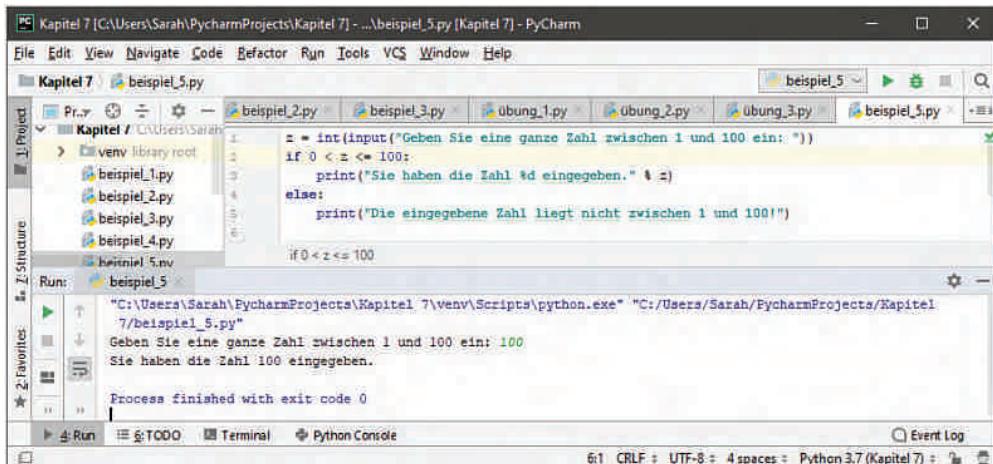
Für Anfänger mag die Verwendung von `else` zunächst eine Umstellung bedeuten, da sich die meisten Verzweigungen ebenfalls mit `if` realisieren lassen und die neue Formulierungsweise womöglich etwas ungewohnt wirkt. Sie werden jedoch schnell feststellen, dass `else` den Code logischer und einfacher gestaltet. Es ist in jedem Fall immer eine gute Idee, die `else`-Anweisung in Ihren Programmen zu verwenden, um damit ungültige Eingaben und Fehler erkennen und ausweisen zu können. Nicht zuletzt kann die Verwendung des `else`-Zweigs bei größeren Programmen vorteilhaft bezüglich der Ausführungsgeschwindigkeit sein.

Lassen Sie uns nun das folgende Beispiel betrachten:

#### Codebeispiel #5

```
1 z = int(input("Geben Sie eine ganze Zahl zwischen 1 und 100 ein: "))
2 if 0 < z <= 100:
3     print("Sie haben die Zahl %d eingegeben." %z)
4 else:
5     print("Die eingegebene Zahl liegt nicht zwischen 1 und 100!")
```

## 7.4 else und elif: weitere Alternativen hinzufügen



The screenshot shows the PyCharm IDE interface. The project is named 'Kapitel 7' and contains several files: 'beispiel\_2.py', 'beispiel\_3.py', 'Übung\_1.py', 'Übung\_2.py', 'Übung\_3.py', and 'beispiel\_5.py'. The current file being edited is 'beispiel\_5.py'. The code in 'beispiel\_5.py' is:

```
z = int(input("Geben Sie eine ganze Zahl zwischen 1 und 100 ein: "))
if 0 < z <= 100:
    print("Sie haben die Zahl %d eingegeben." % z)
else:
    print("Die eingegebene Zahl liegt nicht zwischen 1 und 100!")
if 0 < z <= 100
```

The 'Run' tab is selected, showing the command: "C:/Users/Sarah/PycharmProjects/Kapitel 7/venv/Scripts/python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 7/beispiel\_5.py". The output window displays:

```
Geben Sie eine ganze Zahl zwischen 1 und 100 ein: 100
Sie haben die Zahl 100 eingegeben.

Process finished with exit code 0
```

Abb. 7.5 Die Verwendung der `else`-Anweisung

Innerhalb des Strings wird hier als Argument der Wert der Variablen `z` eingesetzt. Der Buchstabe `d` hinter dem `%`-Operator gibt an, dass es sich dabei um eine ganze Zahl handelt. Möglicherweise erinnern Sie sich, dass wir in Kapitel 6.6 bereits String-Variablen durch den Befehl `%s` eingefügt hatten. Fließkommazahlen lassen sich in analoger Weise durch `%f` verwenden. Möchten Sie mehrere Werte referenzieren, so müssen Sie diese nach dem `%` in Klammern setzen und durch Kommas voneinander trennen. In der Klammer dürfen dabei auch verschiedene Datentypen – also Strings, ganze Zahlen oder Fließkommazahlen – stehen. Probieren Sie diese Methode des Referenzierens an dieser Stelle gerne aus, wenn Sie sich bei ihrer genauen Verwendung nicht sicher sind!

7

Wie Sie bereits anhand des obigen einfachen Beispiels sehen, reduziert `else` im Vergleich zum `not`-Operator sowohl den benötigten Code als auch den Denkaufwand um einiges. `else` ermöglicht zudem eine weitere Vereinfachung: sogenannte *bedingte Ausdrücke* (engl. *conditional expressions*). Stellen Sie sich hierzu vor, dass eine Variable namens `strom` die Werte `an` oder `aus` annehmen kann, je nachdem, ob ein Sensor ein elektrisches Signal feststellt (1) oder nicht (0). Mit der bisher gelernten Standardverwendung von `else` würden die Fallunterscheidungen des Codes dementsprechend folgendermaßen aussehen:

```
1 x = int(input("0 oder 1? Signal: "))
2 if x == 1:
3     strom = "an"
4 else:
5     strom = "aus"
6 print("Der Strom ist", strom)
```

Mittels eines bedingten Ausdrucks ist es jedoch möglich, diese Art von Formulierungen auf die folgende Form zu reduzieren:

## 7 Entscheidungen treffen – Programmabläufe steuern

```

1 x = int(input("0 oder 1? Signal: "))
2 strom = ("an" if x == 1 else "aus")
3 print("Der Strom ist", strom)

```

Die zweite Zeile hat also die allgemeine Form `X if Bedingung else Y`. Wann immer eine Fallunterscheidung dieser logischen Form folgt, können Sie die verkürzte Schreibweise mit bedingten Ausdrücken für `if-else`-Klauseln verwenden.

### 7.4.1 elif

Je komplexer ein Programm wird und je mehr Fälle Sie darin abdecken möchten, umso mehr Verzweigungen werden Sie hierfür benötigen. Wenn Sie für jede Fallunterscheidung einen eigenen `if`-Pfad oder mehrstufige Verschachtelungen mit `if-else`-Klauseln verwenden, erhöht dies die Komplexität und Fehleranfälligkeit des Codes. Dabei kann es etwa leicht passieren, dass Sie versehentlich Fälle vergessen, falsche Einrückungen vornehmen und am Ende gar den Überblick über den eigenen Code verlieren. Mehr Sicherheit bietet hier die `elif`-Anweisung. `elif` steht für `else: if` (engl. für *ansonsten, wenn*).

Mittels `elif` lassen sich direkt unter die erste `if`-Anweisung für jeden weiteren Fall beliebig viele Bedingungen mit Anweisungen anführen. Diese müssen nicht eingerrückt werden, wodurch der Code einfach und gut lesbar bleibt. Eine `elif`-Anweisung wird dabei genau dann ausgeführt, wenn alle vorigen Bedingungen nicht zutrafen, die aktuelle jedoch schon. Dies erhöht die Effizienz des Programms, da dieses nach Nichterfüllung der anfänglichen `if`-Bedingung nur solange die `elif`-Klauseln durchläuft, bis eine ihrer Bedingungen zutrifft. Würde man anstatt von `elif`-Klauseln weitere `if`-Bedingungen anführen, so würde der Python-Interpreter diese in jedem Fall einzeln auswerten.

Mit `elif` hat der Code samt aller möglichen Verzweigungen die folgende Form:

```

1 if Bedingung_1:
2     Anweisungsblock_1
3 elif Bedingung_2:
4     Anweisungsblock_2
5     ...
6 elif Bedingung_N:
7     Anweisungsblock_N
8 else:          # optional
9     Anweisungsblock

```

Ganz am Ende kann hier ein `else`-Block folgen, der alle bislang nicht angesprochenen Fälle umfasst. Das Hinzufügen von `else` ist jedoch nicht notwendig.

Wir möchten nun das Schwimmbeispiel aus Abschnitt 7.3. mit `elif`-Klauseln umformulieren:

## 7.4 else und elif: weitere Alternativen hinzufügen

### Codebeispiel #6

```

1 y = int(input("Was ist die aktuelle Wassertemperatur? "))
2 if (y <= 13) or (y > 39):
3     print("Schwimmen nicht ratsam!")
4 elif y > 24:
5     print("Bestes Badewetter! Auf ins Wasser!")
6 elif y > 19:
7     print("Nicht schlecht! Ein erfrischendes Bad wartet auf Sie!")
8 elif y > 13:
9     print("Nur für Wagemutige! Bleiben Sie in Bewegung!")

```

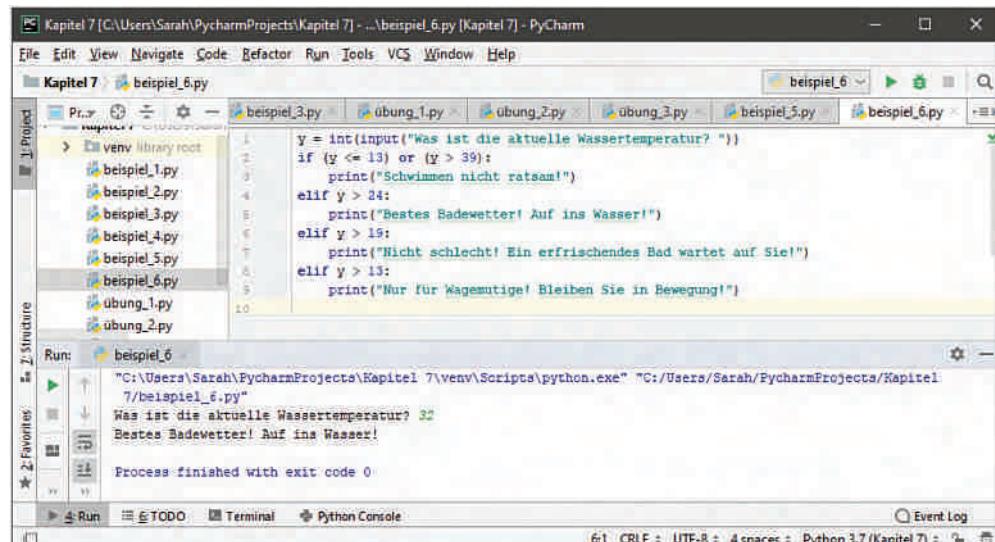


Abb. 7.6 Die Verwendung der `elif`-Anweisung

Wie Sie sehen, lassen sich die anzugebenden Bedingungen durch die Verwendung von `elif` bereits in diesem einfachen Beispiel stark vereinfachen.

Da die erste `if`-Bedingung bereits den Temperaturbereich unterhalb von 13 und oberhalb von 39 °C abdeckt, muss durch die folgenden `elif`-Klauseln nur noch der Bereich zwischen 13 und 39 °C behandelt werden. Dies geschieht mit drei verschiedenen `elif`-Pfaden: Der erste Fall fängt bei über 24 °C an und reicht aufgrund der vorigen `if`-Bedingung automatisch bis zum Endpunkt von 39 °C, der daher nicht explizit angegeben werden muss. Der zweite Bereich liegt dementsprechend oberhalb von 19 bis genau 24 °C, der dritte zwischen oberhalb von 13 und bis genau 19 °C.

Bei zahlenmäßigen Unterscheidungen ist es aufgrund der Logik der `elif`-Klausel also immer eine gute Idee, durch die erste `if`-Bedingung zunächst den größtmöglichen Bereich abzudecken, sodass sich die folgenden `elif`-Bedingungen lediglich auf kleine Zahlenbereiche beziehen und somit leichter zu formulieren sind.

## 7 Entscheidungen treffen – Programmabläufe steuern

### 7.5 Übung: Eigene Abfragen erstellen

Die Logik der `elif`-Klausel sollte nun ersichtlich geworden sein. In den folgenden Aufgaben werden Sie die Möglichkeit haben, weitere Anwendungsfälle zu üben. Denken Sie daran: Oftmals gibt es mehr als eine richtige Möglichkeit, den Code zu formulieren. Wenn Ihre Lösung fehlerfrei ist und für jeden Fall die gewünschte Ausgabe erzeugt, so ist diese Vorgehensweise ebenfalls richtig, auch wenn sie von den hier vorgeschlagenen Lösungswegen abweicht.

#### Übungsaufgaben

1. Kehren wir nochmals zu unserem Supermarkt-Beispiel aus Abschnitt 7.2. zurück: Ändern Sie das Bestandsprogramm mittels `if`- und `elif`-Klauseln dahingehend, dass es je nach Nutzereingabe vier verschiedene Kapazitätsmeldungen für unterschiedliche Bestandsanzahlen ausgeben kann. Verwenden Sie ebenfalls eine `else`-Anweisung für ungültige Eingaben.
2. Erstellen Sie ein Wörterbuch (Datenstruktur: *Dictionary*) namens `Farben`, das als Schlüssel-Wert-Paare die Farben Schwarz, Blau, Grün und Orange mit ihren entsprechenden englischen Übersetzungen enthält. Schreiben Sie nun ein Programm, das die Nutzerinnen zunächst dazu auffordert, eine Farbe einzugeben, deren englische Übersetzung daraufhin angezeigt wird. Das Programm soll sowohl Groß- als auch Kleinbuchstaben als dasselbe Wort erkennen können (Tipp: `lower`-Funktion). Sehen Sie dabei auch einen Fall für ungültige Eingaben vor.
3. Legen Sie eine geeignete Datenstruktur für die acht Planeten unseres Sonnensystems an. Die Nutzer sollen dazu aufgefordert werden, zwei Planeten des Sonnensystems einzugeben. Jeder richtigen und jeder falschen ersten und zweiten Nutzereingabe soll eine entsprechende Ausgabemeldung folgen. Bauen Sie innerhalb der Auswertung der zweiten Eingabe eine weitere Verschachtelung ein, die überprüft, ob zweimal derselbe Planetenname eingegeben wurde und weisen Sie jeden Fall in einer entsprechenden Ausgabe aus.

## 7.5 Übung: Eigene Abfragen erstellen

## Lösungen

1.

```

1 x = int(input("Anzahl der Packungen Spaghetti im Vorrat: "))
2 if x > 200:
3     print("Achtung: Lager voll!")
4 elif x > 50:
5     print("Aktuell sind", x, "Artikel im Lager vorhanden.")
6 elif x > 0:
7     print("Es sind nur noch", x, "Artikel vorrätig. Bitte nachbestellen!")
8 elif x == 0:
9     print("Warnung: Artikel ausverkauft!")
10 else:
11     print("Ungültige Eingabe!") # gilt nur für negative Zahlen, da eine
12                         # ganze positive Zahl erwartet wird

```

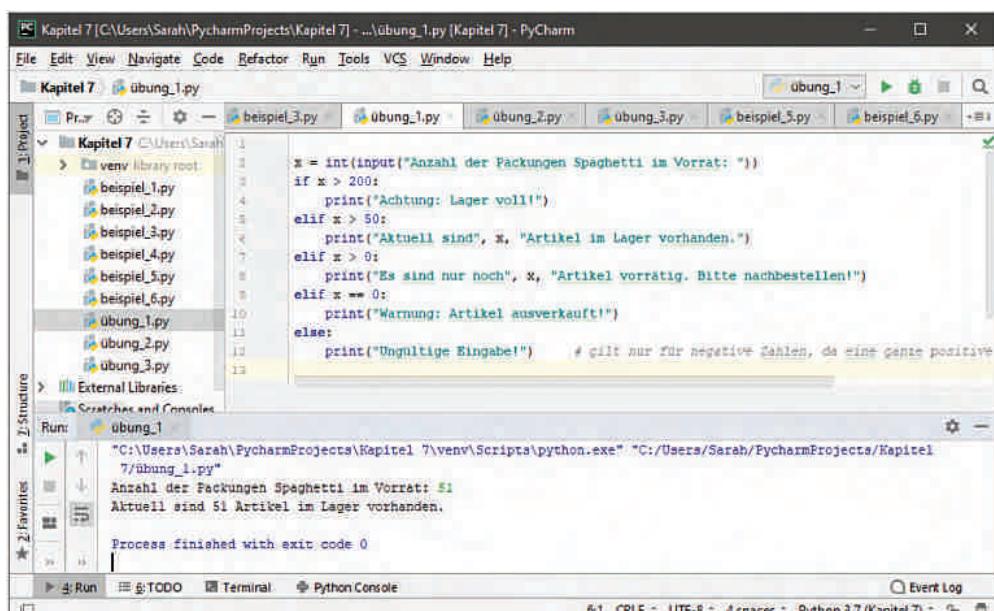


Abb. 7.7 Abfrage des Supermarktbestands mit verschiedenen Kapazitätsausgaben

2.

```

1 farben = {"schwarz": "black", "blau": "blue", "grün": "green", "orange":
2 "orange"}
3 x = input("Geben Sie eine Farbe ein, um die englische Übersetzung
4 herauszufinden! Farbe: ")
5 x = x.lower()
6
7 if x in farben:
8     print("Die englische Übersetzung für", x, "lautet:", farben[x])
9 else:
10    print("Ungültige Eingabe!")

```

## 7 Entscheidungen treffen – Programmabläufe steuern

```

Kapitel 7 [C:\Users\Sarah\PycharmProjects\Kapitel 7] - übung_2.py [Kapitel 7] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 7 übung_2.py
Project J-Project
  Kapitel 7 [C:\Users\Sarah\PycharmProjects\Kapitel 7]
    + venv library root
      beispiel_1.py
      beispiel_2.py
      beispiel_3.py
      beispiel_4.py
      beispiel_5.py
      beispiel_6.py
      übung_1.py
      übung_2.py
      übung_3.py
      beispiel_5.py
      beispiel_6.py
Run: 2: Studiere
  Run: übung_2
    "C:\Users\Sarah\PycharmProjects\Kapitel 7\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 7/übung_2.py"
    Geben Sie eine Farbe ein, um die englische Übersetzung herauszufinden! Farbe: schwarz
    Die englische Übersetzung für schwarz lautet: black
    Process finished with exit code 0
Run TODO Terminal Python Console Event Log
6.1 CRLF : UTF-8 : 4 spaces : Python 3.7 (Kapitel 7) :

```

Abb. 7.8 Das Prinzip eines Wörterbuchs

3.

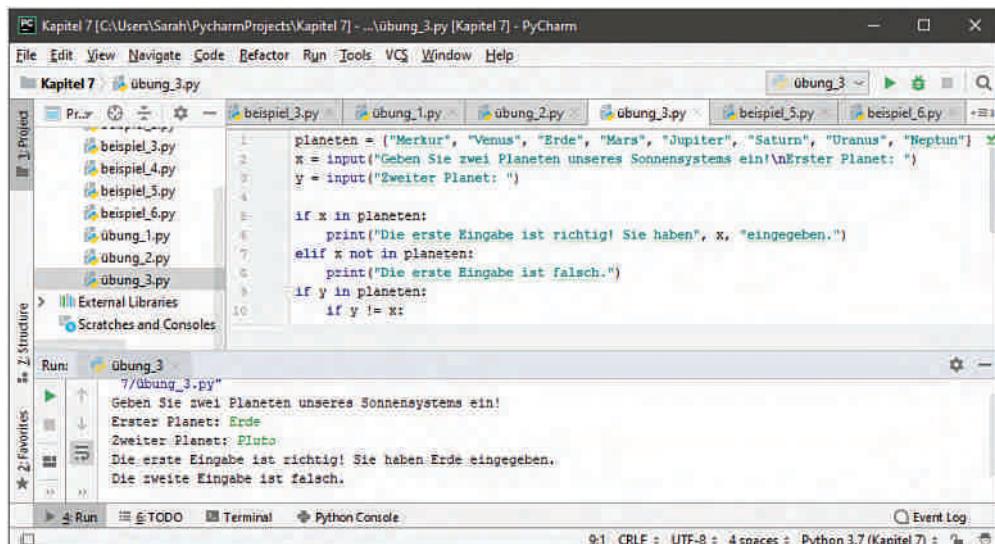
```

1 planeten = {"Merkur", "Venus", "Erde", "Mars", "Jupiter", "Saturn", "Uranus",
2 "Neptun"}
3 x = input("Geben Sie zwei Planeten unseres Sonnensystems ein!\nErster Planet:
4 ")
5 y = input("Zweiter Planet: ")

6
7 if x in planeten:
8     print("Die erste Eingabe ist richtig! Sie haben", x, "eingegeben.")
9 elif x not in planeten:
10    print("Die erste Eingabe ist falsch.")
11 if y in planeten:
12    if y != x:
13        print("Die zweite Eingabe ist richtig! Sie haben", y, "eingegeben.")
14    else:
15        print("Sie haben zweimal denselben Planeten eingegeben!")
16 else:
17    print("Die zweite Eingabe ist falsch.")

```

## 7.5 Übung: Eigene Abfragen erstellen



The screenshot shows the PyCharm IDE interface with the following details:

- Project:** Kapitel 7 [C:\Users\Sarah\PycharmProjects\Kapitel 7] - ...\\Übung\_3.py [Kapitel 7] - PyCharm
- Code Editor:** The file `Übung_3.py` contains the following Python code:

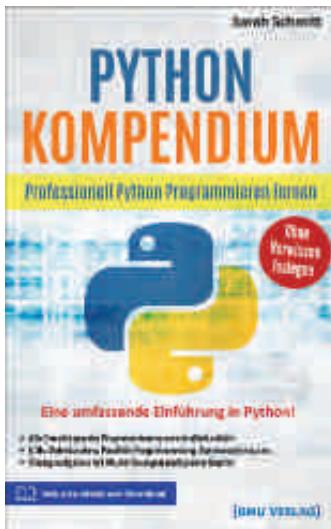
```
1 planeten = ["Merkur", "Venus", "Erde", "Mars", "Jupiter", "Saturn", "Uranus", "Neptun"]
2 x = input("Geben Sie zwei Planeten unseres Sonnensystems ein!\nErster Planet: ")
3 y = input("Zweiter Planet: ")
4
5 if x in planeten:
6     print("Die erste Eingabe ist richtig! Sie haben", x, "eingegeben.")
7 elif x not in planeten:
8     print("Die erste Eingabe ist falsch!")
9 if y in planeten:
10     if y != x:
11         print("Die zweite Eingabe ist richtig! Sie haben", y, "eingegeben.")
12     else:
13         print("Die zweite Eingabe ist falsch!")
```
- Run Tab:** Shows the output of running the script:

```
Geben Sie zwei Planeten unseres Sonnensystems ein!
Erster Planet: Erde
Zweiter Planet: Pluto
Die erste Eingabe ist richtig! Sie haben Erde eingegeben.
Die zweite Eingabe ist falsch.
```
- Status Bar:** Shows the file path as `9:1 CRLF : UTF-8 : 4 spaces : Python 3.7 (Kapitel 7) : 7m`.

Abb. 7.9 Das Programm erkennt Pluto nicht mehr als Planeten an

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

# Kapitel 8

## Schleifen: Programmteile wiederholen

In unseren bisherigen Programmen wurde jeder Befehl jeweils einmal ausgeführt. Manchmal möchte man jedoch bestimmte Schritte mehrmals wiederholen – genauso wie Sie im wirklichen Leben einen Salto wieder und wieder üben würden, bis Sie den Sprung zufriedenstellend beherrschen. Dabei wäre es etwa denkbar, dass Sie einfach einen Salto nach dem anderen machen, bis Sie die Bewegung richtig beherrschen. Alternativ dazu könnten Sie sich anfangs vornehmen, 50 Saltos zu machen, um den Sprung wie erwartet zu können.

Python bietet ebenfalls Werkzeuge an, die diese beiden Wiederholungsarten innerhalb eines Programms realisieren, ohne dass dafür jeder Wiederholungsschritt mühsam einzeln eingetippt werden muss: *Schleifen*. Wie der Name erahnen lässt, sind Schleifen zirkuläre Ausführungsfolgen, innerhalb derer spezifische Aufgaben automatisch wiederholt werden, bis ein vorgegebenes Ereignis oder ein bestimmter Zustand erreicht ist. Wie die im vorigen Kapitel behandelten Entscheidungsvarianten `if`, `else` und `elif` handelt es sich auch bei Schleifen um eine Kontrollstruktur, die ein wesentlicher Bestandteil der meisten Programme ist.

In Python drückt die sogenannte `while`-Schleife den ersten Fall unseres Saltobeispiels aus, der bis zum Erreichen eines bestimmten Ereignisses läuft, während die `for`-Schleife den zweiten Fall mit einer vordefinierten Anzahl an Wiederholungen beschreibt. Wie bei den Entscheidungsanweisungen `if`, `else` und `elif` sind auch bei Schleifen komplexere Verschachtelungen möglich. In diesem Kapitel werden wir Ihnen die unterschiedlichen Gestaltungsmöglichkeiten für Schleifen vorstellen.

### 8.1 Die `while`-Schleife: bedingte Wiederholungen

Ist nicht bekannt, aus wie vielen Daten eine Aufgabe besteht und wie oft eine Schleife wiederholt werden muss, verwendet man hierfür die Schleife `while` (engl. für *solang*). Diese Schleife wird solange ausgeführt, bis eine vorgegebene Bedingung nicht mehr erfüllt ist.

Die simple Struktur der `while`-Abfrage mag dabei sehr an diejenige der `if`-Abfrage erinnern. Im Unterschied zu `if` wird die Bedingung bei `while` jedoch nicht nur einmal abgefragt, sondern wiederholt durchlaufen. Ist das Ende des Anweisungsblocks erreicht, springt die `while`-Schleife wieder zur Bedingung am Anfang zurück und überprüft erneut, ob diese erfüllt ist. Ist dies der Fall, wird die Schleife erneut durchlaufen.

## 8 Schleifen: Programmteile wiederholen

Der Gebrauch von `while` kann auch *Endlosschleifen* erzeugen. Eine Endlosschleife entsteht dann, wenn die Bedingung fortwährend erfüllt bleibt, sodass die Schleife immer wieder wiederholt wird. Betrachten Sie hierfür als Analogie den Mythos des Sisyphos in der Unterwelt: Sisyphos wurde von den Göttern dazu verdammt, bis in alle Ewigkeit einen immensen Felsblock einen Berg hinauf zu rollen. Kurz vor Erreichen des Gipfels rollt der Felsen jedes Mal wieder hinab ins Tal. Aus programmiertechnischer Perspektive befindet sich Sisyphos in einer Endlosschleife, da er unablässig die unerfüllbare Anweisung befolgt, den Felsen solange den Berg hinauf zu wälzen, bis er dessen Gipfel erreicht hat. Die entsprechende `while`-Anweisung des Sisyphos-Algorithmus könnte man demnach folgendermaßen formulieren:

```
1 while (Felsblock nicht auf dem Gipfel):
2     weiterrollen
```

Wie Sie sehen, wird diese absurde Schleife ewig laufen. Das Sisyphos-Programm wird dabei Ihr System verlangsamen, unnötig Kapazitäten beanspruchen und im schlimmsten Fall sogar hängen bleiben. Um das Programm zu beenden, gibt es zum Schluss möglicherweise sogar nur noch eine Lösung: den Rechner auszuschalten.

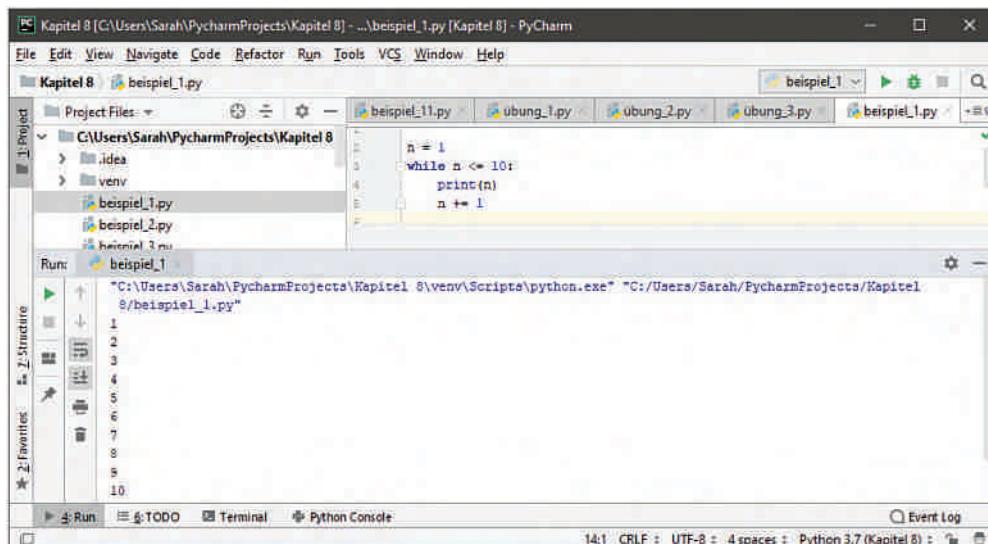
Versuchen Sie daher unbedingt, das Aufkommen derartiger Endlosschleifen in Ihrem Programm zu verhindern, indem Sie sich anfangs Gedanken über die Erfüllbarkeit der angegebenen Bedingungen machen und dementsprechend geschickte Formulierungen aufstellen, damit die Schleife in jedem Fall das Ende erreichen kann. Die Bedingungen sollten dabei so einfach wie möglich gehalten sein. Hier ist es beispielsweise sinnvoll, mindestens eine Variable, die in der Bedingung im Schleifenkopf vorkommt, auch im Anweisungsblock des Schleifenkörpers zu verwenden und in jedem Wiederholungsschritt dahingehend zu verändern, dass sie die Bedingung irgendwann nicht mehr erfüllt. Der Wert dieser Variablen muss also so gewählt werden, dass die Bedingung zu einem bestimmten Zeitpunkt garantiert nicht mehr erfüllt ist.

Lassen Sie uns hierzu nun ein praktisches Beispiel betrachten:

### Codebeispiel #1

```
1 n = 1
2 while n <= 10:
3     print(n)
4     n += 1
```

## 8.1 Die while-Schleife: bedingte Wiederholungen



**Abb. 8.1** Ein einfaches Beispielprogramm für while

Hier wird zunächst der Anfangswert 1 für die Variable `n` definiert. Solange `n` kleiner oder gleich 10 ist, gibt das Programm diesen Wert aus. Im nächsten Anweisungsschritt wird der aktuelle `n`-Wert sodann um 1 erhöht: `n += 1` ist dabei die abgekürzte Schreibweise für `n = n + 1` (Kapitel 5.2.). Die Schleife wird erneut durchlaufen, solange die Bedingung, dass  $n \leq 10$  ist, zutrifft. In diesem Beispiel vermeiden wir eine Endlosschleife dadurch, dass `n` durch die schrittweise Erhöhung irgendwann den Wert 11 erreicht, für den die while-Schleife nicht mehr gilt. Probieren Sie einmal aus, was geschieht, wenn Sie die letzte Zeile im Programm entfernen. Das Programm bleibt nun in der Schleife stecken und gibt fortwährend die Zahl 1 aus. Sie können das Programm anhalten, indem Sie in PyCharm im rechten oberen Fensterbereich auf das rote Stopp-Symbol links neben der Lupe klicken.

8

while-Schleifen lassen sich ähnlich wie if-Abfragen durch eine else-Klausel erweitern. Die else-Verzweigung wird dabei genau dann einmal durchlaufen, sobald die Bedingung der while-Schleife nicht mehr erfüllt ist. Das obige Beispiel könnte etwa folgendermaßen durch eine else-Klausel beendet werden:

```
1 n = 1
2 while n <= 10:
3     print(n)
4     n += 1
5 else:
6     print("Das waren die Zahlen von 1 bis 10!")
```

Bei numerischen Programmen wie diesem endet die Schleife durch das Erreichen einer bestimmten Zahl. while-Schleifen können jedoch auch durch eine Nutzereingabe beendet werden, wie das folgende Beispiel veranschaulicht:

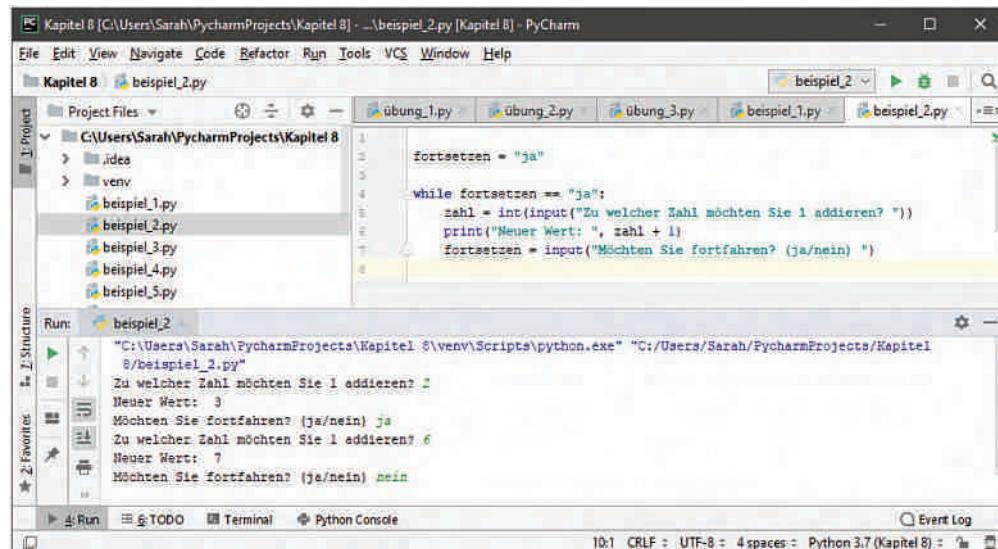
## 8 Schleifen: Programmteile wiederholen

### Codebeispiel #2

```

1  fortsetzen = "ja"
2  while fortsetzen == "ja":
3      zahl = int(input("Zu welcher Zahl möchten Sie 1 addieren? "))
4      print("Neuer Wert: ", zahl + 1)
5      fortsetzen = input("Möchten Sie fortfahren? (ja/nein) ")

```



**Abb. 8.2** Ein Beispielprogramm für while mit Nutzereingabe

Hier wird die Variable `fortsetzen` zu Anfang als String mit dem Inhalt `ja` definiert. Innerhalb der Schleife werden die Nutzerinnen dazu aufgefordert, eine Zahl einzugeben. Das Programm bildet sodann die Summe aus dieser Zahl + 1 und gibt den neuen Wert aus. Anschließend fragt das Programm die Nutzer, ob sie fortfahren möchten. Geben sie `ja` ein, ist die Bedingung der Schleife erfüllt und die Schleife wird wieder ausgeführt. Bei einer anderen Eingabe als `ja` wird die Schleife beendet. Da die Nutzereingabe die Dauer der Schleife bestimmt, kann in diesem Fall keine Endlosschleife entstehen.

## 8.2 Die for-Schleife: elementweises Iterieren

Bei der `for`-Schleife (engl. für *für*) handelt es sich um einen Schleifentyp, ohne den kaum eine Programmiersprache auskommt. Wie bereits erwähnt, ist bei der Verwendung von `for` im Programm vorab festgelegt, wie oft die Schleife ausgeführt wird. Die Anzahl der gewünschten Wiederholungen muss Ihnen also von vornherein bekannt sein, wenn Sie die `for`-Schleife verwenden möchten. Meist ist diese Anzahl durch die Elemente innerhalb eines Datentyps oder einer Datenstruktur gegeben, die die `for`-Schleife einzeln durchläuft – man nennt dies auch *iterieren*.

## 8.2 Die for-Schleife: elementweises Iterieren

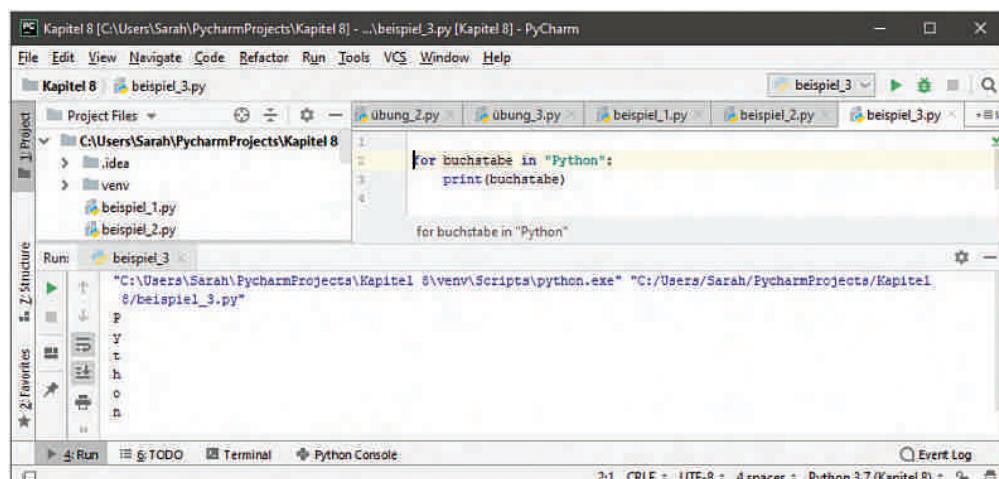
Im Gegensatz zur while-Schleife ist for also nicht an eine bestimmte Bedingung geknüpft, sondern geht sequentiell vor und durchläuft die Bestandteile eines Datentyps oder einer Datenstruktur. Sie hat die folgende allgemeine Form:

```
1 for Variable in Elemente:
2     Anweisungsblock
```

Dazu gleich ein einfaches Beispiel:

### Codebeispiel #3

```
1 for buchstabe in "Python":
2     print(buchstabe)
```



8

**Abb. 8.3** Ein einfaches Beispiel einer for-Schleife

Die Variable `buchstabe` durchläuft hier nach und nach alle Buchstaben der Zeichenkette `Python` und gibt diese sodann jeweils einzeln aus. In diesem Fall mag die Variablenbezeichnung etwas verwirrend erscheinen, da die Variable eigentlich jeweils die einzelnen Elemente der Zeichenkette benennt. Im Unterschied zur sonstigen Benennung von Variablen wird die durchlaufende Schleifenvariable einer for-Schleife bei ganzen Zahlen daher oftmals einfach mit `i` (für engl. *integer*) und bei Zeichen mit `c` (für engl. *character*) bezeichnet. Dabei können Sie selbstverständlich auch jeden anderen beliebigen Variablennamen wählen.

Wir möchten nun weitere Verwendungsmöglichkeiten der for-Schleife untersuchen:

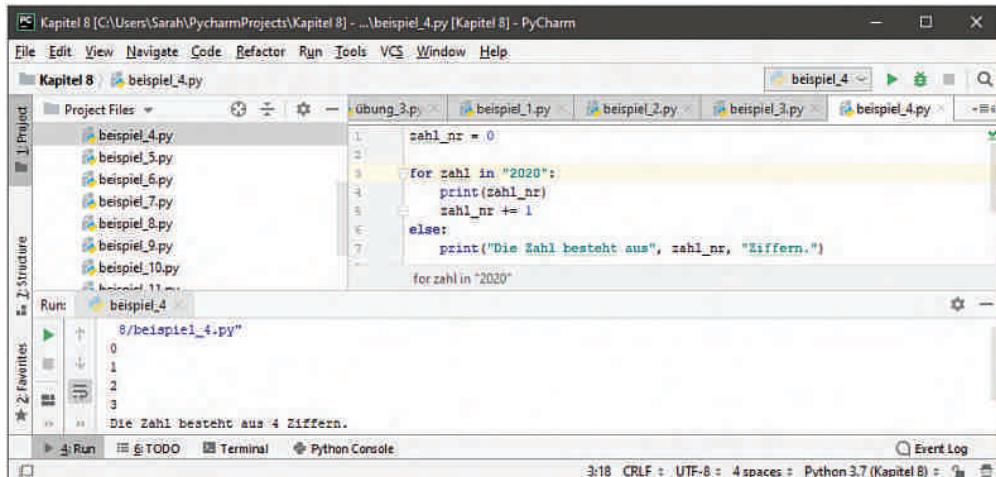
## 8 Schleifen: Programmteile wiederholen

### Codebeispiel #4

```

1 zahl_nr = 0
2 for zahl in "2020":
3     print(zahl_nr)
4     zahl_nr += 1
5 else:
6     print("Die Zahl besteht aus", zahl_nr, "Ziffern.")

```



**Abb. 8.4** Eine weitere Variante der `for`-Schleife

Zahlen-Datentypen müssen also ebenfalls als Strings geschrieben werden, da sie nicht iterierbar sind. Das Programm gibt jede Ziffernposition in 2020 einzeln aus. Gleichzeitig zählt `zahl_nr` beginnend bei 0 die jeweilige Ziffernposition mit und erhöht sich innerhalb des Anweisungsblocks der `for`-Schleife bei jedem Durchlauf einer Ziffer um 1. Nachdem die zweite 0 als letzte Ziffer des Strings an der vierten Stelle durchlaufen wurde, erhält `zahl_nr` den Wert 4. Wie zuvor bei `for`-Schleifen lässt sich nun ebenfalls `else` verwenden: Die `else`-Abfrage wird genau dann ausgeführt, wenn alle Elemente iteriert wurden, und zwar genau einmal. Anhand von `zahl_nr` lässt sich mit `else` nun die Anzahl der Ziffern anzeigen.

Im Zusammenhang mit `for`-Schleifen ist es sinnvoll, gleich eine weitere Funktion in Python kennen zu lernen: `range`. Eine Verwendung von `for` mit `range` könnte etwa folgendermaßen aussehen:

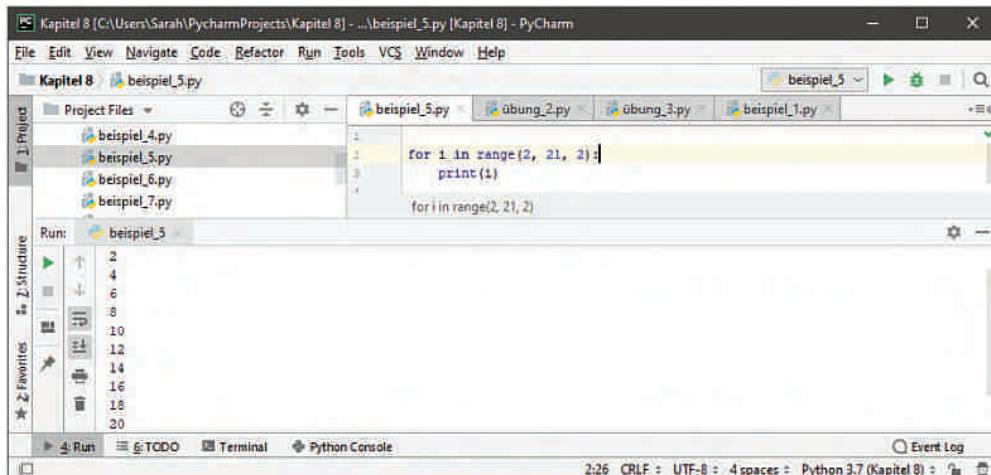
### Codebeispiel #5

```

1 for i in range(2, 21, 2):
2     print(i)

```

### 8.3 break und continue: weitere Werkzeuge für die Steuerung von Schleifen



**Abb. 8.5** Die `for`-Schleife mit `range` zur Ausgabe der Zweierreihe

In diesem Beispiel wird die Zweierreihe ausgegeben. Die Zahlen in der `range`-Klammer bezeichnen jeweils den Start- und Endwert sowie die Schrittweite. Da der angegebene Endwert dabei nicht mitgezählt wird, muss man hier eine 21 anstatt einer 20 eingeben, damit 20 als letzte Zahl in der Reihe ebenfalls ausgegeben wird. Geben Sie in der Klammer von `range` lediglich einen Wert ein, so wird die Zählung ab 0 gestartet und läuft bis zu diesem Wert als Endwert. Werden zwei Werte eingegeben, gelten diese als Start- und Endwert. Die Schrittweite beträgt dann automatisch 1. `range` lässt sich auch mit negativen Schrittweiten verwenden. In diesem Fall wird von einem höheren Startwert um die angegebene Schrittweite bis zum Endwert heruntergezählt.

8

Sie können `range` immer dann verwenden, wenn Sie eine Anweisung eine bestimmte Anzahl an Malen ausführen möchten, beispielsweise auch folgendermaßen:

```

1 for i in range(3):
2     print("Python ist super!")

```

In diesem Fall zählt die Variable beginnend bei 0 die Anzahl der Wiederholungen des `print`-Befehls ab und läuft bis 2 durch. Derartig aufgebaute Schleifen nennt man auch *Zählschleifen*.

### 8.3 break und continue: weitere Werkzeuge für die Steuerung von Schleifen

Sowohl die `for`- als auch die `while`-Schleife lassen sich neben der `else`-Abfrage durch zwei weitere wichtige Zusatzwerkzeuge anpassen und abändern. Diese heißen `break` und `continue`. Mit diesen beiden Befehlen können Schleifen beim Eintreten eines bestimmten Ereignisses auf zwei verschiedene Weisen frühzeitig abgebrochen werden.

## 8 Schleifen: Programmteile wiederholen

### 8.3.1 Ausnahmen mit break handhaben

Die `break`-Anweisung dient dem vorzeitigen Beenden von Schleifen, zum Beispiel auch von Endlosschleifen. Mit `break` wird die Schleife sofort abgebrochen und verlassen, woraufhin das restliche Programm weiter ausgeführt wird. Sie können `break` auch mehrmals innerhalb der Schleife verwenden. Sollten Sie bei Ihrem Programm beispielsweise Bedenken haben, dass unter bestimmten Voraussetzungen eine Endlosschleife entstehen kann, für die sich keine Abbruchbedingung formulieren lässt, so ist die Verwendung von `break` in diesem Fall sicherlich sinnvoll.

Wir möchten zunächst eine Einsatzmöglichkeit von `break` in `while`-Endlosschleifen untersuchen. Eine häufig gesehene Methode, mit `while` eine Endlosschleife zu formulieren, lässt sich in Verbindung mit dem booleschen Datentyp `True` verwirklichen. In diesem Fall ist die Bedingungsklausel `while True:` von vorneherein jederzeit erfüllt und hat daher die Form einer Endlosschleife. Um sicher zu stellen, dass die `while`-Schleife an einem bestimmten Punkt stoppt, gebraucht man innerhalb der Schleife `break`:

#### Codebeispiel #6

```

1  zahl = 7
2  while True:
3      zahl = int(input("Raten Sie! Geben Sie eine Zahl zwischen 0 und
4          10 ein:"))
5      if zahl == 7:
6          break
7      else:
8          print("Ihre Eingabe ist leider falsch. Versuchen Sie es erneut!")
9  print("Glückwunsch! Sie haben die richtige Zahl erraten!")

```

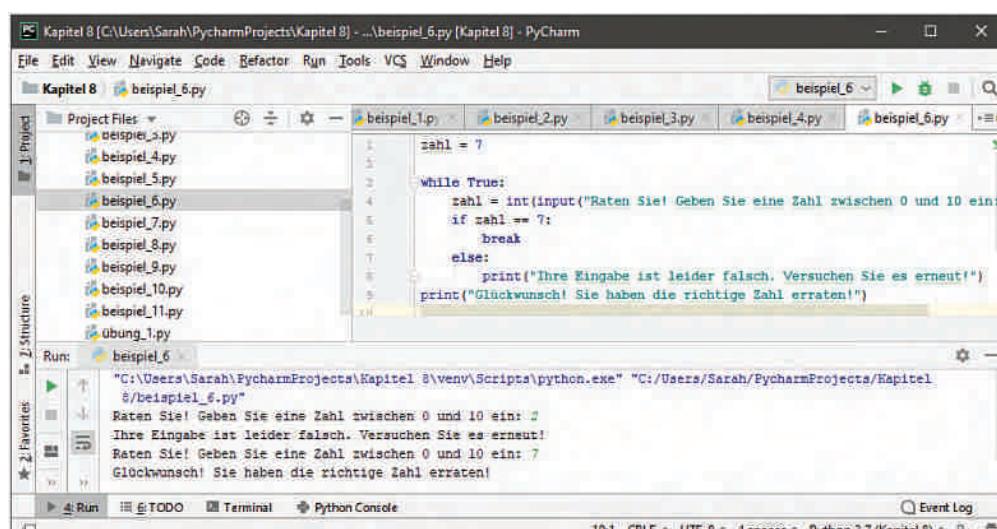


Abb. 8.6 Erraten einer Zahl mit `while` und `break`

### 8.3 break und continue: weitere Werkzeuge für die Steuerung von Schleifen

Es mag anfangs nicht sehr sinnvoll erscheinen, Konstruktionen mit `while True:` zu verwenden. Tatsächlich wird diese Formulierungsweise jedoch recht häufig verwendet, beispielsweise in Programmen, die jederzeit eine Benutzereingabe entgegennehmen können sollen.

Wir möchten nun ein raffiniertes Beispiel mit `break` und `for` betrachten, in dem mehrere uns mittlerweile bekannte Konzepte vereint sind. Die Aufgabe besteht darin, die erste fünfstellige Zahl ausgeben zu lassen, die sich durch 77 teilen lässt:

#### Codebeispiel #7

```
1 for i in range(0, 1000000, 77):
2     if len(str(i)) == 5:
3         print(i)
4         break
```

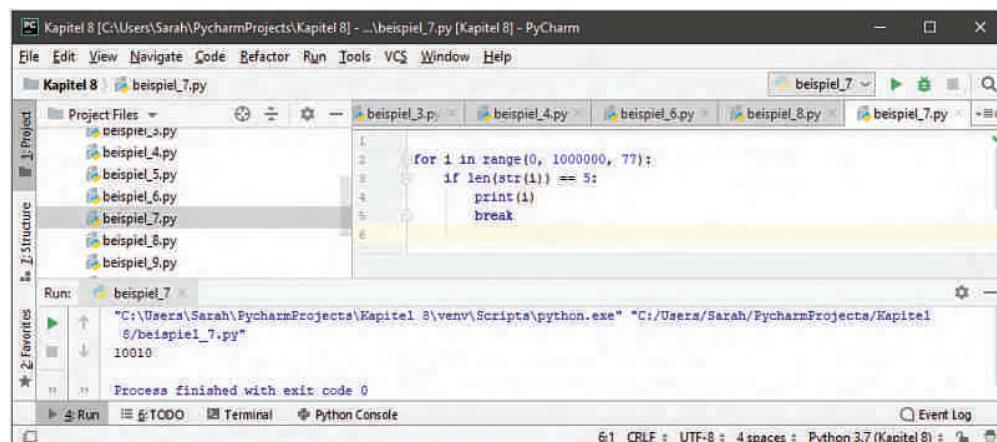


Abb. 8.7 Ausgabe einer gewünschten Zahl mit `for` und `break`

Wir verwenden zur Lösung ganz einfach die `range`-Funktion mit dem Startwert 0, einem ausreichend hohen Endwert und einer Schrittweite von 77. Damit durchläuft die `for`-Schleife automatisch alle Vielfachen von 77. In der untergeordneten `if`-Klausel werden die Vielfachen von 77 eines nach dem anderen mittels der `len`-Funktion (Kapitel 6.6.) dahingehend überprüft, ob sie von der Länge 5 sind. Dafür müssen wir die Zahlenwerte zunächst in einen String-Datentyp umwandeln, da sich `len` nicht auf Zahlen anwenden lässt. Sobald das erste 5-stellige Vielfache von 77 erreicht wurde, wird dieses durch den `print`-Befehl ausgegeben. Um zu verhindern, dass weitere 5-stellige Vielfache angezeigt werden, wird die Schleife im Anschluss sogleich durch `break` unterbrochen. Durch `break` wird hier also die gesamte `for`-Schleife und nicht nur die aktuelle `if`-Anweisung verlassen.

## 8 Schleifen: Programmteile wiederholen

### 8.3.2 Das Schleifenende mit continue überspringen

Mit dem Schlüsselwort `continue` können Sie innerhalb einer Schleife sofort den restlichen Quellcode des aktuellen Schleifendurchlaufs überspringen und wieder zum Anfang der Schleife zurückkehren, um diese erneut auszuführen.

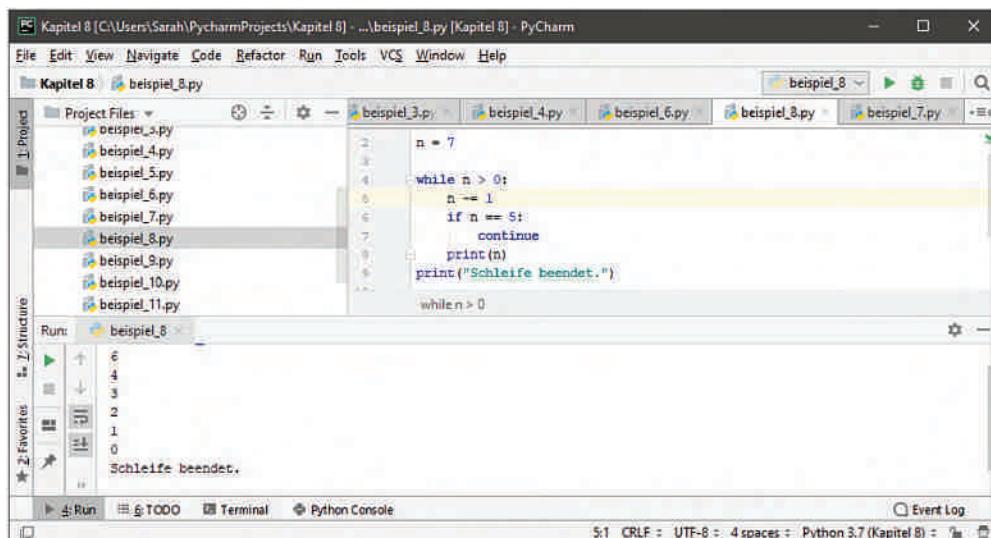
Dazu ein Beispiel:

#### Codebeispiel #8

```

1 n = 7
2 while n > 0:
3     n -= 1
4     if n == 5:
5         continue
6     print(n)
7 print("Schleife beendet.")

```



**Abb. 8.8** Ausgabe einer Zahlenreihe mit `while` und `continue`

Anfangend bei 7 wird die Variable `n` innerhalb der `while`-Schleife schrittweise um 1 verringert. Hat `n` den Wert 5 erreicht, gilt der `continue`-Befehl: 5 wird nicht ausgegeben. Der Durchlauf springt stattdessen wieder zum Schleifenkopf, und die Zahlenreihe wird für `n = 4` fortgesetzt, bis `n` den Wert 0 erreicht hat.

Angenommen, Sie möchten sich aus einer bestimmten Zahlenmenge nun lediglich alle geraden Zahlen anzeigen lassen. Hierfür lässt sich die `for`-Schleife in Verbindung mit `if` und `continue` verwenden:

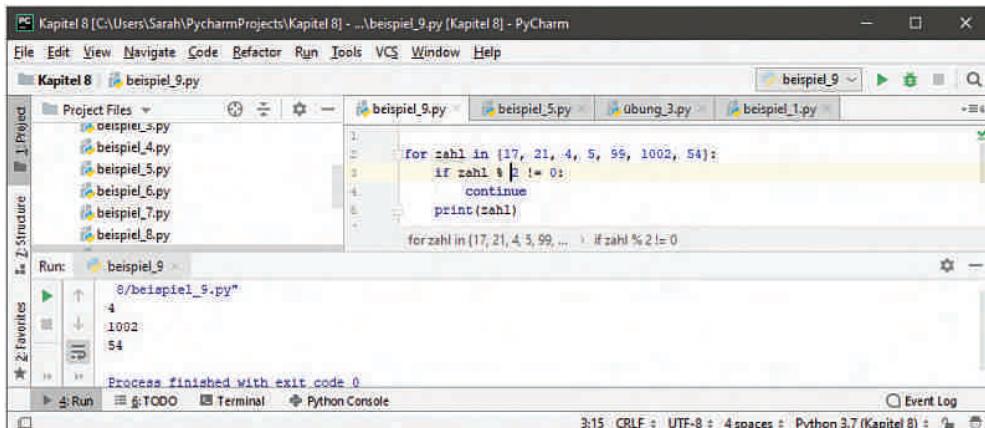
## 8.4 Verschachtelte Schleifen

### Codebeispiel #9

```

1  for zahl in {17, 21, 4, 5, 99, 1002, 54}:
2      if zahl % 2 != 0:
3          continue
4      print(zahl)

```



**Abb. 8.9** Ausgabe einer Zahlenreihe mit `for` und `continue`

Das `%`-Symbol steht hier für die in Kapitel 5.2. behandelte Modulo-Operation. Sobald die Modulo-Rechnung für ein Element nicht Null ergibt, ist `zahl` ungerade, der `print`-Befehl wird übersprungen und das nächste Element in der Menge ausgewertet. Anstatt der Mengen-Datenstruktur können Sie hier übrigens ebenso gut von einer Liste oder einem Tupel Gebrauch machen.

8

Wie Sie eventuell bemerkt haben, lässt sich `continue` oftmals ebenso gut durch `if`, `elif` oder `else` ersetzen, wodurch der Code mitunter zunächst übersichtlicher wirkt. `continue` ist jedoch besonders in längeren Schleifen von Vorteil, in denen bestimmte Fälle komplett übersprungen werden sollen, ohne dass dabei für jede Ausnahme weitere Anweisungen ausformuliert werden müssen, was sich aufgrund der Komplexität vieler Programme schwierig gestalten kann. In diesem Fall lässt sich mit `continue` dadurch Rechenleistung einsparen, dass spezifische Schleifen gar nicht erst durchlaufen werden müssen.

## 8.4 Verschachtelte Schleifen

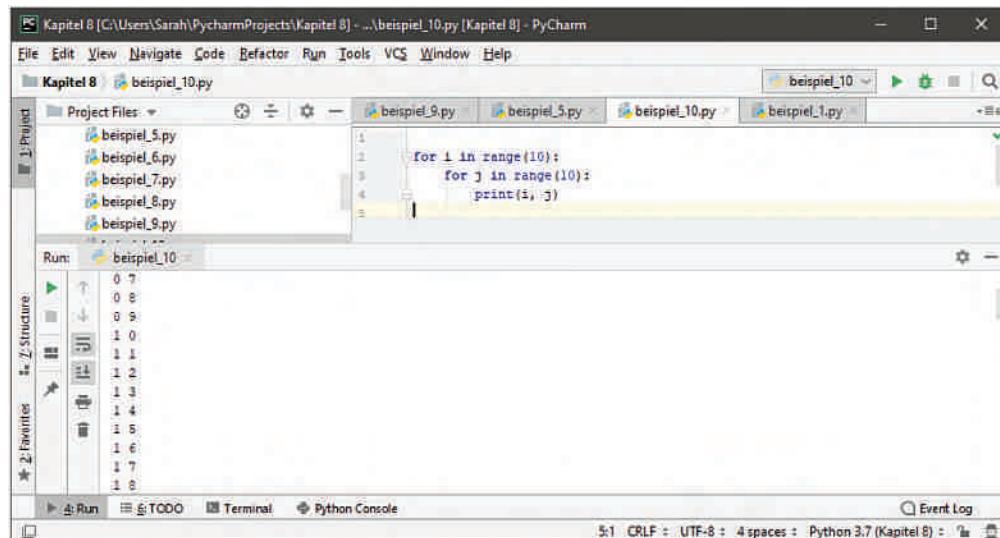
Oftmals ist eine konkrete Aufgabenstellung derart gestellt, dass Sie zu deren Lösung sowohl eine `for`- als auch eine `while`-Schleife oder auch mehrere Schleifen desselben Typs verwenden müssen. Dies ist in Python mit sogenannten *Verschachtelungen* möglich, die oftmals recht komplexe Formen annehmen können.

## 8 Schleifen: Programmteile wiederholen

Angenommen, Sie möchten sich alle zweistelligen Kombinationen der Zahlen von 0 bis 9 anzeigen lassen. Mithilfe der `range`-Funktion lässt sich dies durch die folgende Verschachtelung zweier `for`-Schleifen realisieren:

### Codebeispiel #10

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



**Abb. 8.10** Zwei `for`-Schleifen mit `range` zur Ausgabe von Kombinationsmöglichkeiten

In der übergeordneten `for`-Schleife iteriert die Variable `i` dabei durch die Zahlen von 0 bis 9. Für jede dieser Zahlen wird wiederum die untergeordnete `for`-Schleife ausgeführt, in der die Variable `j` ebenfalls von 0 bis 9 läuft. Sodann gibt das Programm alle möglichen Werte von `i` und `j` aus.

Auch `while`-Schleifen lassen sich ineinander verschachteln. So könnten wir das obige Beispiel stattdessen mit zwei verschachtelten `while`-Schleifen ausdrücken, um uns alle möglichen Kombinationen der Variablen `i` und `j` zwischen 0 und 9 ausgeben zu lassen:

```
1 i = 0
2 while i < 10:
3     j = 0
4     while j < 10:
5         print(i, j)
6         j += 1
7     i += 1
```

## 8.5 Die Platzhalteroption pass

Wie Sie sehen, gestaltet sich der Gebrauch zweier `while`-Schleifen hier etwas komplizierter als mit `for`, obwohl die Ausgabe identisch ist. Zunächst wird die Variable `i` außerhalb der `while`-Schleife auf 0 gesetzt. In der zweiten `while`-Schleife durchläuft `j` sodann alle Werte von 0 bis 9, während `i` weiterhin 0 ist. Die untergeordnete `while`-Schleife gibt sodann alle 0-`j`-Kombinationen aus. Nachdem `j` den Wert 9 erreicht hat, wird die `while`-Schleife verlassen, `i` wird auf 1 erhöht und `j` in der übergeordneten `while`-Schleife erneut auf 0 gesetzt, woraufhin in der zweiten Schleife alle 1-`j`-Kombinationen durchlaufen und ausgegeben werden. Dieser Prozess wird wiederholt, bis sowohl `i` als auch `j` den Wert 9 erreicht haben.

## 8.5 Die Platzhalteroption `pass`

Insbesondere bei komplexeren Programmen mit vielen verschachtelten `if`-, `elif`- und `else`-Bedingungen oder Schleifen kann es eine gute Idee sein, zunächst die grobe Pfadstruktur mitsamt aller Verzweigungen des Programms aufzustellen und im Anschluss die einzelnen Bedingungen und Anweisungsblöcke auszuformulieren. Dies ist beispielsweise auch dann eine sinnvolle Herangehensweise, wenn eine einzelne Bedingung oder Anweisung zu Anfang nicht bekannt ist, man das Programm daran jedoch dennoch bereits ausführen möchte. Eine Lösung dies zu realisieren wäre, den nachträglich auszufüllenden Bestandteil einer `if`-Klausel oder einer Schleife vorläufig zu einem Kommentar zu machen, wodurch dieser innerhalb des Programms ignoriert wird. Auf diese Weise ließe sich zunächst das grobe Gerüst des Programms aufstellen. Beim Durchlaufen würde das Programm jedoch eine Fehlermeldung ausgeben, da die für die Ausführung der Klausel oder Schleife erforderlichen Bestandteile auskommentiert und somit nicht vorhanden sind.

8

Python bietet hier stattdessen die Möglichkeit, ein Schlüsselwort namens `pass` in den Quellcode einzufügen. Mit dem Platzhalter `pass` kann man Stellen im Code überbrücken, die das Programm überspringen soll. Das Schlüsselwort `pass` lässt sich nicht nur für Bedingungen oder Schleifen einsetzen, sondern überall dort im Quellcode, wo man einen Platzhalter benötigt, um das Grundgerüst des Programms entwerfen zu können. Es ist stets ratsam, bei der Verwendung von `pass` einen erklärenden Kommentar vorzusehen, um später besser nachvollziehen zu können, was an der entsprechenden Stelle noch zu tun ist.

Ein einfaches Programm mit `pass` könnte etwa die folgende Gestalt haben:

### Codebeispiel #11

```

1 alter = int(input("Bitte geben Sie Ihr Alter ein! Alter: "))
2 if alter < 18:
3     print("Sie sind nicht zur Wahl zugelassen!")
4 elif alter >= 18:
5     pass # weitere Angaben folgen

```

## 8 Schleifen: Programmteile wiederholen

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_7.py', 'beispiel\_8.py', etc. Below it, the 'Project' view shows a tree structure of files. The main code editor window displays the following Python script:

```
alter = int(input("Bitte geben Sie Ihr Alter ein! Alter: "))

if alter < 18:
    print("Sie sind nicht zur Wahl zugelassen!")
elif alter >= 18:
    pass # weitere Angaben folgen
```

The 'Run' tab in the bottom left shows the command run: "C:\Users\Sarah\PycharmProjects\Kapitel 8\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 8/beispiel\_11.py". The output pane shows the program's interaction with the user:

```
Bitte geben Sie Ihr Alter ein! Alter: 36
```

The status bar at the bottom right indicates the file is 'Python 3.7 (Kapitel 8)'.

Abb. 8.11 Ein Programmaufbau mit pass

## 8.6 Übung: Mit verschiedenen Schleifen arbeiten

### Übungsaufgaben

1. Erstellen Sie eine Liste aus fünf Zahlen, für die Sie sich jeweils das Quadrat ausgeben lassen. Verwenden Sie zur Lösung einmal eine `for`- und einmal eine `while`-Schleife.
2. Erstellen Sie ein Dictionary, das die deutschen und englischen Wörter einiger Farben enthält (vgl. das Beispiel aus 7.5.) Lassen Sie sich alle Schlüssel-Wert-Paare des Dictionaries mithilfe einer passenden Schleife durch einen `print`-Befehl ausgeben.
3. Schreiben Sie ein Programm, das die Nutzerin oder den Nutzer dazu auffordert, ein sechsstelliges Passwort einzugeben. Bei korrekter Passworteingabe soll eine Meldung erfolgen, dass das Passwort gespeichert wurde. Bei einer falschen Eingabe soll eine entsprechende Meldung ausgegeben und die ursprüngliche Eingabeaufforderung wiederholt werden.

## 8 Schleifen: Programmteile wiederholen

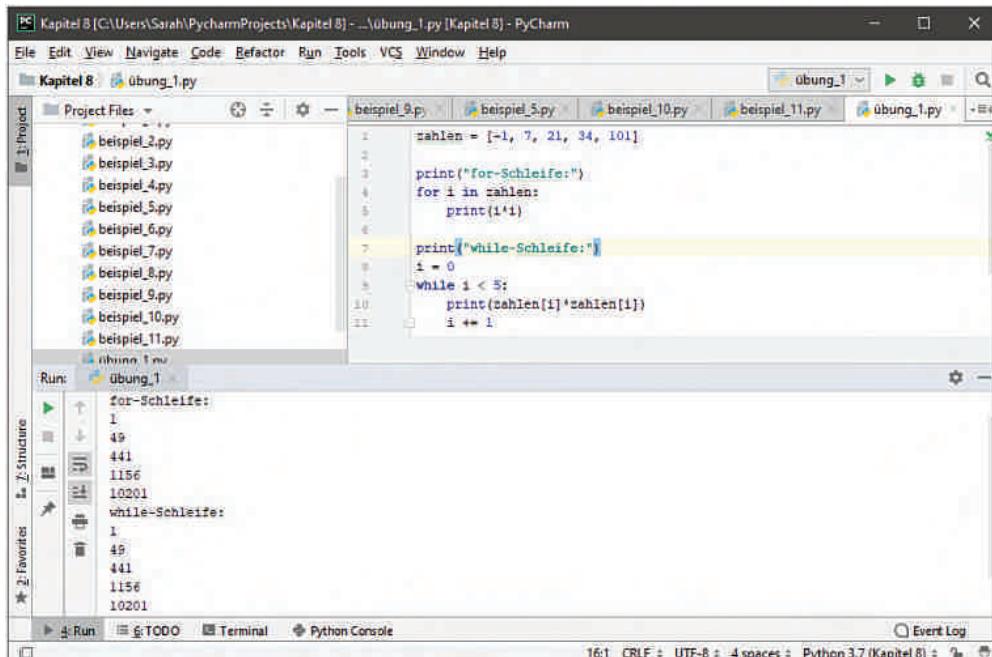
### Lösungen

#### 1. Lösung mit for:

```
1 zahlen = [-1, 7, 21, 34, 101]
2 for i in zahlen:
3     print(i*i)
```

#### Lösung mit while:

```
1 zahlen = [-1, 7, 21, 34, 101]
2 i = 0
3 while i < 5:
4     print(zahlen[i]*zahlen[i])
5     i += 1
```

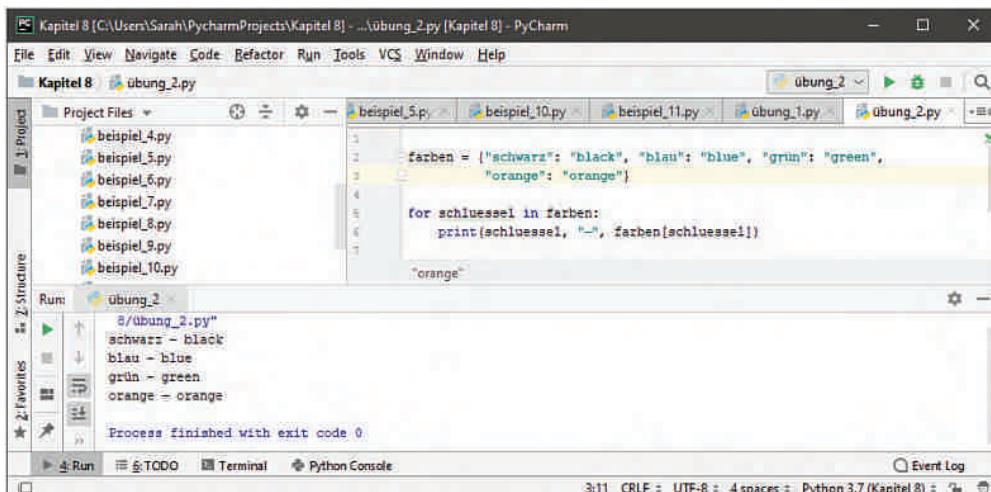


**Abb. 8.12** Zwei Ausgabemöglichkeiten mit for und while

#### 2. In diesem Fall ist die for-Schleife die bessere Wahl:

```
1 farben = {"schwarz": "black", "blau": "blue", "grün": "green", "orange":
2 "orange"}
3 for schluessel in farben:
4     print(schluessel, "-", farben[schluessel])
```

## 8.6 Übung: Mit verschiedenen Schleifen arbeiten



```

Kapitel 8 [C:\Users\Sarah\PycharmProjects\Kapitel 8] - ...\\übung_2.py [Kapitel 8] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 8 übung_2.py
Project Files
  beispiel_4.py
  beispiel_5.py
  beispiel_6.py
  beispiel_7.py
  beispiel_8.py
  beispiel_9.py
  beispiel_10.py
Run: übung_2
  8/Übung_2.py"
  schwarz - black
  blau - blue
  grün - green
  orange - orange
Process finished with exit code 0
Event Log
3:11 CRLF UTF-8 4 spaces Python 3.7 (Kapitel 8)

```

**Abb. 8.13** Ausgabe der Schlüssel-Wert-Paare eines Dictionaries

Wie die Namensgebung der Variablen `schluessel` bereits erraten lässt, nimmt sie lediglich die jeweiligen Schlüsselbegriffe des Dictionaries auf, nicht jedoch die dazugehörigen Werte. Um auf die Werte zuzugreifen, muss man diese durch die `schluessel`-Variable als Index referenzieren.

3.

```

1 while True:
2     passwort = input("Bitte wählen Sie ein sechsstelliges Passwort: ")
3     if len(passwort) == 6:
4         print("Vielen Dank, das Passwort wurde gespeichert!")
5         break
6     else:
7         print("Das Passwort ist ungültig.")

```

8

## 8 Schleifen: Programmteile wiederholen

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_8.py', 'beispiel\_9.py', 'beispiel\_10.py', 'beispiel\_11.py', 'Übung\_1.py', 'Übung\_2.py', and 'Übung\_3.py'. The main code editor window displays the following Python script:

```

passwort = input("Bitte wählen Sie ein sechsstelliges Passwort: ")
if len(passwort) == 6:
    print("Vielen Dank, das Passwort wurde gespeichert!")
    break
else:
    print("Das Passwort ist ungültig.")

```

Below the code editor, the 'Run' tool window shows the execution of 'Übung\_3.py' with the command 'C:\Users\Sarah\PycharmProjects\Kapitel 8\venv\Scripts\python.exe' followed by the file path. The run output pane shows two attempts to enter a password:

```

Bitte wählen Sie ein sechsstelliges Passwort: xyz12
Das Passwort ist ungültig.
Bitte wählen Sie ein sechsstelliges Passwort: xyz123
Vielen Dank, das Passwort wurde gespeichert!

```

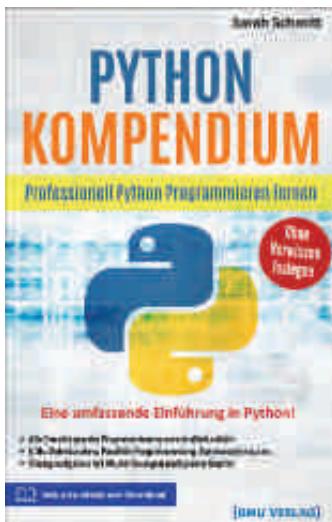
The bottom status bar indicates the file is 'Python 3.7 (Kapitel 8)'.

**Abb. 8.14** Wiederholte Eingabeaufforderung mit while

Durch `while True:` wird die Eingabeaufforderung standardmäßig wiederholt und erst dann unterbrochen, wenn ein sechsstelliges Passwort eingegeben wurde. Die Unterbrechung wird hier durch `break` bewerkstellt. Ist das eingegebene Passwort nicht sechsstellig, wird durch die `else`-Anweisung eine entsprechende Meldung ausgegeben. In diesem Fall wird die `while`-Schleife erneut ausgeführt.

Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

# Kapitel 9

## Funktionen in Python

Sicherlich sind Ihnen Funktionen als wesentliches Konzept aus der Mathematik bekannt: Hier beschreiben Funktionen bestimmte Rechenvorschriften, die jeweils einem Element einer Definitionsmenge genau ein Element einer Zielmenge zuordnen.

In Programmiersprachen dienen Funktionen einem ähnlichen Zweck: Für bestimmte Eingabewerte (sog. *Argumente*) erzeugen sie eine bestimmte Ausgabe, wobei es sich dabei jedoch nicht unbedingt wie in der Mathematik um konkrete Zahlenwerte handeln muss. Auch komplexere Abfolgen von Rechenanweisungen lassen sich mittels Funktionen in Algorithmen zusammenfassen und anschließend an verschiedenen Stellen im Programm anhand des Funktionsnamens erneut aufrufen. Hat man die Funktion anfangs einmal definiert und möchte sie anschließend innerhalb des Programms erneut verwenden, muss hierfür somit nicht der gesamte Codeabschnitt kopiert und an den entsprechenden Stellen im Programm eingefügt werden. Es genügt lediglich das Aufrufen der Funktion durch die Nennung des Funktionsnamens.

Um später Teile einer Funktion anzupassen, ist es ausreichend, Änderungen in der Funktionsdefinition vorzunehmen, die sodann automatisch für alle Aufrufe der Funktion innerhalb des Programms gelten. Funktionen sind somit überaus nützlich: Sie minimieren die Fehleranfälligkeit des Programms, sparen Platz im Quellcode und reduzieren den Arbeitsaufwand beim Programmieren. Zudem erleichtern Funktionen beispielsweise die Zusammenarbeit in Teams, wenn alle Beteiligten zunächst jeweils separate Funktionen bearbeiten und entwickeln können, die im Anschluss zusammengefügt werden. Das Ergebnis ist ein übersichtlicher, leicht nachvollziehbarer Quellcode.

Bevor wir genauer untersuchen, wie man in Python eigene Funktionen schreibt, können Sie sich zunächst unter [www.youtube.com/watch?v=ohDB5gbtaEQ](https://www.youtube.com/watch?v=ohDB5gbtaEQ) vom Monty-Python-Sketch „Argument Clinic“ inspirieren lassen, der eine andere Bedeutung von *Argument* im Englischen veranschaulicht.

### 9.1 Funktionen selbst definieren

In den vorigen Kapiteln haben wir bereits mit einigen Funktionen gearbeitet: Bei `range`, `input`, `print` und `len` etwa handelt es sich um sogenannte *eingegebene Funktionen*, die bereits in Pythons Standardbibliothek definiert sind. Die aktuelle Python-Version verfügt derzeit über 69 solcher Standardfunktionen, die Sie in Pythons Online-Dokumentation aufgelistet finden.

## 9.1 Funktionen selbst definieren

Der wirkliche Vorteil beim Programmieren ergibt sich jedoch erst, wenn Sie eigene Funktionen schreiben lernen. In diesem Abschnitt wollen wir uns daher dieser Möglichkeit widmen. Die allgemeine Methodik ist dabei die folgende: Man definiert eine Funktion, die nach dem Aufrufen einen bestimmten Ausgabewert – ein sogenanntes *Objekt* – liefert, mit dem sich im Anschluss weitere Operationen tätigen lassen.

Um eine Funktion zu erstellen, verwendet man die allgemeine Form:

```
1 def funktionsname(parameter1, ..., parameterN):
2     anweisungsblock
```

Zunächst muss stets das Schlüsselwort `def` angegeben werden. Daraufhin folgt ein selbst gewählter Funktionsname, der die Funktion möglichst gut beschreibt. Die Möglichkeiten der Namensvergabe folgen hier den gleichen Regeln wie bei Variablen. Zwei verschiedene Funktionen müssen stets unterschiedlich benannt werden. Zudem bestehen Funktionsnamen in der Regel nur aus Kleinbuchstaben. Durch einen gut gewählten Funktionsnamen sollte die Aufgabe der Funktion bereits ersichtlich sein.

Die verwendeten Parameter werden anschließend durch Kommas getrennt in einer Klammer aufgelistet. Die Klammer kann einen oder mehrere Parameter enthalten; sie darf jedoch auch leer sein.

Bei den Parametern kann es sich um konkrete Werte (diese heißen *Argumente*), Variablennamen oder auch auszuwertende Ausdrücke handeln. Sie müssen jedoch stets von einem bestimmten Typ sein. In der Regel ist die Parameteranzahl zu Anfang fest durch die Funktion vorgegeben. Dabei gibt es jedoch auch Funktionen, die mit einer variablen Anzahl an Parametern oder fehlenden Argumenten arbeiten können. Wir werden in diesem Kapitel Beispiele für verschiedene Funktionstypen kennenlernen.

9

Wie auch bei den Kontrollstrukturen der vorigen Kapitel wird die Zeile mit dem Funktionskopf durch einen Doppelpunkt abgeschlossen. Eingerückt darunter befindet sich sodann der Funktionskörper, der die auszuführenden Anweisungen enthält. Sobald eine Folgezeile nicht mehr eingerückt ist, erkennt das Programm, dass die Funktionsdefinition an diesem Punkt endet.

Ein einfaches Beispiel einer Funktion könnte etwa folgendermaßen aussehen:

```
1 def willkommen():
2     print("Herzlich willkommen zum Python-Kurs!")
```

Wenn Sie diesen Code laufen lassen, ist die Ausgabe zunächst leer, da das Programm die Definition der Funktion überspringt. Um die Funktion tatsächlich auszuführen und sich die Begrüßungsformel anzeigen zu lassen, müssen Sie den Funktionsnamen `willkommen()` an die gewünschte Stelle des Hauptprogramms schreiben. Wie Sie

## 9 Funktionen in Python

sehen, bleibt in diesem Beispiel die Klammer für die Parameter leer. Im nächsten Abschnitt werden wir untersuchen, wie wir der Funktion Argumente übergeben können.

### 9.2 Argumente für Funktionen verwenden

*„Ein Argument ist eine Aneinanderreihung von Aussagen zur Begründung einer bestimmten Behauptung.“  
„Nein, ist es nicht.“*

– frei übersetzt aus Monty Python

Das in 9.1. vorgestellte Beispiel ist zwar nützlich, wenn man die Begrüßungsformel an verschiedenen Stellen innerhalb des Hauptprogramms verwenden möchte, ohne sie dabei jedes Mal erneut eintippen zu müssen. In Wirklichkeit haben Funktionen im Code jedoch meist einen dynamischeren, flexibleren Aufbau, indem sie etwa auf Nutzereingaben reagieren oder komplexe Berechnungen mit externen Daten ausführen können.

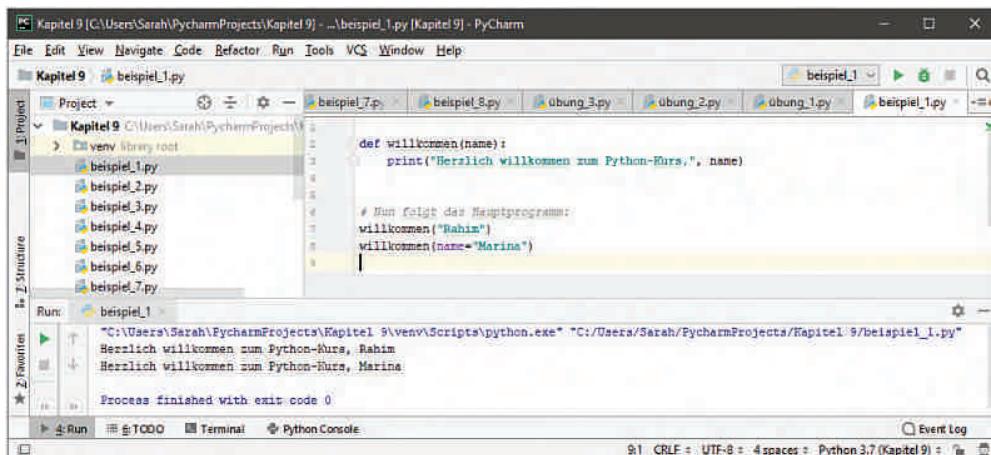
Hierzu dient die Klammer, die im obigen Beispiel leer geblieben war. Sie kann ein oder mehrere Parameter aufnehmen. Wenn Sie eine Funktion mit Parametern angegeben haben, so müssen Sie diese anschließend jeweils mit einem konkreten Wert – dem Argument – versehen, damit die Funktion im Hauptprogramm aufrufbar ist. Sie können bei dieser Methode beim Aufrufen der Funktion also keine leeren Klammern angeben oder Parameter ohne definierte Inhalte verwenden, da das Programm sonst eine Fehlermeldung ausgibt. Der Parameter kann dabei selbstverständlich nicht nur Zeichenketten als Datentypen aufnehmen, sondern auch unterschiedliche Zahlenwerte oder Datenstrukturen wie Listen, Tupel oder Dictionaries.

Vergleichen Sie die folgenden Funktionen:

#### Codebeispiel #1

```
1 def willkommen(name):
2     print("Herzlich willkommen zum Python-Kurs,", name)
3
4 # Nun folgt das Hauptprogramm:
5 willkommen("Rahim")
6 willkommen(name="Marina")
```

## 9.2 Argumente für Funktionen verwenden



**Abb. 9.1** Verschiedene Aufrufmethoden einer Funktion

Die Klammer enthält nun den Parameter `name`, der im Printbefehl der Funktion wiederholt wird. In den letzten beiden Zeilen wird die Funktion sodann mit zwei verschiedenen Argumenten aufgerufen. Dabei ist es entweder möglich, den Personennamen direkt als String anzugeben oder ihn (als sog. *Schlüsselwortparameter*) dem Parameter `name` gleichzusetzen. Das Programm weiß in beiden Fällen automatisch, dass es sich bei den Strings um verschiedene Inhalte von `name` handelt, die in der Begrüßungsformel mit ausgegeben werden.

Eine interaktiverere Begrüßungsmethode wäre die folgende:

### Codebeispiel #2

```

1 def willkommen(name):
2     print("Herzlich willkommen zum Python-Kurs, ", name)
3
4 # Hauptprogramm:
5 Name = input("Geben Sie bitte Ihren Namen ein: ")
6 willkommen(Name)

```

## 9 Funktionen in Python

```

1 def willkommen(name):
2     print("Herzlich willkommen zum Python-Kurs, " + name)
3
4
5 # Hauptprogramm:
6 Name = input("Geben Sie bitte Ihren Namen ein: ")
7 willkommen(Name)

```

**Abb. 9.2** Eine Funktionsausführung in Abhängigkeit von einer Nutzereingabe

Der Parameter `name` in der Funktionsdefinition nimmt hier erst durch die Nutzer eingabe einen beliebigen Wert an, der mittels `Name` an die Funktion übergeben wird. Womöglich ist Ihnen aufgefallen, dass `name` und `Name` hier zwei verschiedene Variablen bezeichnen. Diese Wahl geschah aus gutem Grund: Obwohl es in Python nämlich möglich ist, dieselbe Variable sowohl in der Funktionsdefinition als auch später im Hauptprogramm beim Funktionsaufruf zu verwenden, ist dies nicht ratsam, da hierbei leicht Fehler auftreten können. Nimmt man etwa innerhalb der Funktionsdefinition Änderungen an der Variablen vor oder gibt der Variablen im Hauptprogramm einen anderen Wert, so würden hierdurch zwei verschiedene Variablen entstehen: Eine wäre ausschließlich in der Funktionsdefinition gültig, die andere nur innerhalb des Hauptprogramms. Im folgenden Code werden so etwa für vermeintlich ein und dieselbe Variable zwei verschiedene Outputs erzeugt:

### Codebeispiel #3

```

1 def pluseins():
2     a = 2
3     print("a:", a + 1)
4
5 a = 3
6 pluseins()
7 print("a:", a + 1)

```

## 9.2 Argumente für Funktionen verwenden

```

Kapitel 9 [C:\Users\Sarah\PycharmProjects\Kapitel 9] - ...\\beispiel_3.py [Kapitel 9] - PyCharm
File Edit View Navigate Code Behavior Run Tools VCS Window Help
Kapitel 9 beispiel_3.py
Project Structure Run Favorites Event Log
1 Project 2 Structure 3 Run 4 Favorites 5 Terminal 6 Python Console 7 Event Log
beispiel_1.py beispiel_2.py beispiel_3.py beispiel_4.py beispiel_5.py beispiel_6.py beispiel_7.py beispiel_8.py beispiel_9.py
beispiel_3.py
def pluseins():
    a = 2
    print("a:", a + 1)

a = 3
pluseins()
print("\n", a + 1)
Process finished with exit code 0
10:1 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 9):

```

**Abb. 9.3** Derselbe Variablenname hat zwei verschiedene Gültigkeitsbereiche

Die Funktion `pluseins` addiert zum aktuellen Wert der Variablen `a` jeweils 1 hinzu. Da `a` innerhalb der Funktionsdefinition den Wert 2 hat, lautet die Ausgabe des Printbefehls nach dem Aufrufen der Funktion im Hauptprogramm `a: 3`, obwohl die Variable `a` im Hauptprogramm zuvor mit dem Wert 3 angegeben wurde. Der gleiche Printbefehl, `print("a:", a + 1)`, liefert sodann die Ausgabe `a: 4`, da nun der Variablenwert 3 innerhalb des Hauptprogramms gilt. Obwohl die beiden Variablen dieselbe Bezeichnung haben, unterscheiden sie sich also in ihrem Inhalt. Variablen, die (wie hier im Falle von `a = 2`) nur innerhalb der jeweiligen Funktion gelten, nennt man *lokal*. Im Unterschied hierzu heißen Variablen (hier: `a = 3`), die im gesamten Hauptprogramm Gültigkeit haben, *global*.

Um derartige Unterscheidungsschwierigkeiten zu vermeiden, sollten Sie möglichst darauf verzichten, konkrete Variablenwerte innerhalb der Funktionsdefinition anzugeben und zudem stets zwei verschiedene Variablennamen verwenden: einen innerhalb der Funktion und einen im Hauptprogramm. Dies ist auch in dem Fall sinnvoll, dass die Funktion in einer separaten Datei definiert wird (siehe 9.5.), da die Variablen aus dem Hauptprogramm dann nämlich nicht mehr zur Verfügung stehen. Auch die Verwendung der Funktion innerhalb anderer Programmteile, die eventuell mit abweichenden Bezeichnungen arbeiten, könnte zu Problemen führen.

9

### 9.2.1 Funktionen mit mehreren Parametern

Wir möchten jetzt untersuchen, wie sich mehrere Parameter innerhalb der Funktionsklammer verwenden lassen. Dafür werden die einzelnen Parameter in der Funktionsdefinition jeweils durch ein Komma voneinander getrennt. Die entsprechenden Übergabewerte weisen wir den Variablen sodann in derselben Reihenfolge zu, wie sie im Funktionskopf angegeben sind.

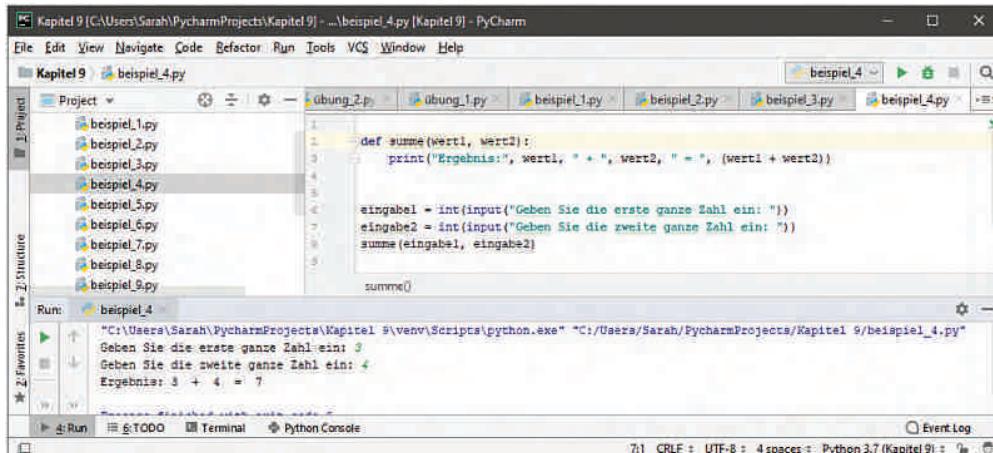
## 9 Funktionen in Python

### Codebeispiel #4

```

1 def summe(wert1, wert2):
2     print("Ergebnis:", wert1, " + ", wert2, " = ", (wert1 + wert2))
3
4 eingabel = int(input("Geben Sie die erste ganze Zahl ein: "))
5 eingabe2 = int(input("Geben Sie die zweite ganze Zahl ein: "))
6 summe(eingabel, eingabe2)

```



**Abb. 9.4** Eine Summen-Funktion mit zwei Eingabeparametern

Wichtig zu beachten ist hierbei, dass dieser Funktion nur genauso viele Werte übergeben werden können, wie es Parameter in der Klammer des Funktionskopfes gibt. Sonst wird eine Fehlermeldung ausgegeben.

Es gibt jedoch auch die Möglichkeit, Funktionen zu formulieren, die eine variable Anzahl an Parametern oder eine variable Anzahl an Übergabewerten aufnehmen können. Wir wollen zuletzt untersuchen, wie diese beiden Alternativen funktionieren.

Beispiele für Standardfunktionen in Python, die mit einer variablen Anzahl an Argumenten arbeiten können, sind etwa `len`, `min` oder `max`. Die erste dürfte Ihnen bereits bekannt sein. Die letzten beiden Funktionen geben automatisch das kleinste bzw. größte aus (mindestens zwei) übergebenen Argumenten aus:

### Codebeispiel #5

```

1 min(1,3,7)
2 max(1,3,7)

```

## 9.2 Argumente für Funktionen verwenden

```
Command Prompt - python
>>> min(1,3,7)
1
>>> max(1,3,7)
7
>>>
```

**Abb. 9.5** Die Standardfunktionen `min` und `max` mit variabler Parameteranzahl

Um eine ähnliche Funktion mit variabler Parameteranzahl zu formulieren, kommt in Python ein Parameter mit einem davorgestellten Stern (\*) zum Einsatz. Der Stern gibt an, dass der folgende Parameter ein Tupel mit keinem, einem oder mehreren Argumenten sein kann. Wir möchten nun eine Durchschnittsfunktion `avg` (für engl. *average*) definieren, die den Durchschnitt einer beliebigen Zahlenmenge bilden kann. Wir nennen den variablen Parameter in unserem Beispiel `*zahlen`:

### Codebeispiel #6

```
1 def avg(*zahlen):
2     summe = 0
3     for x in zahlen:
4         summe += float(x)
5     print("Der Durchschnitt der Zahlen lautet:", summe/len(zahlen))
6
7 avg(1,7,12.5,19,27,34)
```

Kapitel 9 (C:\Users\Sarah\PycharmProjects\Kapitel 9)\...\\beispiel\_6.py [Kapitel 9] - PyCharm

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 9 beispiel_6.py
Project -> beispiel_1.py beispiel_2.py beispiel_3.py beispiel_4.py beispiel_5.py beispiel_6.py beispiel_7.py beispiel_8.py beispiel_9.py beispiel_10.py beispiel_11.py
Kapitel 9 Structure
Run: beispiel_6
  "C:\Users\Sarah\PycharmProjects\Kapitel 9\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 9/beispiel_6.py"
  Der Durchschnitt der Zahlen lautet: 16.75
  Process finished with exit code 0
Run TODO Terminal Python Console Event Log
10:1 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 9) %
```

9

**Abb. 9.6** Die Ausführung einer Durchschnittsfunktion für eine beliebige Zahlenmenge

Hier wurde von der `len`-Funktion Gebrauch gemacht, um die Anzahl der Elemente zu erhalten, aus denen das Zahlentupel besteht. Alternativ hierzu wäre auch denkbar, innerhalb der `for`-Schleife eine Zählvariable zu verwenden, die bei jedem Durchlauf um den Wert 1 erhöht wird und somit am Ende die Anzahl der Elemente liefert. Wie Sie sehen, gibt es in Python oftmals mehrere Möglichkeiten, wie Sie Ihren Code formulieren können um ein gewünschtes Ergebnis zu erhalten.

## 9 Funktionen in Python

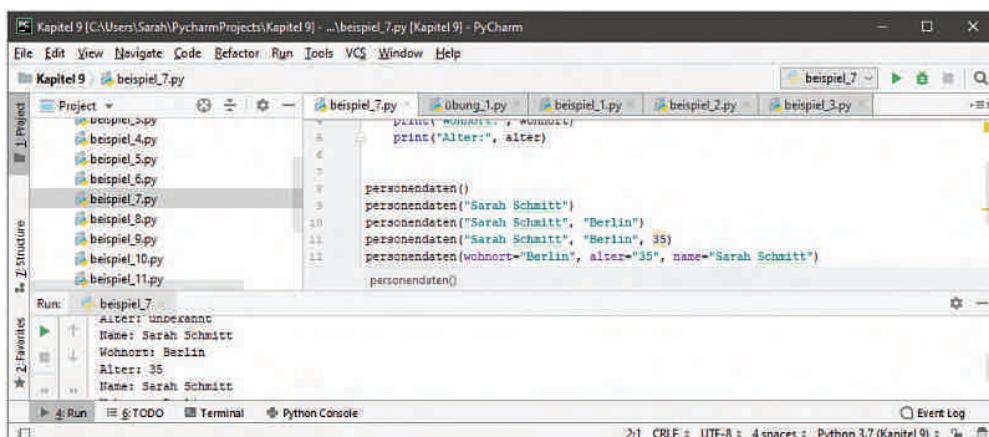
Als Nächstes wollen wir untersuchen, wie sich Funktionen mit einer variablen Anzahl an Übergabewerten realisieren lassen. Für den Fall, dass nicht alle optionalen Parameter ein entsprechendes Argument erhalten, gibt man in der Klammer der Funktionsdefinition Standardwerte an.

### Codebeispiel #7

```

1 def personendaten(name = "unbekannt", wohnort = "unbekannt", alter =
2 "unbekannt"):
3     print("Name:", name)
4     print("Wohnort:", wohnort)
5     print("Alter:", alter)
6
7 personendaten()
8 personendaten("Sarah Schmitt")
9 personendaten("Sarah Schmitt", "Berlin")
10 personendaten("Sarah Schmitt", "Berlin", 35)
11 personendaten(wohnort="Berlin", alter="35", name="Sarah Schmitt")

```



**Abb. 9.7** Eine Funktion mit Standardwerten bei variabler Anzahl an Übergabeargumenten

Beachten Sie dabei, dass die Reihenfolge der Übergabewerte im letzten Funktionsaufruf durch die Angabe der Schlüsselwortparameter keine Rolle mehr spielt; die Werte werden den Parametern anhand des Schlüsselworts automatisch in der richtigen Reihenfolge zugeordnet.

Übrigens ist es ebenfalls möglich, innerhalb der Klammer der Funktionsdefinition sowohl zwingend erforderliche Parameter als auch Parameter mit vordefinierten Standardwerten anzugeben. Zum Beispiel ließe sich der Funktionskopf im obigen Code folgendermaßen abändern:

```
1 def personendaten(name, wohnort="unbekannt", alter="unbekannt"):
```

### 9.3 Einen Rückgabewert verwenden

Dabei gibt das Programm nur für die verbindlichen Parameter eine Fehlermeldung aus, wenn diese keinen Übergabewert erhalten. Sobald mindestens genauso viele Übergabewerte angegeben werden wie es verbindliche Parameter gibt, lässt sich die Funktion ausführen. Damit Python die Reihenfolge richtig zuordnen kann, sollten die verbindlichen Parameter jedoch stets vor den optionalen Parametern stehen, da es beim Funktionsaufruf sonst zu einer Fehlermeldung kommen kann. Beim Aufrufen müssen die verbindlichen Argumente sodann ebenfalls vor den optionalen Argumenten stehen, selbst wenn letztere durch Schlüsselwortparameter angegeben sind.

## 9.3 Einen Rückgabewert verwenden

Im vorigen Abschnitt definierten wir eine Durchschnittsfunktion `avg`, die den Durchschnitt einer beliebigen Zahlenmenge berechnete. Die Ausgabe des Durchschnittswerts erfolgte dabei direkt durch eine Berechnung innerhalb des `print`-Befehls im Funktionskörper. Es gibt jedoch noch einen praktischeren Weg, wie durch die Funktion ermittelte Informationen an das Hauptprogramm zurückgegeben werden können um dort weiterbearbeitet oder mit `print` ausgegeben zu werden. Dies bewerkstelligt die `return`-Anweisung im Funktionskörper. Bei den übermittelten Daten muss es sich selbstverständlich nicht immer zwingend um Zahlenwerte handeln – ebenso denkbar sind beliebige Datentypen, Variablen, Ausdrücke oder Ergebnisse anderer Funktionen, auf die Sie zurückgreifen.

Um den Schlüsselbegriff `return` zu untersuchen, betrachten wir das Beispiel eines Geschäfts, das für seine Produkte eine Funktion aufstellen möchte, die zu jedem eingegebenen Brutto-Preis die derzeitige Mehrwertsteuer berechnet. Mit `return` hätte das entsprechende Programm die folgende Form:

9

### Codebeispiel #8

```

1 def mehrwertsteuer(zahl):
2     mwst = zahl*0.19
3     return mwst
4
5 preis = float(input("Betrag ohne Mehrwertsteuer: "))
6 print("Der Gesamtpreis beträgt", preis + mehrwertsteuer(preis), "€.")

```

## 9 Funktionen in Python

The screenshot shows the PyCharm IDE interface with the following details:

- Project:** Kapitel 9
- File:** beispiel\_8.py
- Code Content:**

```
def mehrwertsteuer(zahl):
    mwst = zahl*0.19
    return mwst

preis = float(input("Betrag ohne Mehrwertsteuer: "))
print("Der Gesamtpreis beträgt", preis + mehrwertsteuer(preis), "€.")
```
- Run:** beispiel\_8
- Output:**

```
"C:/Users/Sarah/PycharmProjects/Kapitel 9/venv/Scripts/python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 9/beispiel_8.py"
Betrag ohne Mehrwertsteuer: 100
Der Gesamtpreis beträgt 119.0 €.

Process finished with exit code 0
```
- Event Log:** CRLF : UTF-8 : 4 spaces : Python 3.7 (Kapitel 9) : %

**Abb. 9.8** Die Rückgabe eines Funktionswerts durch `return`

Um den mwst-Rückgabewert ins Hauptprogramm aufzunehmen, muss man den Funktionswert – in unserem Fall mehrwertsteuer (preis) – lediglich im Hauptprogramm verwenden. Dabei ist es auch möglich, den Funktionswert einer neuen Variablen zuzuordnen, um weitere Berechnungen damit zu tigen.

Um die Funktionsdefinition weiter zu verkürzen, ist es vorteilhaft, von der folgenden Schreibweise Gebrauch zu machen, bei der die Variable `mwst` überflüssig wird:

```
1 def mehrwertsteuer(zahl):  
2     return zahl*0.16  
3  
4 preis = float(input("Betrag ohne Mehrwertsteuer: "))  
5 print("Der Gesamtpreis beträgt", preis + mehrwertsteuer(preis), "€.")
```

Es lassen sich ebenso gut mehrere Werte an das Hauptprogramm übergeben, ohne dass für jeden Wert eine separate Funktion erstellt werden muss. Dazu verwendet man anstatt einer `return`-Variablen ganz einfach eine Listen-, Tupel- oder Dictionary-Datenstruktur. Im folgenden Beispiel wird ein Temperaturwert von 25 °C innerhalb derselben Funktion durch eine `return`-Liste in Fahrenheit und in Kelvin umgerechnet:

## *Codebeispiel #9*

```
1 def temperaturen(x):
2     fahrenheit = x*1.8 + 32
3     kelvin = x + 273.15
4     return[fahrenheit, kelvin]
5
6 werte = temperaturen(25)
7 print("Temperatur in Fahrenheit:", werte[0])
8 print("Temperatur in Kelvin:", werte[1])
```

### 9.3 Einen Rückgabewert verwenden

```

Kapitel 9 [C:\Users\Sarah\PycharmProjects\Kapitel 9] - ...\\beispiel_9.py [Kapitel 9] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 9 beispiel_9.py
Project Z-Struktur Run: beispiel_9
1 Project
2 Favorites
3 beispiel_2.py
4 beispiel_4.py
5 beispiel_5.py
6 beispiel_6.py
7 beispiel_7.py
8 beispiel_8.py
9 beispiel_9.py
10 beispiel_10.py
11 beispiel_11.py
12
13     fahrenheit = x*1.8 + 32
14     kelvin = x + 273.15
15     return[fahrenheit, kelvin]
16
17
18     werte = temperatur(25)
19     print("Temperatur in Fahrenheit:", Werte[0])
20     print("Temperatur in Kelvin:", Werte[1])
21
22     temperaturen()
23
24
25 "C:\Users\Sarah\PycharmProjects\Kapitel 9\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 9/beispiel_9.py"
26 Temperatur in Fahrenheit: 77.0
27 Temperatur in Kelvin: 258.15
28
29 Process finished with exit code 0
30 Run TODO Terminal Python Console
31 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 9) Event Log

```

Abb. 9.9 `return` mit Rückgabewerten in einer Liste

Nach der `return`-Anweisung wird die Funktionsausführung sofort abgebrochen, weitere danach folgende Codezeilen werden ignoriert. Es ist daher möglich, innerhalb einer Funktion mehrere `return`-Anweisungen für verschiedene Fälle einzubauen. Insbesondere bei komplexeren Programmteilen kann dies sehr vorteilhaft sein und Rechenleistung einsparen.

Im folgenden Code wird mittels `if` und `else` überprüft, ob eine bestimmte Zahl gerade ist oder nicht. Ist sie gerade, erhält sie den Rückgabewert `True`, ist sie ungerade, wird sie `False`.

#### Codebeispiel #10

9

```

1 def geradezahl(zahl):
2     if zahl % 2 == 0:
3         return True
4     else:
5         return False
6
7 print(geradezahl(781))
8 print(geradezahl(11) == geradezahl(3))

```

## 9 Funktionen in Python

```

Kapitel 9 | C:\Users\Sarah\PycharmProjects\Kapitel 9 - ...\\beispiel_10.py [Kapitel 9] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 9 beispiel_10.py
Project beispiel_2.py beispiel_4.py beispiel_5.py beispiel_6.py beispiel_7.py beispiel_8.py beispiel_9.py beispiel_10.py beispiel_11.py beispiel_12.py
Structure Run: beispiel_10
Run: *C:\Users\Sarah\PycharmProjects\Kapitel 9\venv\Scripts\python.exe* "C:/Users/Sarah/PycharmProjects/Kapitel 9/beispiel_10.py"
False
True
10:1 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 9) Event Log

```

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_2.py' through 'beispiel\_12.py'. Below it, the 'Structure' tab is selected, showing the file tree. The main code editor window contains the following Python code:

```

def geradezahl(zahl):
    if zahl % 2 == 0:
        return True
    else:
        return False

print(geradezahl(781))
print(geradezahl(11) == geradezahl(3))

```

The 'Run' tab is active, showing the command: "C:\Users\Sarah\PycharmProjects\Kapitel 9\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 9/beispiel\_10.py". The output pane displays the results of the run: "False" on the first line and "True" on the second line.

**Abb. 9.10** Mehrere `return`-Anweisungen innerhalb einer Funktion; `return`-Werte vergleichen

Da 781 keine gerade Zahl ist, lautet die Ausgabe im angezeigten Fall `False`. Eine weitere interessante Verwendungsmethode von Rückgabewerten in Funktionen ist die Möglichkeit, sie durch logische Ausdrücke zu vergleichen. Dies drückt die letzte Zeile im obigen Code aus. Hier wird durch `geradezahl(11) == geradezahl(3)` überprüft, ob die Zahlen 11 und 3 denselben `return`-Wahrheitswert haben. Da es sich sowohl bei 3 und 11 nicht um gerade Zahlen handelt (beide sind `False`), wird die Ausgabe somit `True`.

## 9.4 Rekursion

Es ist möglich, eine Funktion innerhalb derselben Funktion erneut zu verwenden, so dass sie sich beim Ausführen selbst aufruft. Diesen Vorgang nennt man *Rekursion*. Rekursive Funktionen müssen stets mit einer erfüllbaren Abbruchbedingung definiert werden, sodass die Funktion sich nicht ununterbrochen selbst erneut aufruft.

Wir könnten so etwa die Zweierreihe, die wir uns in 8.2. durch eine `for`-Schleife hatten ausgeben lassen, folgendermaßen rekursiv formulieren:

### Codebeispiel #11

```

1 def zweierreihe(zahl):
2     print(zahl)
3     if zahl < 20:
4         zweierreihe(zahl+2)
5
6 zweierreihe(0)

```

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like beispiel\_9.py, beispiel\_10.py, beispiel\_11.py, beispiel\_12.py, übung\_1.py, übung\_2.py, and übung\_3.py. The main code editor window displays the following Python code:

```

def zweierreihe(zahl):
    print(zahl)
    if zahl < 20:
        zweierreihe(zahl+2)

zweierreihe(0)

```

The run configuration dropdown shows "beispiel\_11". The run history shows outputs for 0, 2, 4, and 6. Below the code editor are tabs for Run, TODO, Terminal, and Python Console. The status bar at the bottom indicates the current file is "beispiel\_11.py" and the Python version is "Python 3.7 (Kapitel 9)".

**Abb. 9.11** Eine rekursive Funktionsformulierung der Zweierreihe

Innerhalb der `if`-Anweisung ruft sich die `zweierreihe`-Funktion hierbei selbst erneut auf, jedoch mit einem anderen Argument als dem ursprünglichen (`zahl+2` anstatt `zahl`). Im ersten Schritt wird mit `zweierreihe(0)` also durch den `print`-Befehl innerhalb der Funktionsdefinition die Zahl 0 ausgegeben, woraufhin `zweierreihe(2)` aufgerufen und 2 ausgegeben wird, woraufhin `zweierreihe(4)` im dritten Schritt eine 4 ausgibt, und so weiter.

Die Fakultät einer ganzen Zahl, bei der eine Zahl mit ihren ganzzahligen Vorgängern multipliziert wird (z. B.:  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ ), lässt sich in Python folgendermaßen durch eine rekursive Formel ausdrücken:

#### Codebeispiel #12

9

```

1 def fak(n):
2     if n > 0:
3         return fak(n - 1) * n
4     else:
5         return 1

```

Für `fak(5)` ist der `return`-Wert also `fak(4) * 5`, wodurch `fak(4)` aufgerufen wird, was wiederum den Wert `fak(3) * 2` ergibt, woraufhin `fak(3)` aufgerufen wird, etc.

## 9 Funktionen in Python

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_9.py', 'beispiel\_10.py', 'beispiel\_11.py', 'beispiel\_12.py' (which is currently selected), 'beispiel\_8.py', and 'beispiel\_9.py'. Below the navigation bar is a code editor window containing the following Python code:

```

1 def fak(n):
2     if n > 0:
3         return fak(n - 1) * n
4     else:
5         return 1
6
7 print(fak(6))

```

On the left side of the interface, there's a 'Project' view showing a folder named 'Kapitel 9' containing several Python files. Below the project view is a 'Run' configuration panel with the command: "C:\Users\Sarah\PycharmProjects\Kapitel 9\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 9/beispiel\_12.py". At the bottom of the interface, there are tabs for 'Run', 'TODO', 'Terminal', and 'Python Console', along with status information like '10:1 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 9)'.

Abb. 9.12 Eine rekursive Berechnung der Fakultät

Die Funktionsweise rekursiver Funktionen mag zu Beginn nicht ganz leicht zu verstehen sein. Zudem ist es in vielen Fällen, wie etwa bei der oben beschriebenen zweierreihe-Funktion, oftmals praktischer und effektiver, anstatt einer Rekursionsfunktion eine Schleife zu verwenden. Bei mathematischen Algorithmen wie der Berechnung der Fakultät oder von Fibonacci-Zahlen kann es jedoch oftmals naheliegender und eleganter sein, von rekursiven Funktionen Gebrauch zu machen.

### 9.5 Funktionen in einer eigenen Datei abspeichern

Wie bereits erwähnt, ist es oftmals vorteilhaft, Funktionen außerhalb des Hauptprogramms in eine eigene Datei zu schreiben um sie dort abzuspeichern. Das macht nicht nur das eigentliche Hauptprogramm übersichtlicher, sondern erleichtert auch die Arbeit, wenn mehrere Menschen an unterschiedlichen Programmteilen beteiligt sind. Es ist daher eine gute Idee, sich von Anfang an anzugeben, längere Funktionen in eigene Dateien auszulagern. Wie das genau geht, werden wir nun erklären.

Python-Funktionen, die man in eine separate Datei schreibt, um im Rahmen eines Hauptprogramms darauf zuzugreifen, werden *Module* genannt. Genau genommen handelt es sich eigentlich bei jeder .py-Datei automatisch um ein Modul, das sich in einen anderen Programmteil importieren lässt.

Um dies zu veranschaulichen, verwenden wir unsere zuvor definierte Durchschnittsfunktion avg, die wir in die Datei *durchschnitt.py* geschrieben hatten. Sie bestand aus dem folgenden Code:

```

1 def avg(*zahlen):
2     summe = 0
3     for x in zahlen:
4         summe += float(x)
5     print("Der Durchschnitt der Zahlen lautet:", summe/len(zahlen))

```

## 9.5 Funktionen in einer eigenen Datei abspeichern

Um die Funktion `avg` im Hauptprogramm benutzen zu können, müssen wir zunächst das *durchschnitt*-Modul durch den Befehl `import` und die Nennung des Dateinamens ohne die Endung importieren: `import durchschnitt`. Damit die Funktion `avg` auch im Hauptprogramm zur Verfügung steht, müssen wir sie durch `durchschnitt.avg()` aufrufen. Der entsprechende Codeteil im Hauptprogramm hätte also die folgende Form:

```
1 import durchschnitt
2
3 durchschnitt.avg()
```

Da die Klammer keine Argumente enthält, würde es beim Ausführen dieses Codes zu einer Fehlermeldung kommen. Sie können die Funktionalität von `avg` im Hauptprogramm testen, indem Sie konkrete Werte in die Klammer schreiben.

Um allzu lange Funktionsaufrufe wie den obigen zu vermeiden, können Sie die importierte Funktion im Hauptprogramm mit einem frei wählbaren Namen neu definieren. Den neuen Namen ordnet man ganz einfach durch ein Gleichheitszeichen der importierten Funktion zu. In der Regel verwendet man dabei den gleichen Namen wie den der Funktion innerhalb des Moduls, zum Beispiel: `avg = durchschnitt.avg()`.

Möchten Sie im Hauptprogramm mehrere Module verwenden, so können Sie diese durch Kommas getrennt angeben, etwa: `import durchschnitt, temperatur`. Werden viele Module importiert, so ist es der Übersichtlichkeit halber dennoch sinnvoll, separate `import`-Anweisungen zu verwenden.

Manchmal kann es vorkommen, dass ein Modul eine Vielzahl an Funktionen enthält, die Sie nicht allesamt benötigen. So ist es etwa möglich, dass das *durchschnitt*-Modul viele weitere mathematische Funktionen beinhaltet, obwohl Sie lediglich die `avg`-Funktion in Ihrem Hauptprogramm benötigen. Um in diesem Fall Ressourcen zu sparen, gibt es die Möglichkeit, aus einem Modul nur bestimmte Teile zu selektieren. Dies kann durch eine spezielle `import`-Anweisung realisiert werden, die den allgemeinen Aufbau `from modul import funktion` hat. Beachten Sie hierbei die Konvention, dass Modulnamen genauso wie Funktionen in Kleinbuchstaben stehen. In unserem konkreten Fall wäre dies beispielsweise `from durchschnitt import avg`. Bei dieser Formulierungsweise müssen Sie den Modulnamen beim Funktionsaufruf übrigens nicht vor der Funktion mit angeben.

9

Um zu vermeiden, dass es in PyCharm zu einer Fehlermeldung kommt, sollten Sie sicherstellen, dass sich die Dateien der Module und des Hauptprogramms im selben Ordner befinden. Sollte dies einmal nicht möglich oder zu umständlich sein, können Sie in der PyCharm-IDE angeben, in welchen Pfaden der Interpreter nach Modulen suchen soll. Gehen Sie hierzu auf **File → Settings → Project: [Name] → Project Structure**. Unter **+ Add Content Root** rechts im Fenster können Sie zusätzliche Pfade oder

## 9 Funktionen in Python

Dateien auswählen, die zu Ihrem Projekt hinzugefügt werden sollen, damit der Interpreter die Dateien als Module finden kann.

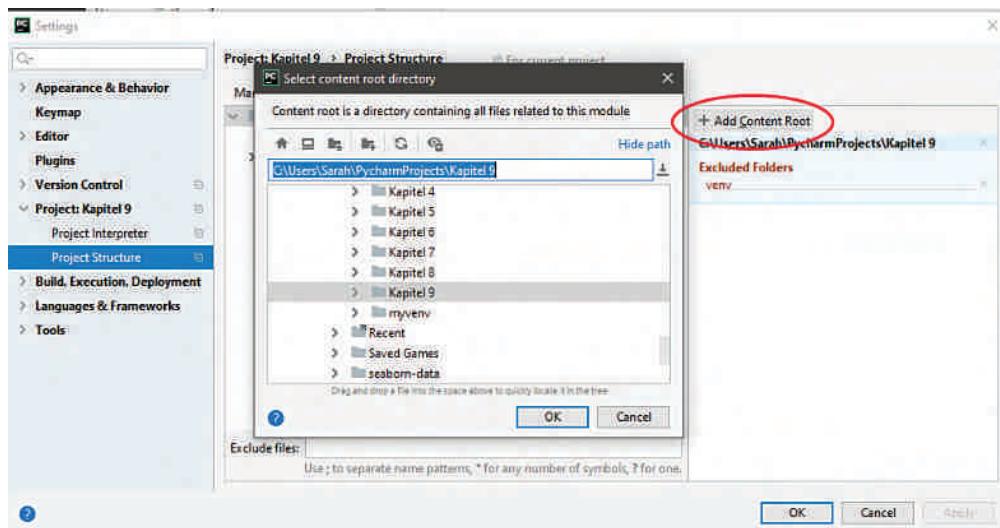


Abb. 9.13 Die Angabe des Pfads für ein importiertes Modul

## 9.6 Übung: Funktionen selbst gestalten

### Übungsaufgaben

1. Sie möchten drei Termine für die nächste Woche mit jeweils einer Tag- und Zeitangabe speichern um sie bei Bedarf durch eine `termine`-Funktion aufrufen und anzeigen lassen zu können. Wählen Sie eine passende Datenstruktur für die Daten jedes Termins und formulieren Sie geschickte Funktionsaufrufe für alle drei Termine. Sehen Sie dabei auch einen optionalen Parameter vor.
2. Definieren Sie eine Funktion, die die Werte zwischen zwei Zahlen (inkl. der beiden Zahlen) in Zweierschritten ausgibt. Für die Ausgabe müssen Sie zwischen zwei Fällen unterscheiden – je nachdem, ob die erste oder die zweite Zahl größer ist. Die beiden Zahlen sollen im Hauptprogramm durch eine Nutzereingabe vorgegeben werden.
3. Im folgenden Code befinden sich vier Fehler. Korrigieren Sie diese und führen Sie das Programm aus. Beschreiben Sie die Aufgabe des Programms.

```
1 def primzahl(zahl)
2     if Zahl <= 2:
3         return True
4     else:
5         for i in range(2, zahl//2):
6             if zahl % i = 0:
7                 return False
8         return True
9
10 print(primzahl(7))
```

## 9 Funktionen in Python

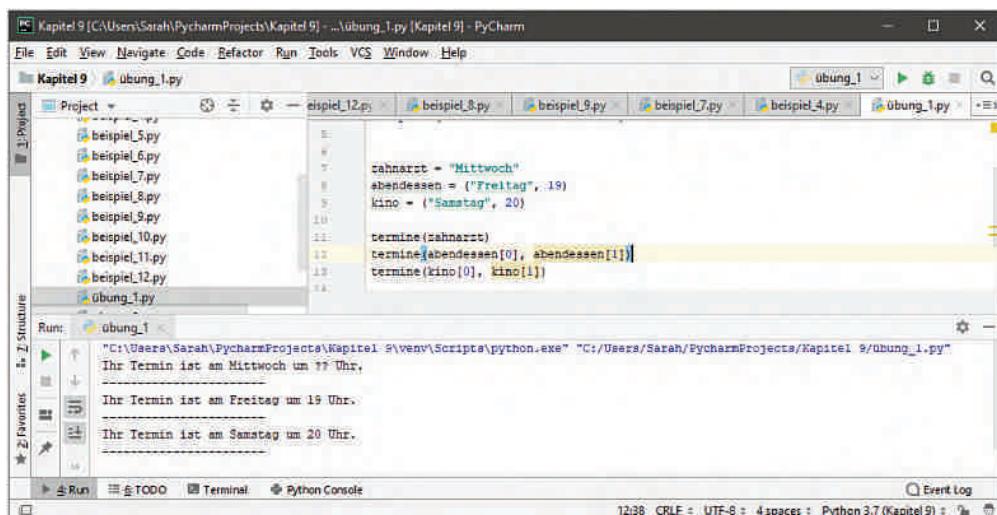
### Lösungen

1.

```

1 def termine(tag, uhrzeit="??") :
2     print("Ihr Termin ist am", tag, "um", uhrzeit, "Uhr.")
3     print("-----")
4
5 zahnarzt = "Mittwoch"
6 abendessen = ("Freitag", 19)
7 kino = ("Samstag", 20)
8
9 termine(zahnarzt)
10 termine(abendessen[0], abendessen[1])
11 termine(kino[0], kino[1])

```



**Abb. 9.14** Termine durch eine Funktion anzeigen lassen

2.

```

1 def zweierschritte(x, y) :
2     if x < y:
3         for i in range(x, y+1, 2):
4             print(i)
5     else:
6         for i in range(y, x+1, 2):
7             print(i)
8
9 zahl1 = int(input("Geben Sie die erste Zahl ein: "))
10 zahl2 = int(input("Geben Sie die zweite Zahl ein: "))
11 zweierschritte(zahl1, zahl2)

```

## 9.6 Übung: Funktionen selbst gestalten

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_8.py', 'beispiel\_9.py', etc., and the current file is 'übung\_2.py'. The code editor displays the following Python script:

```

1 def zweierschritte(x, y):
2     if x < y:
3         for i in range(x, y+1, 2):
4             print(i)
5     else:
6         for i in range(y, x+1, 2):
7             print(i)
8
9 zahl = int(input("Geben Sie die erste Zahl ein: "))
10
11 zahl2 = int(input("Geben Sie die zweite Zahl ein: "))
12
13
14
15
16
17

```

The terminal window below shows the output of running the script:

```

"C:\Users\Sarah\PycharmProjects\Kapitel 9\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 9/übung_2.py"
Geben Sie die erste Zahl ein: 17
Geben Sie die zweite Zahl ein: 5
5
11
13
15
17

```

**Abb. 9.15** Eine beliebige Zahlenreihe in Zweierschritten

3. Fehler: 1. Zeile (:), 2. Zeile (Zahl statt zahl), 6. Zeile (= statt ==), 7. Zeile (falsche Einrückung)

Das Programm überprüft, ob eine gegebene Zahl eine Primzahl ist. Im ersten Unterscheidungsschritt werden die Zahlen 1 und 2 automatisch als Primzahlen erkannt; die Funktion liefert den Rückgabewert True. In der zweiten Fallunterscheidung wird mittels Modulorechnung überprüft, ob die Zahl ganze Teiler hat. Wenn ja, erhält sie den return-Wert False und die Funktion wird abgebrochen. Gibt es keine Teiler, handelt es sich um eine Primzahl und der return-Wert ist True.

```

1 def primzahl(zahl):
2     if zahl <= 2:
3         return True
4     else:
5         for i in range(2, zahl//2):
6             if zahl % i == 0:
7                 return False
8             return True
9
10 print(primzahl(7))

```

## 9 Funktionen in Python

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_8.py', 'beispiel\_9.py', etc., and the current file is 'Übung\_3.py'. The code editor displays the following Python code:

```
def primzahl(zahl):
    if zahl <= 2:
        return True
    else:
        for i in range(2, zahl//2):
            if zahl % i == 0:
                return False
    return True

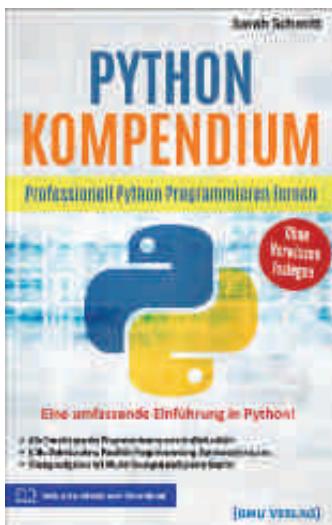
print(primzahl(7))
```

The 'Run' tab in the bottom left shows the command: "C:\Users\Sarah\PycharmProjects\Kapitel 9\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 9/Übung\_3.py". The output window shows the result: "True". The status bar at the bottom right indicates the file is Python 3.7 (Kapitel 9).

Abb. 9.16 Überprüfen, ob eine Zahl eine Primzahl ist

Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 10

# Mit Modulen aus der Standardbibliothek arbeiten

Im vorigen Kapitel haben Sie gelernt, wie man Module erstellt und wie sich diese in ein längeres Hauptprogramm importieren lassen. Module, die von allgemeinem Interesse sind und häufig in Programmen Anwendung finden, werden in Python in so genannten *Bibliotheken* zusammengefasst. Nützliche Funktionen müssen Sie somit nicht jedes Mal erneut schreiben, sondern lassen sich einfach direkt in das eigene Programm importieren, was den Programmierungsvorgang erheblich erleichtert.

Pythons *Standardbibliothek* vereint wesentliche Komponenten und Module für die wichtigsten Anwendungsbereiche des Programmierens. Sie hält eine umfangreiche Bandbreite passender Funktionalitäten für ein sehr vielfältiges Themenspektrum bereit. So gibt es beispielsweise Module zur Erstellung von Datums- und Zeitangaben, für spezifische mathematische Operationen, für Wahrscheinlichkeitsberechnungen, zur Textbearbeitung und viele mehr. Was die Standardbibliothek genau beinhaltet und wie man ihre unterschiedlichen Module verwendet, ist Thema dieses Kapitels.

### 10.1 Was ist die Standardbibliothek und welche Module enthält sie?

Im Laufe von Pythons Entwicklung hat sich eine Ansammlung nützlicher Komponenten und Module herausgebildet, die allen Anwenderinnen und Anwendern innerhalb der Standardbibliothek zur Verfügung stehen. Die Komponenten sind in den Python-Interpreter eingebaut und lassen sich beim Programmieren direkt verwenden, ohne dass sie durch weitere Befehle importiert werden müssen. Einige von ihnen haben wir bereits bei der Arbeit mit diesem Buch verwendet, zum Beispiel eingebaute Funktionen wie `range`, `print` oder `len`, Konstanten wie `True` oder `False` und Typen wie zum Beispiel Fließkommazahlen, Dictionaries oder Tupel.

Die Standardbibliothek umfasst darüber hinaus derzeit über 200 Kernmodule, die die wesentliche Funktionalität der Sprache ausmachen. Die Kernmodule werden bei der Installation des Python-Interpreters automatisch auf Ihrem Computer installiert. Im Aufbau und in ihrer Verwendungsweise ähneln sie dabei den Modulen, mit denen wir uns im vorigen Kapitel beschäftigt haben. Sie lassen sich durch den `import`-Befehl in eigene Programme aufnehmen. Die Verwendung dieser bereits vorhandenen Module spart offensichtlich viel Arbeit beim Programmieren.

Auf der Seite <https://docs.python.org/3/library/> können Sie sich einen Überblick über den gesamten Inhalt der Standardbibliothek verschaffen. Sie müssen sich den Code

## 10.2 Die Verwendung der Standardbibliothek

in den Bibliotheken nicht unbedingt anschauen, um ihn zu verwenden. Ein genauerer Blick in den Code kann jedoch dann sinnvoll sein, wenn Sie dessen Funktionsweise verstehen oder sich neue Programmierkenntnisse aneignen möchten. In jedem Fall sollten Sie aber wissen, wie Sie Zugang zur Dokumentation der Standardbibliothek erhalten und einzelne Module daraus einsetzen können. Dies werden wir Ihnen im Folgenden anhand einiger Modulbeispiele veranschaulichen.



**Abb. 10.1** Die Übersichtsseite über die Standardbibliothek im Internet

Über den oben genannten Link gelangen Sie zur Übersicht über die Python-Standardbibliothek. Sie können dieses Nachschlagewerk verwenden, um mehr Informationen zu einzelnen Modulen und deren Funktionsweise zu erhalten. Insbesondere dann, wenn Sie noch keine Erfahrung mit einem bestimmten Modul und dessen Funktionen haben, kann es eine gute Idee sein, darüber nachzulesen, bevor Sie diese anwenden.

## 10.2 Die Verwendung der Standardbibliothek

Schauen Sie sich zunächst einmal den Inhalt der Standardbibliothek auf der angegebenen Seite an. Anhand der Bezeichnungen der unterschiedlichen Module ist oftmals bereits ersichtlich, welches Modul zur Lösung einer bestimmten Aufgabe das passende ist. Wenn Sie das entsprechende Modul anklicken, gelangen Sie zur Modulbeschreibung, in der Sie die unterschiedlichen Funktionen samt notwendigen Angaben zu Argumenten und Rückgabewerten finden.

## 10 Mit Modulen aus der Standardbibliothek arbeiten

Wenn Sie nicht auf Anhieb das passende Modul finden, können Sie anhand eines Suchbegriffs danach suchen. Es werden Ihnen daraufhin Vorschläge zu infrage kommenden Modulen angezeigt. Die Suchfunktion befindet sich im rechten oberen Teil der Seite. Da die Dokumentation der Standardbibliothek derzeit nicht in deutscher Sprache vorliegt, sollten Sie Ihren Suchbegriff standardmäßig in englischer Sprache eingeben. Links oben auf der Seite haben Sie zudem die Möglichkeit, andere Sprachen wie Französisch oder Japanisch auszuwählen.

Lesen Sie sich die Funktionsbeschreibung genau durch und gehen Sie sicher, dass Sie die Aufgabe der Funktion wirklich verstanden haben, ehe Sie sie benutzen. Achten Sie dabei insbesondere darauf, wie die Argumente und Rückgabewerte verwendet werden. Um ein bestimmtes Modul in Ihr Programm aufzunehmen, müssen Sie dieses wie in Kapitel 9.5. beschrieben durch den Befehl `import` importieren. Danach stehen Ihnen die Funktionen innerhalb des Programms zur Verfügung.

Wir werden Ihnen im Folgenden einige der nützlichsten Standardbibliothek-Module samt deren Funktionen vorstellen. Sollten einige der Module für Ihr aktuelles Lern- oder Arbeitsvorhaben nicht von Interesse sein, können Sie diese auch zunächst überspringen und bei Bedarf zu einem späteren Zeitpunkt darauf zurückkommen. Das Wissen über diese Module ist für den weiteren Verlauf des Buches nicht zwingend erforderlich.

### 10.2.1 Datum- und Zeitanzeigen mit `time`, `datetime` und `calendar`

Die Module `time`, `datetime` und `calendar` enthalten unterschiedliche Funktionen für die Arbeit mit zeitbezogenen Informationen und sind für vielerlei Zwecke sehr nützlich. Wir wollen uns in diesem Abschnitt vorrangig mit dem Modul `time` beschäftigen, das Sie unter <https://docs.python.org/3/library/time.html#> finden.

Python arbeitet mit Zeitangaben, indem es die Zeit in Sekunden ab Beginn einer so genannten *epoch* zählt, die auf den meisten Systemen am 1. Januar 1970 um 00:00:00 beginnt. Die seit *epoch* bis zum aktuellen Zeitpunkt verstrichenen Sekunden können mithilfe der Funktion `time.time()` angezeigt werden, die kein Argument aufnimmt. Etwas nützlicher ist hier jedoch die Funktion `time.ctime()`, die die aktuelle Ortszeit als String ausgibt. Enthält die Funktionsklammer eine Zahl, so wird diese Zahl als Sekunden zu *epoch* (eventuell unter Berücksichtigung der Ortszeit) addiert und der berechnete Zeitpunkt angezeigt. Die Ausgabe als String ist hilfreich, wenn man mit den Zeitangaben weitere Programmschritte tätigen möchte.

#### Codebeispiel #1

```
1 import time
2
3 time.time()
```

## 10.2 Die Verwendung der Standardbibliothek

```
4 time.ctime()
5 time.ctime(60)
```

```
C:\ Command Prompt - python
>>> import time
>>> time.time()
1616632142.1379817
>>> time.ctime()
'Thu Mar 25 01:29:11 2021'
>>> time.ctime(60)
'Thu Jan 1 01:01:00 1970'
>>>
```

**Abb. 10.2** Die Zeitfunktionen `time` und `ctime` im `time`-Modul

Um zu überprüfen, was der *epoch*-Zeitpunkt in Ihrem System ist, können Sie nach dem Importieren des `time`-Moduls den Befehl `time.gmtime(0)` eingeben. Die Null bedeutet dabei, dass seit *epoch* keine Zeit (d. h. null Sekunden) vergangen ist. Das im Anschluss angezeigte, recht kryptisch aussehende Darstellungsformat lässt sich durch die Funktion `time.asctime()` in eine verständlichere Form bringen. Ist in der Klammer kein Argument angegeben, so gibt der Interpreter das aktuelle Datum und die Uhrzeit aus:

### Codebeispiel #2

```
1 zeit = time.gmtime(0)
2 time.asctime(zeit)
3 time.asctime()
```

```
C:\ Command Prompt - python
>>> import time
>>> zeit = time.gmtime(0)
>>> time.asctime(zeit)
'Thu Jan 1 00:00:00 1970'
>>> time.asctime()
'Thu Mar 25 01:31:05 2021'
>>>
```

10

**Abb. 10.3** Verschiedene Zeitanzeigen

Die Funktion `asctime()` kann entweder ein aus neun Elementen bestehendes Tupel oder ein `struct_time`-Objekt (wie oben durch `gmtime()` ausgegeben) aufnehmen und verwandelt dieses in einen String-Datentyp. Sie können unter der Klasse `time.struct_time` in der Beschreibung des `time`-Moduls nachlesen, welche Elementreihenfolgen und Zahlenbereiche dabei jeweils für die Argumente gelten. Was ein Objekt und eine Klasse ist, werden wir im nächsten Kapitel ausführlicher behandeln. Betrachten Sie nun als konkretes Beispiel das Tupel:

## 10 Mit Modulen aus der Standardbibliothek arbeiten

### Codebeispiel #3

```
1 zeitpunkt = (2019, 9, 20, 21, 8, 00, 4, 263, 1)
2 time.asctime(zeitpunkt)
```

```
>>> zeitpunkt = (2019, 9, 20, 21, 8, 00, 4, 263, 1)
>>> time.asctime(zeitpunkt)
'Fri Sep 20 21:08:00 2019'
>>>
```

**Abb. 10.4** Datum- und Zeitausgabe anhand eines Tupels

Die letzte Zahl in der Klammer gibt an, ob Winter- oder Sommerzeit gilt und kann dementsprechend die Werte 0, 1 oder -1 (falls nicht bekannt) annehmen. Die vorletzte Zahl in der Klammer bezieht sich auf Schalt- und Gemeinjahre und gibt an, um den wievielten Tag im Jahr es sich handelt.

Im Unterschied zur Funktion `gmtime()`, die mit der koordinierten Weltzeit (UTC) arbeitet, arbeitet `localtime()` mit der Ortszeit. Wird der Klammer eine Zahl übergeben, so werden dementsprechend viele Sekunden zum Startzeitpunkt *epoch* hinzugefügt und dieser Zeitpunkt wird als der aktuelle genommen. Auch `localtime` gibt, genauso wie `gmtime`, ein Objekt vom Typ `struct_time` aus. Da es sich bei diesem Objekt um ein Tupel handelt, kann man sich die einzelnen Elemente darin entweder durch deren Index oder durch ihre Attribute anzeigen lassen, wie das folgende Beispiel zeigt. Hier wird das aktuelle Datum in deutscher Schreibweise ausgegeben:

### Codebeispiel #4

```
1 time.localtime(10)
2 zeit_akt = time.localtime()
3 zeit_akt
4 print("Datum: %d.%d.%d\nUhrzeit: %d:%d:%d" % (zeit_akt.tm_mday, zeit_akt[1],
5 zeit_akt[0], zeit_akt.tm_hour, zeit_akt.tm_min, zeit_akt[5]))
```

```
>>> import time
>>> time.localtime(10)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=1, tm_min=0, tm_sec=10, tm_wday=3, tm_yday=1, tm_isdst=0)
>>> zeit_akt = time.localtime()
>>> zeit_akt
time.struct_time(tm_year=2021, tm_mon=3, tm_mday=25, tm_hour=1, tm_min=34, tm_sec=18, tm_wday=3, tm_yday=84, tm_isdst=0)
>>> print("Datum: %d.%d.%d\nUhrzeit: %d:%d:%d" % (zeit_akt.tm_mday, zeit_akt[1], zeit_akt[0], zeit_akt.tm_hour, zeit_akt.tm_min, zeit_akt[5]))
Datum: 25.03.2021
Uhrzeit: 1:34:18
>>>
```

**Abb. 10.5** Die Funktion `localtime`

## 10.2 Die Verwendung der Standardbibliothek

Eine sehr nützliche Funktion ist ebenso `strftime()`. Diese nimmt ein Tupel oder ein `struct_time`-Objekt als Argument auf und gibt je nach angegebenem Format eine Zeichenkette aus. Versuchen Sie einmal, zu welchem Output der Befehl `zeitstring = time.strftime("%d/%m/%Y, %H:%M:%S", zeitpunkt)` mit unserem obigen `zeitpunkt`-Tupel führt, indem Sie sich die Variable `zeitstring` ausgeben lassen. `%Y`, `%m`, `%d`, `%H`, etc. stehen dabei für bestimmte Formatcodierungen, die Sie auf der Standardbibliothek-Webseite im `time`-Modul erklärt finden.

Eine weitere interessante Funktion des `time`-Moduls ist `sleep`. Hierbei wird die weitere Ausführung eines Programms um die Anzahl der Sekunden verzögert, die in den Klammern angegeben ist. Das folgende Programm gibt beispielsweise nach 7,5 Stunden eine Weckmeldung aus:

### Codebeispiel #5

```
1 time.sleep(7.5*60*60)
2 print("Guten Morgen! Es ist Zeit, aufzustehen!")
```

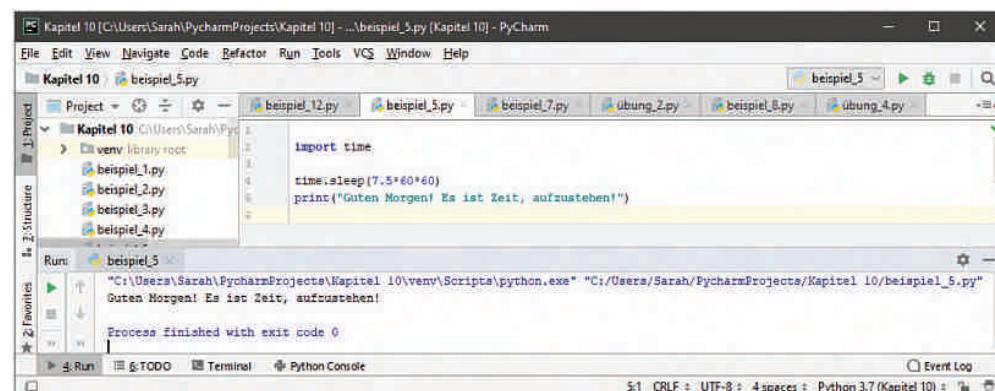


Abb. 10.6 Verzögerte Programmausführung mit `sleep`

Allein das `time`-Modul beinhaltet über 20 Funktionen. Nun, da Sie ein Gespür dafür bekommen haben, wie dieses Modul verwendet wird, können Sie einen Blick auf die anderen zeitbezogenen Module `datetime` und `calendar` werfen. Obwohl deren Funktionalität Ähnlichkeiten zum `time`-Modul hat, unterscheiden sie sich in ihrer Methodik und ihren Möglichkeiten mitunter sehr stark.

### 10.2.2 math: ein häufig verwendetes Modul

Pythons Kernfunktionen enthalten bereits die Grundrechenarten, mithilfe derer sich viele mathematische Funktionen erstellen lassen. Beispielsweise hatten wir in Abschnitt 9.4. die Fakultät einer ganzen Zahl rekursiv durch `if`, `else` und einfache

## 10 Mit Modulen aus der Standardbibliothek arbeiten

mathematische Operatoren definiert, was einige Zeilen an Code erforderte. Genau so könnte man die Absolutfunktion, die einer Zahl ihren Abstand zu Null, den so genannten *Absolutbetrag*, zuordnet (z. B. haben -5 und 5 beide den Absolutbetrag 5) durch mehrere Codezeilen definieren. Um sich diese zusätzliche Programmierarbeit zu ersparen, können Sie das Modul `math` benutzen, dessen Beschreibung Sie unter <https://docs.python.org/3/library/math.html> finden.

Das `math`-Modul enthält vordefinierte Funktionen wie die Fakultät `factorial()` oder die Absolutfunktion `fabs()`, die sich nach dem Importieren des Moduls durch `import math` anhand des Funktionsbegriffs aufrufen lassen. In den folgenden Beispielen werden wir den Importbefehl nicht jedes Mal erneut wiederholen und möchten an dieser Stelle lediglich daran erinnern, dass sich anstatt des gesamten Moduls auch einzelne Funktionen daraus importieren lassen. Für den nächsten Anwendungsfall lässt sich dies beispielsweise durch den Befehl `from math import factorial, fabs` realisieren. Wie in Kapitel 9.5. erwähnt, muss der Modulname bei dieser Importierungsmethode nicht vor der Funktion stehen:

### Codebeispiel #6

```
1 factorial(5)
2 fabs(-5) == fabs(5)
3 fabs(-5)
```

```
c:\> Command Prompt - python
>>> from math import factorial, fabs
>>> factorial(5)
120
>>> fabs(-5) == fabs(5)
True
>>> fabs(-5)
5.0
>>> -
```

**Abb. 10.7** Die math-Funktionen `fabs` und `factorial`

In der zweiten Eingabezeile wird überprüft, ob -5 und 5 jeweils denselben Absolutbetrag haben. Da der Abstand beider Zahlen zu Null 5 beträgt, lautet die Ausgabe `True`. Wie Sie sehen, gibt der Interpreter in der letzten Ausgabezeile eine Fließkommazahl aus.

In der Tat sind die meisten Ausgaben der Funktionen innerhalb des `math`-Moduls Fließkommazahlen. Aufgrund dieser Eigenschaft ist dieses Modul oftmals besser für präzise Berechnungen geeignet als viele der im Python-Interpreter standardmäßig eingebauten Funktionen. Betrachten Sie diesbezüglich den folgenden Code, in dem zweimal die Summe mehrerer Zahlen berechnet wird:

## 10.2 Die Verwendung der Standardbibliothek

### Codebeispiel #7

```

1 from math import fsum
2
3 sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
4 fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])

```

```

C:\ Command Prompt - python
>>> from math import fsum
>>> sum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
0.9999999999999999
>>> fsum([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
1.0
>>>

```

**Abb. 10.8** Die beiden Summenfunktionen `sum` und `fsum`

Die Funktion `sum` ist eine der 69 Standardfunktionen, die nach der Installation des Interpreters automatisch verfügbar sind, `fsum` hingegen muss aus dem `math`-Modul importiert werden. Ohne auf den genauen Grund für den unterschiedlichen Output und die Ungenauigkeit von `sum` einzugehen, möchten wir Ihnen nahelegen, insbesondere bei Berechnungen mit Fließkommazahlen lieber `fsum` anstatt `sum` zu benutzen.

Wir werden abschließend ein Beispiel betrachten, in dem das Programm von der Anwenderin oder vom Anwender zwei Zahlen abfragt, die anschließend mit der Potenzfunktion `pow()` berechnet werden. Wenn Sie auf der `math`-Seite nach dem Begriff „power“ suchen (z. B. durch die Tastenkombination **Strg + F**), so gelangen Sie zur Beschreibung von `math.pow(x, y)`, die folgendermaßen beginnt: „Return  $x$  raised to the power  $y$ “. Das bedeutet, dass der linke Operand,  $x$ , mit dem rechten Operanden,  $y$ , folgendermaßen potenziert wird:  $x^y$ . Vielleicht können Sie sich nun bereits vorstellen, wie der entsprechende Code mit Anwenderaufforderungen aussehen wird:

10

### Codebeispiel #8

```

1 x = float(input("Geben Sie den Basiswert ein: "))
2 y = float(input("Geben Sie den Exponenten ein: "))
3 ergebnis = math.pow(x, y)
4 ergebnis

```

```

C:\ Command Prompt - python
>>> import math
>>> x = float(input("Geben Sie den Basiswert ein: "))
Geben Sie den Basiswert ein: 2
>>> y = float(input("Geben Sie den Exponenten ein: "))
Geben Sie den Exponenten ein: 3
>>> ergebnis = math.pow(x, y)
>>> ergebnis
8.0
>>>

```

**Abb. 10.9** Die Berechnung einer Potenz mit `pow`

## 10 Mit Modulen aus der Standardbibliothek arbeiten

Es gibt weitere hilfreiche Funktionen und Konstanten in `math`, die das umständliche, händische Formulieren mathematischer Rechenvorschriften unnötig machen. Diese umfassen gewöhnliche Zahlen- und Darstellungsfunktionen (z. B. `fmod` für Modulo-rechnung oder `gcd` für den größten gemeinsamen Teiler), Potenz- und logarithmische Funktionen (z. B. die Exponentialfunktion  $e^x$  mit `exp`, Wurzelberechnung durch `sqrt`, den Logarithmus zur Basis 10, `log10`), trigonometrische, hyperbolische und Winkelfunktionen samt andere Spezialfunktionen sowie Konstanten wie  $e$  (`e`) und  $\pi$  (`pi`).

Wenn Sie an Aufgaben arbeiten, in denen mathematische Berechnungen vorkommen, lohnt es sich auf jeden Fall, einen Blick in das `math`-Modul der Standardbibliothek zu werfen.

### 10.2.3 Den Zufall mit `random` programmieren

Da Computer – sofern sie funktionieren – bekanntlich immer bestimmten logischen Vorschriften folgen, mag es zunächst überraschen, dass sie überhaupt zufallsähnliche Operationen ausführen können. In der Tat arbeiten Computer in der Regel nicht mit echten Zufallszahlen, sondern mit sogenannten Pseudozufallszahlen. Diese sind nichts anderes als eine scheinbar zufällige, deterministisch generierte Zahlenfolge, die sich zyklisch innerhalb des Zufallsalgorithmus wiederholt, sobald eine bestimmte Anzahl an generierten Zufallszahlen durchlaufen wurde. Innerhalb von Python's Standard-Zufallsalgorithmus besteht ein solcher Zyklus aus  $2^{19937} - 1$  Pseudozufallszahlen.

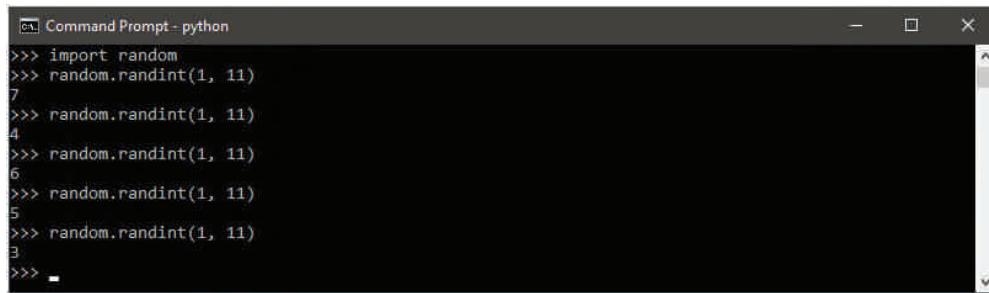
Um in Python mit Zufallszahlen zu arbeiten, müssen Sie das Modul `random` (engl. für *zufällig*) importieren. Aufgrund seiner deterministischen Natur ist dieses Modul jedoch nicht für Sicherheits- oder kryptographische Anwendungen geeignet. Für derartige Zwecke sollten Sie stattdessen das Modul `secrets` in Betracht ziehen. Auf der Seite <https://docs.python.org/3/library/random.html#module-random> finden Sie die Beschreibung des `random`-Moduls.

Um sich eine zufällig ausgewählte Zahl zwischen zwei ganzen Zahlen anzeigen zu lassen, können Sie die Funktion `randint` verwenden. Die komplette Funktionsbeschreibung lautet `random.randint(a, b)`. Dabei sind die Endpunkte `a` und `b` ebenfalls im Zahlenbereich der möglichen Zufallszahlen enthalten.

#### Codebeispiel #9

```
1 random.randint(1, 11)
```

## 10.2 Die Verwendung der Standardbibliothek



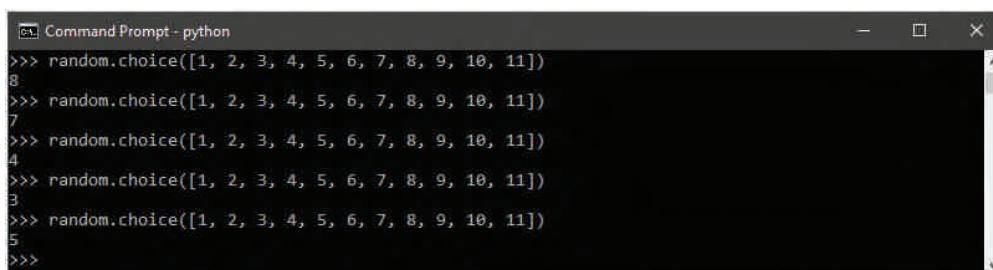
```
Command Prompt - python
>>> import random
>>> random.randint(1, 11)
7
>>> random.randint(1, 11)
4
>>> random.randint(1, 11)
6
>>> random.randint(1, 11)
5
>>> random.randint(1, 11)
3
>>>
```

**Abb. 10.10** Eine Zufallsausgabe von Zahlen in einem wählbaren Zahlenbereich

Eine ähnliche Funktionalität bietet die Funktion `choice`. Hierbei wird aus einer vor gegebenen Zahlenfolge zufällig eine Zahl ausgewählt. Die einzelnen Zahlen müssen dabei durch Kommas getrennt in einer Sequenz innerhalb der Funktionsklammer stehen, wie das folgende Beispiel zeigt:

### Codebeispiel #10

```
1 random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```



```
Command Prompt - python
>>> random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
8
>>> random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
7
>>> random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
4
>>> random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
3
>>> random.choice([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
5
>>>
```

**Abb. 10.11** Eine Zufallsausgabe einer Zahl in einer Zahlenfolge

10

Für dieses Beispiel wählten wir als Sequenz eine Liste, deren einzelne Elemente Zahlen waren. Anstatt einer Liste kämen jedoch auch die zwei anderen sequentiellen Datentypen – Tupel und Zeichenketten – in Frage. Die enthaltenen Elemente müssen dabei nicht unbedingt numerischer Art, sondern können auch beliebige sequentielle Datentypen sein. Um dies zu testen, geben Sie in der Eingabeaufforderung beispielsweise `random.choice(("Athen", "Berlin", "Oslo", ("Python", 77), 1, 2, 3))` oder `random.choice("Berlin")` ein. Im erstgenannten Fall sind die Elemente innerhalb der Funktionsklammer wiederum eine Mischung aus beliebigen sequentiellen Datentypen. Im zweiten Fall wird ein Buchstabe innerhalb der Zeichenkette ausgewählt und ausgegeben. Möchten Sie mehr als ein Element aus einer Sequenz zufällig auswählen, so können Sie hierfür die Funktion `choices` verwenden. Dabei gibt ein  $k$ -Zahlenwert innerhalb der Klammer an, wie viele Elemente ausgewählt werden sollen. So könnten Sie etwa mit `random.choices([1, 2, 3], k=2)` zwei Zahlen

## 10 Mit Modulen aus der Standardbibliothek arbeiten

aus 1, 2 und 3 auswählen. Diese Funktion beschreibt die Methode *Ziehen mit Zurücklegen*. Im `random`-Modul existiert auch eine Funktion, die die Methode *Ziehen ohne Zurücklegen* realisiert. Diese heißt `sample`. Wir werden sie gleich im Anschluss an das folgende Codebeispiel betrachten.

Die Funktion `random.shuffle` ordnet die Elemente einer Sequenz zufällig neu an. Da hierbei die komplette Sequenz geändert und neu abgespeichert wird, kann `shuffle` nur veränderliche sequentielle Datentypen aufnehmen. Vielleicht erinnern Sie sich: Bei Zeichenketten und Tupeln handelt es sich um unveränderliche Datentypen. Aus diesem Grund kommen hier nur Listen als Funktionsargumente infrage. Allerdings kann die Liste wiederum aus unterschiedlichen Datentypen bestehen, wie das folgende Beispiel zeigt. Beachten Sie hier auch die letzte Eingabezeile, deren Output beweist, dass die Liste tatsächlich in einer neuen Reihenfolge abgespeichert wurde.

### Codebeispiel #11

```
1 x = [1, 2, 3, "Python"]
2 random.shuffle(x)
3 x
```

```
Command Prompt - python
>>> x = [1, 2, 3, "Python"]
>>> random.shuffle(x)
>>> x
[2, 'Python', 3, 1]
>>>
```

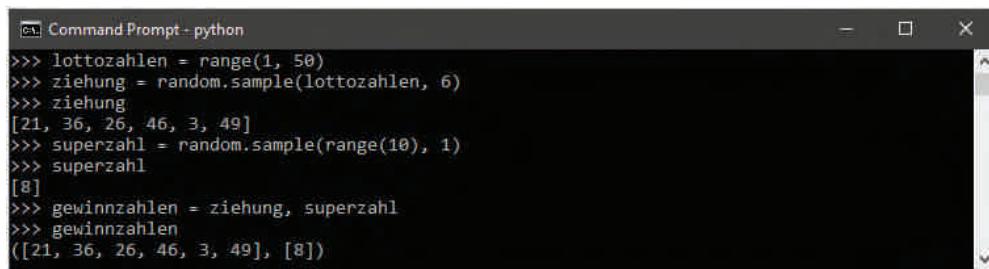
Abb. 10.12 Eine Sequenz zufällig neu anordnen

Ein Paradebeispiel für Zufallsrechnungen ist das Lottospiel. Selbstverständlich verfügt Python auch für diesen Fall über eine passende Funktion: `sample`. Hier wird eine frei wählbare Anzahl an Elementen aus einer Gesamtmenge zufällig ausgewählt und angeordnet. Die Gesamtmenge muss dabei als sequentieller Datentyp vorliegen. Wir möchten dies sogleich anhand eines Lottobeispiels untersuchen, bei dem 6 Zahlen aus 49 möglichen sowie eine Superzahl (0-9) gezogen werden:

### Codebeispiel #12

```
1 lottozahlen = range(1, 50)
2 ziehung = random.sample(lottozahlen, 6)
3 ziehung
4 superzahl = random.sample(range(10), 1)
5 superzahl
6 gewinnzahlen = ziehung, superzahl
7 gewinnzahlen
```

## 10.2 Die Verwendung der Standardbibliothek



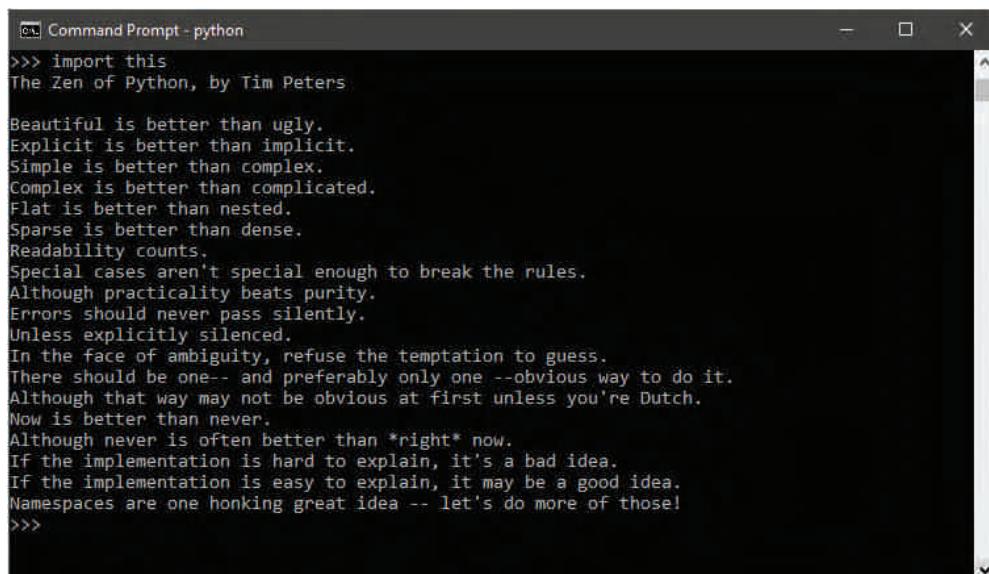
```
>>> lottozahlen = range(1, 50)
>>> ziehung = random.sample(lottozahlen, 6)
>>> ziehung
[21, 36, 26, 46, 3, 49]
>>> superzahl = random.sample(range(10), 1)
>>> superzahl
[8]
>>> gewinnzahlen = ziehung, superzahl
>>> gewinnzahlen
([21, 36, 26, 46, 3, 49], [8])
```

Abb. 10.13 Lottospiel „6 aus 49“ mit `sample`

Bei der Definition von `superzahl` wurde `range` der Einfachheit halber sofort in die Funktionsklammer mit aufgenommen. Selbstverständlich können Sie `ziehung` ebenfalls ähnlich verkürzt definieren. Am Ende werden die Gewinnzahlen als Tupel ausgegeben, das die beiden Listen der gezogenen Zahlen enthält.

Das Modul `random` verfügt über weitaus mehr Funktionen für verschiedene Anwendungsfälle. Insbesondere finden sich darin zusätzlich viele unterschiedliche Methoden zur Erzeugung von Pseudozufallszahlen. In den Übungen werden Sie gleich die Gelegenheit haben, Ihr bisheriges Wissen zu vertiefen.

Geben Sie zuletzt einmal den Befehl `import this` in den Interpreter ein. Die vollständige „Zen of Python“ wird angezeigt – eine nützliche Sammlung aus 19 Prinzipien, die Sie beim Programmieren im Hinterkopf behalten und befolgen können, sollte der Weg durch den Python-Dschungel einmal unklar werden.



```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

10

Abb. 10.14 Die Anzeige der „Zen of Python“- Prinzipien mit `import this`

## 10 Mit Modulen aus der Standardbibliothek arbeiten

Übrigens lassen sich auch externe Module, die nicht Teil der Standardbibliothek sind, innerhalb von Python verwenden. Dies geschieht standardmäßig durch den Befehl `pip install Paketname` aus der Kommandozeile heraus. Eine Sammlung dieser Module finden Sie auf der Webseite des *Python Package Index* unter <https://pypi.org/>. Wir werden diese Methodik beispielsweise in Kapitel 15.1. anhand eines konkreten Beispiels einsetzen.

## 10.3 Übung: Mit der Standardbibliothek arbeiten

### Übungsaufgaben

1. Schreiben Sie ein Programm, das das aktuelle Datum und die Ortszeit in deutscher Schreibweise in einem Satz anzeigt.
2. Berechnen Sie Quersumme und Quadratwurzel einer seitens der Nutzerin oder des Nutzers eingegebenen ganzen Zahl.
3. Schreiben Sie ein Programm, bei dem aus einer Trommel mit 100 Losen genau ein Los gezogen wird. Eines der Lose ist der Hauptgewinn, die anderen 99 sind Nieten. Wird das Gewinnerlos gezogen, gibt das Programm eine entsprechende Erfolgsmeldung aus. Ist dies nicht der Fall, erfolgt eine Meldung, dass der Hauptgewinn nicht gezogen wurde.
4. Simulieren Sie den schrittweisen Vorgang des Kartenmischens und -austeilens in einem Algorithmus. Verwenden Sie hierfür einen Standard-Kartensatz von 52 Karten, von denen Sie nach dem Mischen jeweils vier Stück an fünf Spieler austeilten. Lassen Sie sich das Blatt jedes Spielers ausgeben.

**Tipp:** Legen Sie zunächst zwei Listen der Kartenwerte und -farben an. Erstellen Sie anschließend mithilfe der `product`-Funktion des `itertools`-Moduls in der Standardbibliothek eine neue, vollständige Liste aller Kombinationen aus beiden Listen. Lesen Sie hierfür gegebenenfalls über die Verwendung von `product` in der Standardbibliothek nach. Anschließend können Sie den Mischvorgang durch eine passende Funktion aus `random` beschreiben. Das wiederholte Austeilen lässt sich durch eine Schleife realisieren, die nach jedem Austeilen die bereits verwendeten Karten aus dem restlichen Kartenstapel löscht.

Versuchen Sie, das Programm möglichst geschickt mit wenigen Codezeilen zu formulieren.

## 10 Mit Modulen aus der Standardbibliothek arbeiten

### Lösungen

1.

```

1 import time
2
3 datum_zeit = time.localtime()
4 zeit = time.strftime("Heute ist der %d.%m.%Y um %H:%M:%S Uhr.", datum_zeit)
5 print(zeit)

```

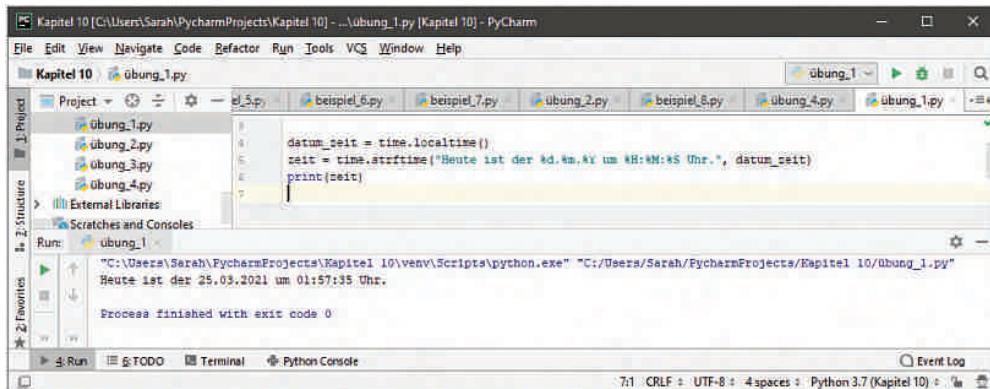


Abb. 10.15 Eine Ausgabemöglichkeit des Datums mit der Ortszeit

2.

```

1 import math
2
3 ganze_zahl = int(input("Geben Sie eine ganze Zahl ein: "))
4 qs = math.fsum(int(ziffer) for ziffer in str(ganze_zahl)) # Quersumme
5 print("Die Quersumme von %d lautet:" % ganze_zahl, qs)
6 print("Die Quadratwurzel von %d lautet:" % ganze_zahl, math.sqrt(ganze_zahl))
7 # Wurzel

```

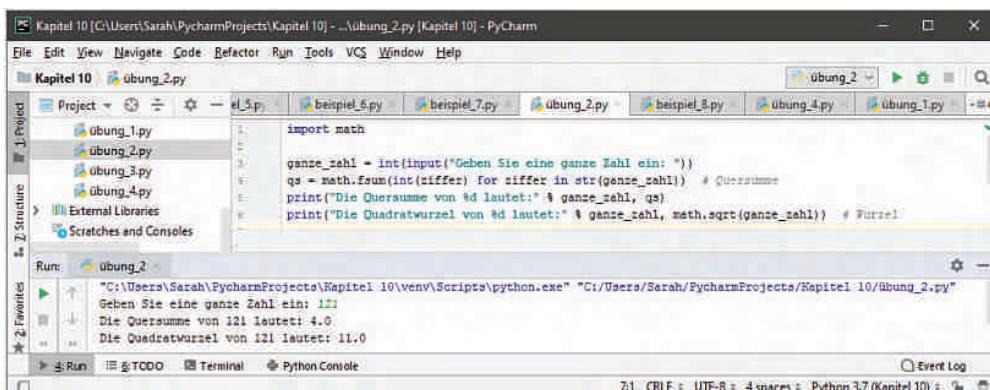


Abb. 10.16 Die Berechnung der Quersumme und Wurzel einer Zahl mit math

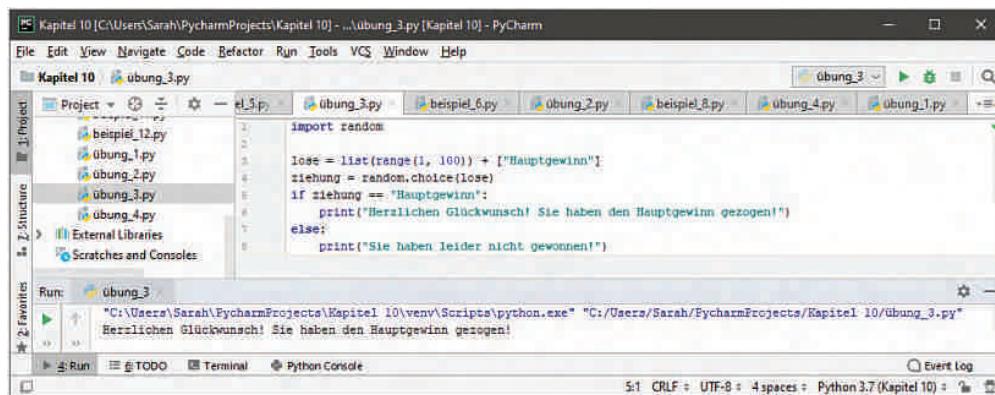
### 10.3 Übung: Mit der Standardbibliothek arbeiten

3. Sie können die 100 Lose auch auf eine andere Art und Weise erzeugen. Für das Ziehen eines einzigen Elements ist die Funktion `random.choice` die passende. Dieses Beispiel ist ein klassischer Fall für `if-else`-Klauseln.

```

1 import random
2
3 lose = list(range(1, 100)) + ["Hauptgewinn"]
4 ziehung = random.choice(lose)
5 if ziehung == "Hauptgewinn":
6     print("Herzlichen Glückwunsch! Sie haben den Hauptgewinn gezogen!")
7 else:
8     print("Sie haben leider nicht gewonnen!")

```



**Abb. 10.17** Ein Los aus einer Lostrommel ziehen

4. Vergessen Sie nicht, die beiden Module zu importieren. Nachdem der Kartenstapel `Kartenspiel` erstellt wurde, wird dieser durch `shuffle` gemischt. Danach werden dem Stapel von oben jeweils vier Karten entnommen und an die Spieler ausgegeben. Die `while`-Schleife wiederholt den Vorgang für alle fünf Spieler und entfernt bei jeder Wiederholung die bereits verwendeten Karten aus `Kartenspiel`.

Eine andere Lösungsmöglichkeit, bei der jedoch die schrittweise Vorgangsbeschreibung verloren geht, lässt sich durch die Funktion `sample` realisieren. Versuchen Sie es!

```

1 import random
2 import itertools
3
4 werte = list(range(2, 11)) + ["Bube", "Dame", "König", "Ass"]
5 farben = ["Kreuz", "Pik", "Herz", "Karo"]
6 kartenspiel = list(itertools.product(werte, farben)) # Legt eine neue Liste
7 mit den Kombinationen beider Listen an
8
9 random.shuffle(kartenspiel) # Hier werden die Karten gemischt
10
11 n = 1
12 while n <= 5:
13     spieler = kartenspiel[:4] # Verteilt 4 Karten an einen Spieler

```

## 10 Mit Modulen aus der Standardbibliothek arbeiten

```

14     print("Karten des %d. Spielers:" % n, spieler)
15     for i in spieler:
16         kartenspiel.remove(i) # Entfernt 4 Karten aus dem Stapel
17     n += 1

```

The screenshot shows the PyCharm IDE interface with the project 'Kapitel 10' open. The file 'Übung\_4.py' is selected in the 'Run' dropdown. The code in the editor is:

```

import random
import itertools

werte = list(range(2, 11)) + ["Bube", "Dame", "König", "Ass"]
farben = ["Kreuz", "Pik", "Herz", "Karo"]
kartenspiel = list(itertools.product(werte, farben)) # Löst eine neue Liste mit den Kombinationen h

random.shuffle(kartenspiel) # Hier werden die Karten gemischt

n = 1
while n <= 5:
    spieler = kartenspiel[:4] # Verteilt 4 Karten an einen Spieler
    print("Karten des %d. Spielers:" % n, spieler)
    for i in spieler:
        kartenspiel.remove(i) # Entfernt 4 Karten aus dem Stapel
    n += 1

```

The 'Run' tab shows the command: "C:\Users\Sarah\PycharmProjects\Kapitel 10\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 10/Übung\_4.py". The output window displays the cards dealt to each player:

```

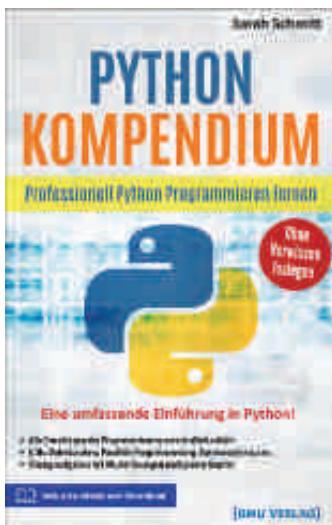
Karten des 1. Spielers: [(4, 'Herz'), (10, 'Karo'), (6, 'Pik'), (7, 'Herz')]
Karten des 2. Spielers: [(4, 'Pik'), (4, 'Karo'), (5, 'Karo'), (5, 'Kreuz')]
Karten des 3. Spielers: [(10, 'Kreuz'), (7, 'Pik'), (7, 'Kreuz'), ('Bube', 'Karo')]
Karten des 4. Spielers: [(10, 'Herz'), ('Dame', 'Karo'), (3, 'Karo'), (9, 'Karo')]
Karten des 5. Spielers: [(2, 'Karo'), ('Ass', 'Karo'), (10, 'Pik'), ('König', 'Herz')]

```

Abb. 10.18 Karten aus einem Kartenstapel mischen und an fünf Spieler verteilen

Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

# Kapitel 11

## Objektorientierte Programmierung

In den vorigen Kapiteln – insbesondere Kapitel 5, 6, 9 und 10 – gebrauchten wir mitunter bereits stillschweigend die Begriffe Objekt, Methode und Klasse gemäß dem alltäglichen, intuitiven Sprachgebrauch, ohne dabei genau darauf einzugehen, was diese Wörter in der Welt des Programmierens genau bedeuten.

Vielleicht haben Sie schon einmal von objektorientierter Programmierung (kurz: *OOP*) gehört. Objektorientierung ist ein sehr umfassender Begriff, der in vielen Programmiersprachen von zentraler Bedeutung ist. Kurz gesagt steckt dahinter der Versuch, die Welt als System von Objekten zu betrachten und diese realen Objekte auf der Programmebene wiederzugeben. Dadurch soll einerseits die Wirklichkeit besser in Programmen repräsentiert und andererseits durch Komplexitätsreduktion eine effizientere Arbeitsweise ermöglicht werden. Die meisten modernen Programmiersprachen unterstützen objektorientierte Programmierung. Im Folgenden werden wir Ihnen erklären, was sich genau hinter diesem Begriff verbirgt und wie sich Objektorientierung im Rahmen eines Programms implementieren lässt. Sie werden lernen, selbst Objekte und die ihnen übergeordneten Klassen zu definieren sowie dazugehörige Eigenschaften und Funktionen festzulegen.

Um Ihnen einen kleinen Vorgesmack auf Pythons Welt der Klassen, Methoden und Objekte zu geben, empfehlen wir den Monty-Python-Sketch „Cheese Shop“, der die Klasse „Käse“ mit ihren zahlreichen Käsesorten (den Objekten) auf treffende Weise beschreibt: [www.youtube.com/watch?v=HzIJWzyvv8A](https://www.youtube.com/watch?v=HzIJWzyvv8A).

### 11.1 Was ist objektorientierte Programmierung?

Objektorientierte Programmierung (*OOP*) in der Informatik beschreibt einen fundamentalen Programmierstil – ein sogenanntes *Programmierparadigma* – bei dem ein Ausschnitt aus der Wirklichkeit in vereinfachter Form innerhalb eines Programms nachgebildet wird. Die Architektur des Programms ahmt dabei gewissermaßen denjenigen Bereich der Realität nach, den sie beschreibt. Die Grundstruktur des Programms ist jedoch nicht bloß beschreibend, sondern stets auf ein konkretes Ziel bzw. auf eine Aufgabe ausgerichtet, die das Programm handhaben soll. Sie können sich als analoges Beispiel etwa vorstellen, dass ein Stift (das sogenannte *Objekt*) in einem Ladengeschäft einen anderen Zweck erfüllen wird, als wenn er in Ihrem Zuhause auf einem Blatt Papier liegt. Dementsprechend werden in den beiden Situationen auch unterschiedliche, den Stift beschreibende Informationen von Bedeutung sein. So würde man einem Stift im Ladengeschäft etwa Angaben zu Artikelnummer, Preis

## 11.1 Was ist objektorientierte Programmierung?

oder Marke zuordnen, wohingegen bei Ihnen zuhause Eigenschaften wie Qualität, Farbe oder Auffindbarkeit von vorrangiger Bedeutung sind.

*Objekte* machen den Grundbaustein objektorientierter Programmiersprachen aus, zu denen neben Sprachen wie C++, Java oder C# auch Python zählt. Objekte im Python-Code können jedoch weitaus mehr sein als Gegenstände mit einem realen Pendant wie der Stift in unserem obigen Beispiel. Grundsätzlich hat man es in Python immer mit einem konkreten Objekt irgendeiner Klasse zu tun. So handelt es sich nicht nur bei explizit definierten Klassen, sondern auch bei Datentypen wie Zahlen oder Zeichenketten, bei Funktionen und Modulen und bei eingebauten Datenstrukturen wie Listen oder Dictionaries im Wesentlichen um Klassen mit entsprechenden spezifischen Objekten. Die definitorische Abgrenzung dessen, was ein Objekt innerhalb einer Programmiersprache ist, ist also nicht sehr scharf gefasst. Trotzdem haben alle Objekte gemeinsam, dass ihnen gewisse Eigenschaften (sog. *Attribute*) und *Methoden* zugeordnet werden. Die Attribute drücken dabei die spezifische Charakteristik aus, die allen Objekten der Klasse eigen ist, während die Methoden – ganz wie Funktionen – Operationen mit den einzelnen Objekten ermöglichen. Vom objektorientierten Standpunkt aus betrachtet sind Objekte somit lebendige Instanzen, die auf entsprechende Botschaften von außen reagieren und selbstständig die geforderten Operationen durchführen. In diesem Punkt unterscheiden sie sich also von unserer üblichen Definition eines Objekts.

In unseren bisherigen Programmen hatten wir es meistens mit einzelnen Variablen und deren Werten zu tun. So hätten wir für einen Stift beispielsweise farbe = "schwarz", preis = 3,99 und qualität = "sehr gut" angeben können. Diese einzelnen Eigenschaften hätten wir jedoch für jeden Gegenstand vom Typ „Stift“ separat erfassen müssen, wobei der übergeordnete Typ ungenannt im Hintergrund geblieben wäre. Beim objektorientierten Programmieren verhält es sich anders. Hier werden alle Eigenschaften und zugehörigen Programmteile sogleich unter einem übergeordneten Typ zusammengefasst, der sich in seiner festen Struktur direkt abrufen lässt, sodass sich die einzelnen Gegenstände bzw. Objekte darin leichter erzeugen und handhaben lassen. Diese Vorgehensweise bildet auch unsere menschliche Wahrnehmung der Wirklichkeit besser nach.

11

Das Konzept der *Klasse* bezeichnet hierbei diese den Objekten übergeordnete Struktur (z. B. die Klasse *Stift* in unserem Beispiel), in der alle Objekte mit der gleichen Charakteristik zusammengefasst werden. Das bedeutet, dass man zur Beschreibung der Objekte einer Klasse dieselben Attribute verwendet, obwohl deren konkrete Ausprägungen unterschiedlich sein können. Klassen bilden somit die wesentliche Grundlage objektorientierter Programme.

Bei *Methoden* handelt es sich um Funktionen, die Bestandteil einer bestimmten Klasse sind. Die Objekte einer Klasse können die durch die Methoden definierten Ope-

## 11 Objektorientierte Programmierung

rationen ausführen, wenn sie im Programm die entsprechende Botschaft hierfür erhalten. Durch das Versenden und Empfangen solcher Nachrichten (die wiederum aus Objekten bestehen), können Objekte also untereinander kommunizieren und Informationen austauschen.

In diesem Buch haben wir bereits klassenähnliche Strukturen erzeugt, ohne dabei explizit objektorientierten Code geschrieben zu haben. So hatten wir beispielsweise in Kapitel 6.3. anhand von Dictionaries mit Schlüssel-Wert-Paaren einen Bibliotheksbestand erstellt. Dabei hatten wir für jedes einzelne Werk ein neues Dictionary mit den jeweiligen Angaben angelegt. Eine feste, übergeordnete Struktur fehlte dabei jedoch. Mit objektorientierter Programmierung hätten wir für den Bibliotheksbestand ganz einfach eine Klasse `Werk` definieren können, die für jedes Werk automatisch alle Attribute mit unterschiedlichen Werten enthält.

In einem zweiten Schritt könnte man den Bibliotheksbestand in weitere Untergruppen wie etwa Belletristik, Kinderbücher, Sachliteratur, fremdsprachige Literatur etc. aufteilen. Dabei würde jede untergeordnete Kategorisierung wiederum dieselben beschreibenden Eigenschaften wie die ihr übergeordnete Klasse erhalten. Zusätzlich kann sie jedoch weitere Details beinhalten, die nur für die spezifische Untergruppe gelten. So könnten Kinderbücher etwa mit einer Altersempfehlung versehen sein, die für die anderen Büchergruppen keine Rolle spielen, oder es könnte die Sprache eines fremdsprachigen Buchs angegeben sein.

Hinsichtlich einer übergeordneten Kategorisierung wäre es etwa denkbar, dass eine Stadt den kompletten Bestand ihrer einzelnen Stadtbibliothekfilialen in einem übergeordneten System erfasst, in denen der Bestand jeder Filiale eine Untergruppe mit weiteren Unterkategorien ist. Die Klasse aller Stadtbibliothekfilialen würde dabei nur diejenigen Eigenschaften beinhalten, die alle Filialbestände gemeinsam haben. Dieses Prinzip von unter- und übergeordneten Kategorien verwenden wir im Alltag täglich in den verschiedensten Bereichen. Die objektorientierte Programmierung macht sich diese Denkweise unter dem Begriff der *Vererbung* ebenfalls zunutze. Auf welche Weise Oberklassen ihre Attribute und Methoden an von ihnen abgeleitete Klassen vererben können, werden wir genauer in Abschnitt 11.6. untersuchen.

Nachdem Sie nun den – zugegebenermaßen etwas komplexen – Grundgedanken der OOP verstanden haben, werden wir uns in den nächsten Abschnitten genauer ihren einzelnen Aspekten widmen und untersuchen, wie man objektorientiert programmiert. Keine Sorge – das ist viel einfacher, als es klingt.

### 11.2 Klassen: die Grundlage der objektorientierten Programmierung

Genauso wie Menschen automatisch bestimmte Kategorisierungen für ähnliche Gegenstände in ihrer Umwelt vornehmen und somit Denk- und Wahrnehmungsprozes-

## 11.2 Klassen: die Grundlage der objektorientierten Programmierung

se beschleunigen und vereinfachen, dienen Klassen in Programmiersprachen dazu, gleichartige Objekte auf praktische Art und Weise zu verwalten. Man kann sich eine Klasse wie eine Konstruktionsvorlage für ein konkretes Objekt vorstellen. So bedarf es beispielsweise zunächst eines Entwurfs für ein Stiftmodell samt allen wichtigen Details, bevor die Stifte (jeweils einzelne Objekte) in Massenproduktion hergestellt werden und über das Fließband laufen können.

Klassen können Sie genauso wie Funktionen entweder im Hauptprogramm oder in einer separaten Datei anlegen. Entscheiden Sie sich für Letzteres, müssen Sie das Modul später zusätzlich in das Hauptprogramm importieren.

Beim Erstellen einer Klasse sollten Sie sich im ersten Schritt überlegen, welche Eigenschaften diese beinhalten soll. Da ein Objekt, zum Beispiel ein Stift, theoretisch betrachtet unzählige Eigenschaften haben kann, müssen Sie eine passende Auswahl treffen, die dem Verwendungszweck Ihres Programms dient.

Stellen Sie sich vor, Sie sind Schreibwarenhändlerin und möchten ein Programm für Ihren Ladenbestand schreiben. Sie können dabei eine Klasse für alle Objekte vom Typ „Stift“ anlegen, die von unterschiedlichen Herstellern produziert, jeweils mit Artikelnummern versehen und unterschiedlich bepreist sind. Nehmen wir also diese drei Eigenschaften, um die Objekte unserer Klasse `Stift` zu beschreiben.

Zur Definition der Klasse wird im Definitionskopf der Schlüsselbegriff `class` verwendet, auf den der entsprechende Klassenname, in diesem Fall also `Stift`, folgt. Wählen Sie hier vorzugsweise eine kurze Bezeichnung, die die Klassenobjekte passend beschreibt. Um Klassennamen optisch von Variablen und Funktionen abzugrenzen, wählt man in der Regel Großbuchstaben als Anfangsbuchstabe. Der Doppelpunkt am Ende des Definitionskopfes darf dabei auch hier nicht fehlen:

```
1 class Stift:
```

In der ersten Zeile wird zunächst der Behälter für die gewünschten Daten erzeugt, der die möglichen Objekte aufnimmt und zu einer Einheit zusammenfasst. In der nächsten Zeile steht der Körper der Klassendefinition. Hier wollen wir nun die Grundstruktur der Klassenobjekte näher durch unsere drei gewählten Attribute `marke`, `artikelnummer` und `preis` beschreiben. Diese können wir jedoch nicht direkt unter den Klassenkopf eingeben, da wir sonst den einzelnen Variablen sogleich konkrete Werte zuordnen müssten, die sodann für alle Objekte aus der Klasse gelten würden. Sie können sich denken, dass es eher unpraktisch wäre, wenn alle Stifte in unserer Klasse dieselbe Marke oder denselben Preis hätten. Konkrete Attributwerte verwendet man im Klassenkörper also nur, wenn der Wert des Attributs tatsächlich für alle Objekte gelten soll. Sein Gebrauch gleicht dann dem einer gewöhnlichen Variablen. Attribute dieser Art nennt man *Klassenattribute* (auch: *statische Attribute*). Für unsere Klasse könnten wir hier beispielsweise als Klassenattribut `schreibgeraet = True`

## 11 Objektorientierte Programmierung

angeben, um die Klasse `Stift` von der Klasse `Papier` im Schreibwarengeschäft abzugrenzen.

Um einzelne Objekte innerhalb des Klassenkörpers mit Attributnamen und konkreten Werten zu versehen, kommt in Python eine spezielle Initialisierungsmethode mit einem sogenannten *Konstruktor* zum Einsatz. Hierbei handelt es sich um eine integrierte Funktion, die für die spätere Erzeugung der konkreten Objekte Parameter entgegennehmen kann. Jede Klasse muss einen Konstruktor enthalten. Wird kein solcher explizit angegeben, verwendet Python den Standardkonstruktor, der mit keinerlei Aktionen verbunden ist. Die *Konstruktormethode* wird durch `def` eingeleitet und im Anschluss durch den Funktionsnamen `__init__()` ausgewiesen. Nur so weiß Python, dass es sich bei der Funktion um die Konstruktormethode handelt. In der Klammer steht sodann der Konstruktor-Begriff `self` gefolgt von den jeweiligen Übergabewerten der Funktion – wie wir dies auch in Kapitel 9 beim Definieren von Funktionen getan hatten.

Unsere Klassendefinition hat nun insgesamt die folgende Form:

```
1 class Stift:
2     schreibgeraet = True # dies ist hier optional
3     def __init__(self, marke, artnr, preis):
```

Vergessen Sie hier nicht den Doppelpunkt am Ende der letzten Zeile! Der Begriff `self` in der Funktionsklammer gibt dabei an, dass die Konstruktormethode mit Objekten innerhalb dieser Klasse arbeitet. Die Parameter `marke`, `artnr` und `preis` stehen im Anschluss jeweils für die Werte, die vom Hauptprogramm an den Konstruktor übergeben werden. Die Bezeichnungen der Werte sind wie bei allen Funktionen frei wählbar. Ihnen werden die gewählten Eigenschaften zugeordnet, die die Objekte der Klasse beschreiben – in unserem Fall `marke`, `artikelnummer` und `preis`. Anschließend werden die einzelnen Variablen und die Übergabewerte nach dem Schema `self.artikelnummer = artnr` darunter in Verbindung gebracht. In vollständiger Form lautet unsere Klassendefinition:

```
1 class Stift:
2     schreibgeraet = True # dies ist hier optional
3     def __init__(self, marke, artnr, preis):
4         self.marke = marke
5         self.artikelnummer = artnr
6         self.preis = preis
```

In dieser Form lässt sich die `Stift`-Klasse bereits verwenden. Wenn Sie die Klasse als Python-Code in PyCharm ausführen, werden Sie keinerlei Output erhalten. In Abschnitt 11.3. werden wir genauer untersuchen, wie man konkrete Instanzen für Objekte erstellt und wie sich diese innerhalb eines Programms verwenden lassen. Doch zuerst wollen wir Ihnen an dieser Stelle kurz eine praktische Kommentaroption für Klassen, Module und Funktionen vorstellen.

## 11.2 Klassen: die Grundlage der objektorientierten Programmierung

### 11.2.1 Docstrings

Wenn man eine Klasse definiert, fügt man üblicherweise mindestens einen sogenannten *Docstring* (von engl. *documentation string*) ein. Dieser ist im Prinzip nichts anderes als ein Kommentar, der keinerlei Auswirkungen auf das Programm hat, jedoch eine hilfreiche Beschreibung der Klasse darstellt. Der Gebrauch von Docstrings ist dabei nicht nur auf Klassen beschränkt. Docstrings werden beispielsweise auch gerne für die Dokumentation von Modulen oder Funktionen eingesetzt. Im Falle von Funktionen etwa beinhaltet ein Docstring in der Regel eine Beschreibung der Aufgabe der Funktion, eine Auflistung der Eigenschaften, die die Parameter erfüllen müssen samt deren zurückgegebenen Objekten, einen Überblick über die globalen Parameter sowie ganz am Ende den Namen der Autorin oder des Autors und das Datum der letzten Änderung.

Insbesondere dann, wenn man mit fremdem Quellcode arbeitet, sind Docstrings sehr hilfreich. Anhand von Docstrings lassen sich Zweck und Anwendung einer Klasse schneller verstehen. Docstrings werden durch drei doppelte Anführungszeichen (""""") eingeleitet und auf dieselbe Weise abgeschlossen. In einer Klassendefinition fügt man meist einen Docstring direkt unter den Klassenkopf sowie einen unter die erste Zeile der Konstruktormethode ein. Innerhalb des Konstruktors beschreiben Docstrings beispielsweise die zu übergebenen Argumente.

Wir können unsere `Stift`-Klasse nach dem folgenden Muster mit Docstrings ausstatten:

```

1 class Stift:
2     schreibgeraet = True # dies ist hier optional
3     """Erstellt das Objekt Stift für einen Schreibwarenladen."""
4     def __init__(self, marke, artnr, preis):
5         """
6             Initialisiert ein neues Objekt Stift
7
8             Argumente:
9             * marke (string): Marke des Stifts
10            * artikelnummer (int): Artikelnummer des Stifts
11            * preis (float): Verkaufspreis des Stifts
12
13            self.marke = marke
14            self.artikelnummer = artnr
15            self.preis = preis
16

```

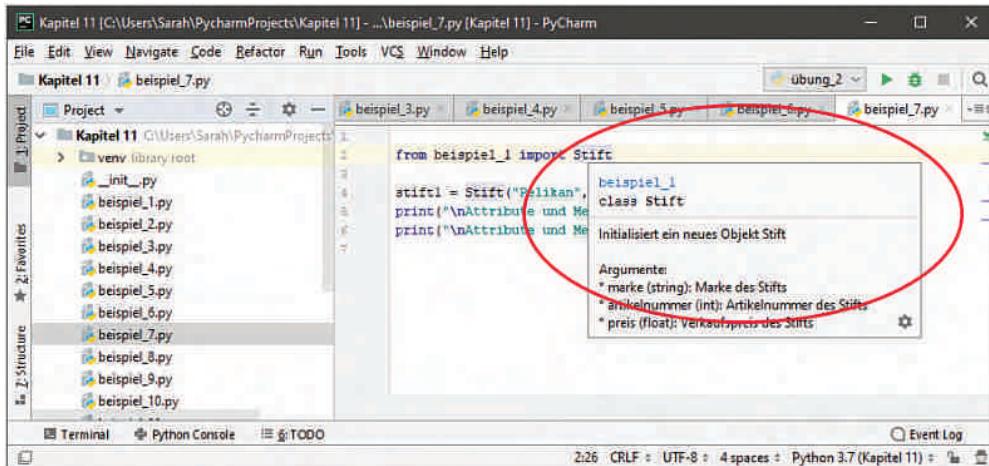
11

Wie Sie sehen, können Docstrings je nach Länge ein- oder mehrzeilig sein.

Angenommen, Sie haben an anderer Stelle eine Klasse definiert und möchten diese anschließend im Hauptprogramm verwenden, haben allerdings die genaue Funktionsweise und weitere Details der Klasse vergessen. In PyCharm gibt es mehrere praktische Möglichkeiten, wie Sie sich alle Informationen zu einer Klasse anzeigen

## 11 Objektorientierte Programmierung

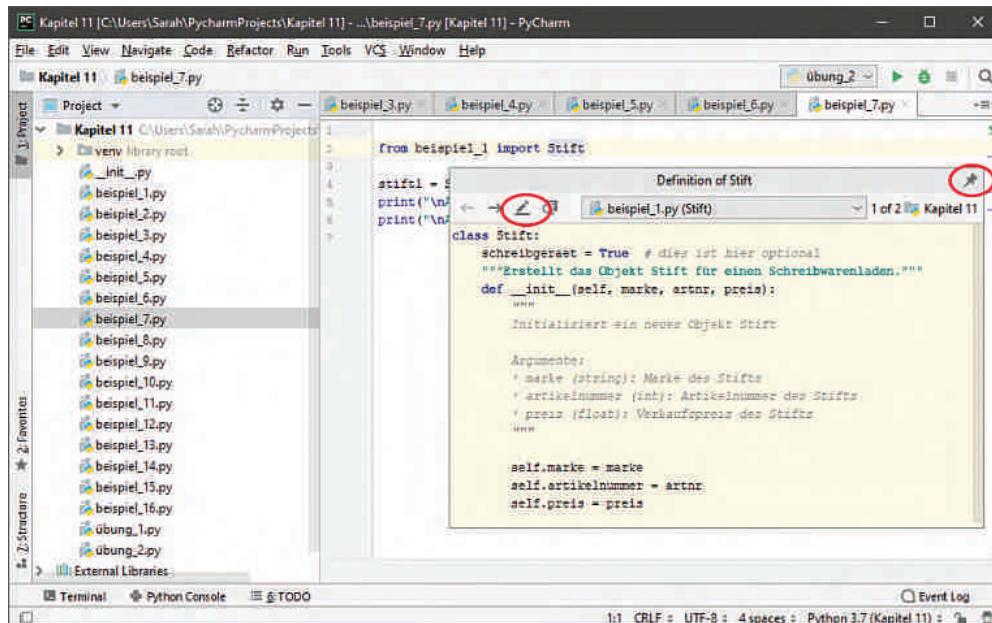
lassen können. Dazu tippen Sie einfach den Klassennamen an der Stelle, wo Sie ihn im Hauptprogramm verwenden möchten, als Code ein. Anschließend klicken Sie mit dem Cursor auf den Klassennamen. Um sich nur die Docstrings anzeigen zu lassen, drücken Sie **Strg + Q**. Ein Pop-up-Fenster mit den Docstrings wird eingeblendet. Zum gleichen Fenster gelangen Sie auch, wenn Sie nach dem Markieren der Klasse oben im Menü auf **View → Quick Documentation** gehen.



**Abb. 11.1** Die Docstrings zu einer Klasse anzeigen lassen

Drücken Sie stattdessen **Strg + Shift + I**, öffnet sich ein Pop-up-Fenster mit der gesamten Klassendefinition. Zum Definitionsfenster gelangen Sie auch oben im Menü unter **View → Quick Definition**.

## 11.2 Klassen: die Grundlage der objektorientierten Programmierung



**Abb. 11.2** Die Informationen zu einer Klasse anzeigen lassen.

Aus dem Pop-up-Fenster können Sie die Klassendefinition direkt bearbeiten (markiertes Symbol links) oder im Suchfenster (rechts) öffnen.

Alternativ gelangen Sie in PyCharm direkt zur Klassendefinition, wenn Sie **Strg** gedrückt halten und dann mit dem Cursor über den Klassennamen fahren. Die Definition wird als kurzer Tooltip und die Klasse als Link angezeigt. Wenn Sie auf den Link klicken, können Sie direkt zur Klassendefinition springen. Hier können Sie bei Bedarf auch nachträgliche Änderungen an der Klasse vornehmen.

Diese Funktionalität gilt in PyCharm übrigens nicht nur für Klassen, sondern für alle möglichen Symbole, zum Beispiel auch für Funktionen.

11

Python enthält standardmäßig Docstrings für alle eingebauten Funktionen, Module und Klassen. Es gibt zwei Möglichkeiten, wie Sie sich diese Docstrings beim Programmieren anzeigen lassen können: entweder mit dem `doc`-Attribut oder durch die `help`-Funktion. Vergessen Sie nicht, die Komponenten zunächst zu importieren, damit Python die entsprechende Dokumentation finden kann. Wir betrachten dies einmal anhand des `time`-Moduls:

```

1 import time
2 print(time.__doc__)
3 print("-----\n"
4      "Dokumentation des time-Moduls:\n"
5      "-----\n")
6 help(time)

```

## 11 Objektorientierte Programmierung

Wie Sie sehen, liefert der `help`-Befehl eine weitaus ausführlichere Dokumentation, die darüber hinaus alle Klassen und Funktionen des `time`-Moduls auflistet. Sie können `help` und `__doc__` auch auf Funktionen und Klassen anwenden. Probieren Sie das Gleiche beispielsweise einmal mit der `print`-Funktion!

### 11.3 Objekte: Instanzen der Klassen

Sie haben nun gelernt, wie man eine Klasse erzeugt und die nötigen strukturellen Zerlegungen als Grundgerüst definiert. Doch wie entstehen daraus konkrete Objekte?

Wie mittlerweile bekannt sein dürfte, gehört jedes Objekt einer bestimmten Klasse an. Man nennt ein konkretes Objekt dabei eine *Instanz* oder *Inkarnation* einer Klasse innerhalb des Programmcodes. Die Objekte einer Klasse sind einzigartig, haben jedoch alle dieselbe Struktur, die durch sogenannte *Attribute* ausgedrückt wird. Man kann diese Attribute als charakteristische Merkmale der Klassenobjekte betrachten. Ein Attribut besteht wiederum jeweils aus einem *Attributnamen* und einem *Attributwert*. Die Objekte innerhalb einer Klasse haben aufgrund ihrer Ähnlichkeit also alle dieselben Attributnamen und unterscheiden sich lediglich in den konkreten Werten, die diese annehmen können. So können zwei unterschiedliche Stifte in unserem Schreibwarenladen etwa unterschiedliche Preise haben, obwohl der sie bezeichnende Attributname `preis` derselbe ist, da alle Stifte derselben Klasse angehören.

Wenn man ein Objekt *instanziert*, so erteilt man Python den Befehl, eine Instanz aus der Klasse zu erzeugen, die die vorgegebenen Eigenschaften erfüllt. Die Erzeugung des konkreten Objekts und die Zuordnung der einzelnen Werte geschieht dabei durch den Konstruktor, mithilfe dessen wir zuvor die Struktur der Klasse festgelegt hatten. Nach dem Instanziieren lässt sich die Instanz sodann innerhalb des Programms verwenden.

Zunächst wählt man für jede Instanz einen eigenen Namen. Wir können für ein konkretes Stiftmodell im Schreibwarengeschäft etwa die Bezeichnung `stift1` wählen. Diese neue Instanz setzt man dem Klassennamen gleich, der in Klammern die konkreten Argumente enthält, die an den Konstruktor übergeben werden sollen. Wir können also in das Hauptprogramm unter unsere `Stift`-Klassendefinition schreiben:

#### Codebeispiel #1

```
1 stift1 = Stift("Pelikan", 206532329, 2.99)
```

Womöglich mag Ihnen dabei aufgefallen sein, dass der Konstruktormethode hier nur drei anstatt von ursprünglich vier Werten übergeben wurden. Der Parameter `self`, der in der Definition des Konstruktors enthalten war, wird automatisch übergeben und muss hier nicht erneut explizit aufgeführt werden. Bei der Initialisierung einer

### 11.3 Objekte: Instanzen der Klassen

neuen Instanz muss die Anzahl der in der Klammer enthaltenen Parameter also genau der Anzahl der in der Klassendefinition ursprünglich angegebenen Parameter entsprechen, `self` ausgenommen. Stimmen die Anzahlen nicht überein, gibt das Programm eine Fehlermeldung aus.

Wie in Kapitel 5.5 erwähnt, können Sie sich den Typ, d. h. die übergeordnete Klasse eines beliebigen Objekts, in Python mit dem Befehl `type` anzeigen lassen. Die Eingabe `print(type(stift1))` liefert damit die Ausgabe `<class '__main__.Stift'>`. Das soeben erzeugte Objekt `stift1` gehört also zur Klasse `Stift`, die wir im vorigen Kapitelabschnitt definiert hatten. Versuchen Sie den `type`-Befehl einmal zum Vergleich mit der eingebauten Funktion `print` oder einem Modul wie `math`. Bei Letzterem dürfen Sie das Importieren nicht vergessen.

Nach dem oben angegebenen Schema lassen sich außerhalb der Klassendefinition nun beliebig viele Instanzen von `Stift` erzeugen. Wir möchten daher noch zwei weitere Stiftmodelle hinzufügen:

```
1 stift2 = Stift("Lamy", 101789478, 8.99)
2 stift3 = Stift("Parker", 465510036, 36.00)
```

Wenn Sie nun außerhalb der Klasse auf einzelne Attributwerte der drei Stiftmodelle zugreifen möchten, so müssen Sie hierfür den Stiftnamen gefolgt von einem Punkt und dem gewünschten Attributnamen angeben:

```
1 print(stift1.marke, stift1.artikelnummer, stift1.preis)
2 print(stift1.preis, stift2.preis, stift3.preis)
```

Hier werden auf den ersten `print`-Befehl hin alle Informationen zu `stift1` angezeigt, während der Output der zweiten Zeile die Preise der drei Stifte auflistet. Sie können sich nach diesem Schema beliebige Attributwerte aller Instanzen einer Klasse anzeigen lassen. Vergleichen Sie zusätzlich die folgenden Befehle:

```
1 print("Die Stifte kosten %g €." % (stift1.preis + stift2.preis + stift3.
2 preis))
3 print("Stift2 ist von %s und kostet %g €." % (stift2.marke, stift2.preis))
```

11

Wir verwenden hier `%g` statt `%f` für die Fließkommazahlen, da Python sonst standardmäßig sechs Nachkommastellen anzeigt.

## 11 Objektorientierte Programmierung

The screenshot shows the PyCharm IDE interface. The top bar displays the title 'Kapitel 11 (C:\Users\Sarah\PycharmProjects\Kapitel 11) ... \beispiel\_1.py [Kapitel 11] - PyCharm'. Below the title is a menu bar with File, Edit, View, Navigate, Code, Befactor, Run, Tools, VCS, Window, Help. The left sidebar has sections for Project, Favorites, and Structure. The main area shows a file tree for 'Kapitel 11' with files like '\_init\_.py', 'beispiel\_1.py', 'beispiel\_2.py', etc. A code editor window is open for 'beispiel\_1.py' with the following content:

```
1 class Stift:
2     schreibgeraet = True # dies ist hier optional
3     """Erstellt das Objekt Stift für einen Schreibwarenladen."""
4     def __init__(self, marke, artnr, preis):
5         """
6             Initialisiert ein neues Objekt Stift
7
8             Argumente:
9                 > marke (string): Marke des Stifts
10                > artikelnummer (int): Artikelnummer des Stifts
11                > preis (float): Verkaufspreis des Stifts
12
13
14         self.marke = marke
15         self.artikelnummer = artnr
```

The bottom status bar shows the path 'C:\Users\Sarah\PycharmProjects\Kapitel 11\venv\Scripts\python.exe' and the command 'C:/Users/Sarah/PycharmProjects/Kapitel 11\beispiel\_1.py'. The bottom right corner has an 'Event Log' tab.

**Abb. 11.3** Die Ausgabe der Attributwerte

Wie Sie sehen, folgt die Erzeugung der einzelnen Instanzen einem intuitiv leicht verständlichen Schema. In den letzten beiden Zeilen sehen Sie sogar bereits verschiedene Möglichkeiten, wie Sie einzelne Werte ausgeben oder damit zusätzliche Operationen durchführen können.

Bevor wir uns in 11.4. dem Thema Methoden widmen, wollen wir an dieser Stelle das bisher Gelernte in einem vollständigen Code zusammenfassen, mit dem wir in den folgenden Abschnitten weiterarbeiten werden. Die beiden letzten `print`-Befehle für einzelne Wertausgaben werden wir hierfür nicht benötigen.

**Zur Erinnerung:** Zunächst führten wir unsere Stift-Klasse ein, darauf folgte die Konstruktormethode mit den gewünschten Attributen sowie die Dokumentation durch Docstrings. Anschließend erzeugten wir im Hauptprogramm drei einzelne Instanzen von Stift. Der dazugehörige Code lautet:

```
1 class Stift:
2     schreibgeraet = True # dies ist hier optional
3     """Erstellt das Objekt Stift für einen Schreibwarenladen."""
4     def __init__(self, marke, artnr, preis):
5         """
6             Initialisiert ein neues Objekt Stift
7
8             Argumente:
9                 * marke (string): Marke des Stifts
```

## 11.3 Objekte: Instanzen der Klassen

```

10     * artikelnummer (int): Artikelnummer des Stifts
11     * preis (float): Verkaufspreis des Stifts
12     """
13
14     self.marke = marke
15     self.artikelnummer = artnr
16     self.preis = preis
17
18 # Hauptprogramm
19 stift1 = Stift("Pelikan", 206532329, 2.99)
20 stift2 = Stift("Lamy" 101789478, 8.99)
21 stift3 = Stift("Parker", 465510036, 36.00)

```

Bemerken Sie zuletzt auch die beiden möglichen Zugriffsoptionen auf das Klassenattribut `schreibgeraet`, die beide die Ausgabe True liefern:

```

1 print(Stift.schreibgeraet)
2 print(stift1.schreibgeraet)

```

Da Klassenattribute für alle Instanzen gleichermaßen gelten, kann der Zugriff entweder über den Klassen- oder über den Instanznamen erfolgen. Vorsicht ist jedoch bei Änderungen von Klassenattributwerten geboten: Diese sollten immer über den Klassennamen vorgenommen werden, da es sonst leicht zu uneinheitlichen Wertzuschreibungen für unterschiedliche Instanzen kommen kann. Wir werden im nächsten Abschnitt genauer auf die beiden möglichen Definitionsebenen – Klasse und Instanz – für Variablen bzw. Attribute eingehen.

### 11.3.1 Objekt- und Klassenattribute

Aus früheren Kapiteln sind Ihnen Variablen als Behälter für Informationen ganz unterschiedlicher Art bekannt. Auch bei Attributen handelt es sich eigentlich um nichts Anderes als um Variablen, die unterschiedliche Datentypen mit konkreten Werten aufnehmen können. Wie bereits erwähnt, gibt es innerhalb von Klassen zwei verschiedene Definitionsebenen für diese Variablen: Sie können entweder nur für eine bestimmte Instanz gelten oder direkter Bestandteil der gesamten Klasse sein. Im ersten Fall nennt man sie *Objektattribute* (auch *Instanzattribute*), im zweiten *Klassenattribute*. Da wir Objektattribute bereits in den vorigen Abschnitten behandelt haben, werden wir nun genauer auf den Gebrauch von Klassenattributen eingehen.

11

**Zur Erinnerung:** Objektattribute stehen innerhalb der Konstruktormethode. Sie werden mit `self` gefolgt von einem Punkt und dem Variablennamen aufgerufen, etwa: `self.marke`. Sie haben bereits die Objektattribute `marke`, `artikelnummer` und `preis` kennen gelernt. Die verschiedenen Werte, die diese Variablen für unterschiedliche Objekte wie `stift1`, `stift2` und `stift3` annahmen, waren dabei völlig unabhängig voneinander.

## 11 Objektorientierte Programmierung

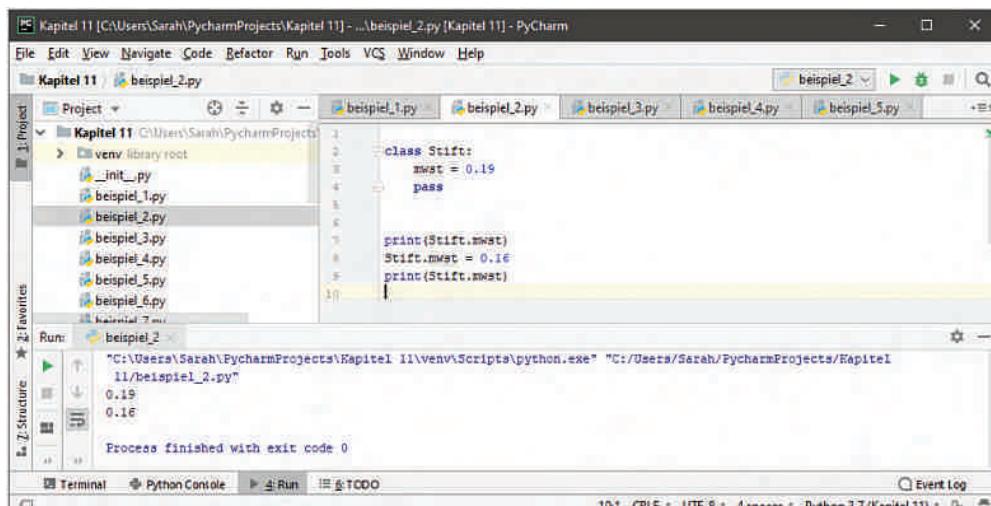
Die Werte der Klassenattribute gelten hingegen für die gesamte Klasse und für alle Instanzen, die diese enthält. Klassenattribute können global verwendet und überschrieben (engl: *override*) werden, ohne dass es hierfür konkrete Objekte geben muss. Dies birgt gleichzeitig ihr größtes Sicherheitsrisiko: Klassenattribute lassen sich aus dem Hauptprogramm heraus ändern. Das wird im folgenden Beispiel deutlich, in dem wir die Mehrwertsteuer als Klassenattribut definieren, dem wir sodann im Hauptprogramm einen neuen Wert zuweisen:

### Codebeispiel #2

```

1  class Stift:
2      mwst = 0.19
3      pass
4
5  print(Stift.mwst)
6  Stift.mwst = 0.16
7  print(Stift.mwst)
8

```



**Abb. 11.4** Die Änderung eines Klassenattributs

Auf Klassenattribute wird also Bezug genommen, indem man den Namen der Klasse gefolgt von einem Punkt und dem Attributnamen angibt, etwa: `Stift.mwst`. Wie Sie bereits aus dem letzten Abschnitt des vorigen Teilkapitels 11.3. wissen, lässt sich im Falle der Existenz einer konkreten Instanz – etwa: `stift1` – ebenfalls durch `stift1.mwst` auf Klassenattribute zugreifen. Wollen Sie den Wert eines Klassenattributs jedoch nicht nur lesen oder verwenden, sondern auch abändern, sollte der Zugriff stets über die Klasse und nicht über einzelne Instanzen erfolgen. Würden Sie im obigen Beispiel etwa den Mehrwertsteuersatz durch `stift1.mwst = 0.16` ändern, so hätte dies lediglich Auswirkung auf die Mehrwertsteuer von `stift1` und nicht auf

### 11.3 Objekte: Instanzen der Klassen

alle anderen Stiftobjekte. Nur durch die Eingabe `Stift.mwst = 0.16` gilt die gewünschte Änderung für alle Objekte. Die beste Option ist selbstverständlich, den Wert direkt in der Klasse anzupassen.

Klassenattribute können ebenfalls innerhalb von Methoden definiert werden. Wir werden nun eine weitere nützliche Verwendung von Klassenattributen innerhalb der Konstruktormethode kennenlernen.

Stellen Sie sich hierfür vor, dass Sie als Schreibwarenhändlerin in Ihrem Programm ab speichern möchten, wie viele unterschiedliche Stiftmodelle sich in Ihrem Stiftbestand befinden. Hierfür können Sie als Klassenattribut eine neue Variable für die Anzahl der Stiftmodelle definieren. Das Klassenattribut `anzahl` wird zunächst oberhalb des Kons truktors auf null gesetzt und erhöht sich anschließend jedes Mal automatisch um eins, wenn innerhalb der Konstruktormethode ein neues `Stift`-Objekt instanziert wird.

Wie bereits erwähnt, lassen sich über `Stift.anzahl` Änderungen an der Variablen `anzahl` vornehmen (schreibender Zugriff). Diese Änderungen gelten dann für alle Objekte und nicht nur für das aktuelle. Um sich den Wert der Variablen anzeigen zu lassen (lesender Zugriff), kann man entweder den Klassennamen `Stift` oder den Namen eines beliebigen Objekts, etwa `stift1`, nach dem Schema `Stift.anzahl` bzw. `stift1.anzahl` aufrufen.

#### Codebeispiel #3

```

1  class Stift:
2      schreibgeraet = True # dies ist hier optional
3      """Erstellt das Objekt Stift für einen Schreibwarenladen."""
4      anzahl = 0
5      def __init__(self, marke, artnr, preis):
6          """
7              Initialisiert ein neues Objekt Stift
8
9          Argumente:
10             * marke (string): Marke des Stifts
11             * artikelnummer (int): Artikelnummer des Stifts
12             * preis (float): Verkaufspreis des Stifts
13
14
15             self.marke = marke
16             self.artikelnummer = artnr
17             self.preis = preis
18
19             Stift.anzahl += 1

```

11

Beachten Sie insbesondere die letzte Zeile, in der die Variable `anzahl` innerhalb der Konstruktormethode bei jedem neuen Instanziieren eines Objekts um eins erhöht wird. Die Schreibwarenhändlerin fügt ihrer Klasse nun nach und nach Objekte hinzu:

## 11 Objektorientierte Programmierung

```

1 print(Stift.anzahl)
2 stift1 = Stift("Pelikan", 206532329, 2.99)
3 print(stift1.anzahl)
4 stift2 = Stift("Lamy", 101789478, 8.99)
5 print(Stift.anzahl)
6 stift3 = Stift("Parker", 465510036, 36.00)
7 print(stift3.anzahl)

```

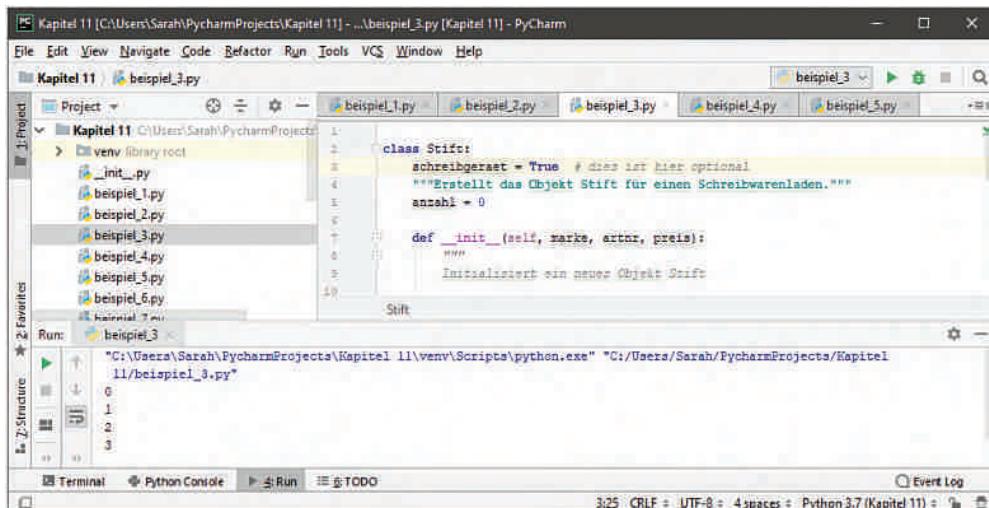


Abb. 11.5 Die automatische Änderung des Klassenattributs `anzahl`

### 11.4 Methoden: Funktionen für Objekte

Klassen sind zusätzlich zu Attributen mit Methoden ausgestattet. In der objektorientierten Programmierung werden Attribute und Methoden einer Klasse unter dem Begriff *Member* (engl. für *Mitglied*) zusammengefasst. Bei Methoden handelt es sich im Prinzip um nichts Anderes als Funktionen innerhalb einer `class`-Definition. Genauso wie Attribute gehören Methoden also immer einer bestimmten Klasse an und stehen nach den Attributen innerhalb der Klasse. Methoden können mit globalen oder lokalen Variablen (vgl. Abschnitt 9.2.) arbeiten und berechnete Werte durch `return` an das Hauptprogramm zurückgeben.

Dabei gibt es verschiedene Arten von Methoden: solche, die unabhängig von einzelnen Instanzen operieren können und solche, die sich auf Instanzen in der Klasse beziehen. Erstere werden *statische Methoden* oder *Klassenmethoden*, letztere *Instanzmethoden* genannt. Im Folgenden werden wir nicht auf den Unterschied zwischen statischen und Klassenmethoden eingehen, sondern lediglich statische Methoden und Instanzmethoden behandeln.

### 11.4.1 Statische Methoden

Wie bei Klassenattributen (bzw. statischen Attributen) handelt es sich auch bei *statischen Methoden* um *statische Member* einer Klasse. Sie benötigen keinerlei Instanzen aus der Klasse und nehmen daher nur Argumente auf bzw. geben nur Werte an das Hauptprogramm zurück, die nicht von Instanzen stammen. Aus diesem Grund benötigen sie auch keinen `self`-Parameter. Statische Methoden können sogar für eine Klasse definiert werden, ohne dass diese überhaupt konkrete Objekte enthält. Sie sollten jedoch durch den Vermerk `@staticmethod` oberhalb der Methodendefinition als solche definiert werden.

Stellen Sie sich beispielsweise vor, dass das Bestandsprogramm der Schreibwarenhändlerin jedes Mal eine Begrüßungsformel ausgibt, wenn es aufgerufen wird. Hierfür würde der folgende vereinfachte Code genügen:

#### Codebeispiel #4

```
1 class Stift:
2     @staticmethod
3     def begruessung():
4         print("Guten Tag, Arne!")
```

Durch `Stift.begruessung()` können Sie diese Methode ganz einfach im Hauptprogramm verwenden und sich die Begrüßung ausgeben lassen. Wie Sie sehen, wird hierfür keine konkrete Instanz benötigt.

Einer statischen Methode können zwar Argumente übergeben werden, jedoch dürfen diese nicht von einer Instanz kommen. Fügen Sie etwa den folgenden Code in die Klassendefinition ein, können zur Begrüßung auch Namen anderer Mitarbeiter gewählt werden:

```
1     @staticmethod
2     def begruessung_name(name):
3         print("Guten Tag, %s!" % name)
4
5 Stift.begruessung()    # Hauptprogramm
6 Stift.begruessung_name("Rita")
```

## 11 Objektorientierte Programmierung

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top indicates 'Kapitel 11 [C:\Users\Sarah\PycharmProjects\Kapitel 11] - ...\\beispiel\_4.py [Kapitel 11] - PyCharm'. Below it is a menu bar with File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help. A toolbar follows with icons for file operations. The main area is divided into several panes: 'Project' (listing files like \_\_init\_\_.py, beispiel\_1.py, etc.), 'Run' (showing the command run), and 'Terminal' (displaying the execution output). The code editor pane contains the following Python code:

```

1 class Stift:
2     @staticmethod
3     def begruessung():
4         print("Guten Tag, Arne!")
5
6     @staticmethod
7     def begruessung_name(name):
8         print("Guten Tag, " + name)

```

The 'Run' pane shows the command: "C:\Users\Sarah\PycharmProjects\Kapitel 11\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 11/beispiel\_4.py". The output terminal pane shows:

```

Guten Tag, Arne!
Guten Tag, Rita!
Process finished with exit code 0

```

**Abb. 11.6** Die Ausgabe der beiden statischen Methoden

Wie Sie sehen, ähnelt der Gebrauch statischer Methoden sehr der Verwendung von Funktionen, wie wir sie bereits aus Kapitel 9 kennen. Funktionen können zudem durch die Funktion `staticmethod()` als statische Methode an eine bestimmte Klasse gebunden werden.

### 11.4.2 Instanzmethoden

Um eine Instanzmethode verwenden zu können, müssen Sie bereits eine konkrete Instanz erzeugt haben. Es ist dabei Usus, Methodennamen mit kleingeschriebenen Verben im Imperativ zu beginnen, zum Beispiel `berechne_mwst`. Damit Python weiß, dass sich die Methode auf Instanzen innerhalb der Klasse bezieht, müssen Sie auch hierbei immer das Schlüsselwort `self` verwenden. Diese Referenz auf die Instanz wird als erster, obligatorischer Parameter in der Funktionsklammer angegeben und bei jedem erneuten Aufruf der Methode automatisch auf die entsprechende Instanz angewandt, ohne dass man den Parameter dabei jedes Mal explizit nennen muss. Um sich für jeden Preis eines Stifts die enthaltene Mehrwertsteuer anzeigen zu lassen, könnte die Schreibwarenhändlerin etwa die folgende Methode innerhalb der `Stift`-Klassendefinition verwenden:

#### Codebeispiel #5

```

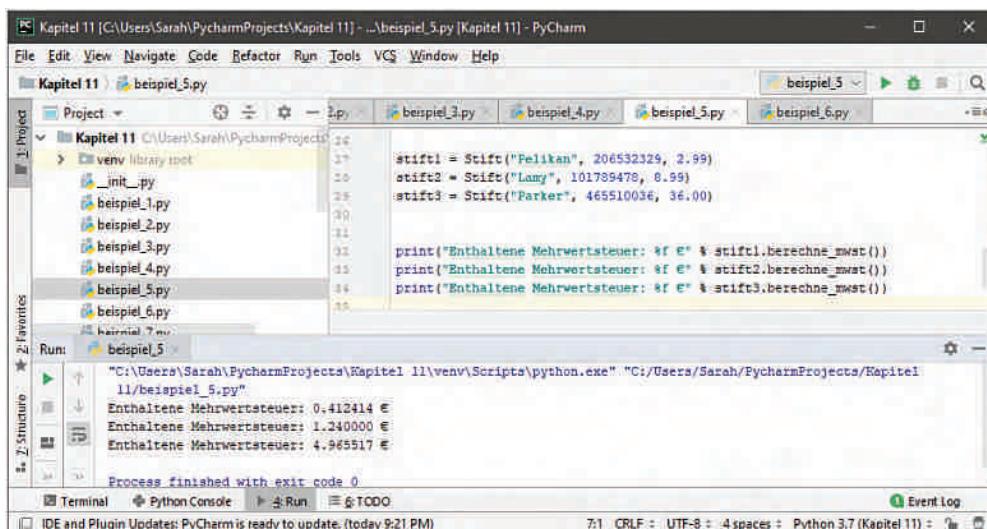
1 def berechne_mwst(self):
2     ohne_mwst = self.preis/1.16
3     betrag_mwst = self.preis - ohne_mwst
4     return betrag_mwst

```

## 11.4 Methoden: Funktionen für Objekte

Zusammen mit der Angabe unserer drei Stiftinstanzen im Hauptprogramm lässt sich der jeweilige Mehrwertsteuerbetrag sodann folgendermaßen ausgeben:

```
1 print("Enthaltene Mehrwertsteuer: %f €" % stift1.berechne_mwst())
2 print("Enthaltene Mehrwertsteuer: %f €" % stift2.berechne_mwst())
3 print("Enthaltene Mehrwertsteuer: %f €" % stift3.berechne_mwst())
```



**Abb. 11.7** Die Anzeige einer Instanzmethode

Wie bei der ursprünglichen Festlegung der Attribute geschieht der Zugriff auf bestimmte Attribute von innerhalb der Klasse durch `self.attributname`, hier etwa `self.preis`. Bemerken Sie hierbei auch den Unterschied zum Zugriff auf Attribute von außerhalb der Klasse: Dieser wird durch `instanzname.attributname` bewerkstelligt, also beispielsweise `stift1.preis`.

Wir möchten Ihnen die Funktionsweise von Instanzmethoden anhand eines weiteren kurzen Beispiels veranschaulichen, in dem die Preise zweier Stifte addiert werden. Hierbei werden der `plus`-Methode zwei Parameter übergeben:

### Codebeispiel #6

```
1 def plus(self, wert):
2     summe = self.preis + wert
3     return summe
4
5 # Hauptprogramm
6 print("Preis insgesamt: %g €" % (stift1.plus(stift3.preis)))
7 print("Preis insgesamt: %g €" % (stift1.plus(5.99)))
```

## 11 Objektorientierte Programmierung

```

Kapitel 11 [C:\Users\Sarah\PycharmProjects\Kapitel 11] - ..\beispiel_6.py [Kapitel 11] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 11 > beispiel_6.py
Project 2.p... beispiel_3.py beispiel_4.py beispiel_5.py beispiel_6.py
Kapitel 11 C:\Users\Sarah\PycharmProjects\Kapitel 11\venv\library root...
__init__.py
beispiel_1.py
beispiel_2.py
beispiel_3.py
beispiel_4.py
beispiel_5.py
beispiel_6.py
heinrich_7.nu...
Run beispiel_6
"C:\Users\Sarah\PycharmProjects\Kapitel 11\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 11/beispiel_6.py"
Preis insgesamt: 38.99 €
Preis insgesamt: 0.98 €

Process finished with exit code 0

```

**Abb. 11.8** Beispiel einer Instanzmethode zur Summenberechnung

Wie Sie sehen, sind die Gestaltungsmöglichkeiten für Instanzmethoden sehr vielfältig und bieten großen Spielraum für fortgeschrittene algorithmische Berechnungen.

### 11.4.3 Die dir-Funktion

Auch die in 11.2. behandelte Konstruktormethode `__init__` war eigentlich nichts anderes als eine Methode. Hierbei handelt es sich jedoch um eine eingebaute Methode, die beim Erzeugen von Objekten aufgerufen wird und die Python zum Initialisieren der Objektattribute benötigt. Sie können sich alle standardmäßig eingebauten Methoden und Attribute für eine Klasse oder ein spezifisches Objekt mithilfe der Funktion `dir()` anzeigen lassen. Hierbei handelt es sich um eine sehr nützliche Funktion, die Sie nicht nur für Klassen, sondern auch allgemein für alle möglichen Python-Objekte – Funktionen, Module, Dictionaries, Listen, Strings, etc. – verwenden können, um sich eine Liste ihrer Attribute und Methoden anzeigen zu lassen.

Wir wollen `dir` nun auf `Stift` und `stift1` anwenden:

#### Codebeispiel #7

```

1 from Beispiel_1 import Stift
2
3 stift1 = Stift("Pelikan", 206532329, 2.99)
4 print("\nAttribute und Methoden der Klasse Stift: ", dir(Stift))
5 print("\nAttribute und Methoden des Objekts stift1: ", dir(stift1))

```

## 11.4 Methoden: Funktionen für Objekte

```

Kapitel 11 [C:\Users\Sarah\PycharmProjects\Kapitel 11] - ...\\beispiel_7.py [Kapitel 11] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 11 beispiel_7.py
Project beispiel_1.py beispiel_2.py beispiel_3.py beispiel_4.py beispiel_5.py beispiel_6.py beispiel_7.py
Run: beispiel_7
Attribute und Methoden der Klasse Stift: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'schreibgeraet']

Attribute und Methoden des Objekts stift1: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'artikelnummer', 'marke', 'preis', 'schreibgeraet']

```

**Abb. 11.9** Mit `dir` Attribute und Methoden anzeigen lassen

Um zu verdeutlichen, wie das Importieren von Klassen funktioniert, haben wir an dieser Stelle nochmals den `import`-Befehl wiederholt. Unsere Importdatei heißt `beispiel_1.py`. Nach diesem Schema lassen sich auch einzelne Methoden sowie zuvor definierte Instanzen importieren.

Im Output wird jeweils eine Liste derjenigen Attribute und Methoden angezeigt, die der Klasse `Stift` bzw. dem Objekt `stift1` zur Verfügung stehen. Die Klasse `Stift` hält neben `__init__` und anderen eingebauten Methoden nun auch das von uns definierte Klassenattribut `schreibgeraet` bereit. Dem Objekt `stift1` sind ebenfalls das Klassenattribut `schreibgeraet` sowie die Instanzattribute `artikelnummer`, `marke` und `preis` zugeordnet. Wenn Sie in Ihrem Code noch zusätzliche Methoden definiert haben, werden diese hier ebenfalls sowohl für die Klasse als auch das Objekt angezeigt.

Abgesehen von unseren selbst definierten Attributen und Methoden finden sich noch viele weitere eingebaute Methoden, die Klassen bzw. Objekten immer zur Verfügung stehen. Diese Methoden werden wir im nächsten Unterkapitel genauer untersuchen.

11

### 11.4.4 Magische Methoden

Sie mögen sich zu Recht gefragt haben, was es genau mit den mysteriösen Methoden mit jeweils zwei vorangestellten und zwei angehängten Unterstrichen auf sich hat, die uns die obige Ausgabe der `dir`-Funktion anzeigte. Diese sogenannten *magischen Methoden* (auch *Dunder-Methoden* genannt, für engl. **double under**) werden nicht direkt, sondern implizit im Hintergrund ausgeführt, wenn sie hierfür einen ent-

## 11 Objektorientierte Programmierung

sprechenden Befehl aus dem Hauptprogramm erhalten. Dafür müssen sie zuvor als Methoden innerhalb einer Klasse definiert worden sein. Der Befehl, der anschließend im Hauptprogramm auf sie verweist, hat dabei meist eine sehr einfache Form. Mehr Magie als im Namen steckt also wirklich nicht dahinter!

Um zu sehen, wie das Ganze funktioniert, wollen wir genauer auf zwei der oben aufgelisteten eingebauten Klassenmethoden eingehen. Hierbei sind vor allem die magischen Methoden `__add__` und `__string__` zu nennen. Die magische Methode `__string__` gibt dabei für jedes Objekt aus der `Stift`-Klasse eine Zeichenkette zurück. In unserem Fall definieren wir hierfür einfach eine Zusammenfassung der Informationen zu jedem Stift, die anschließend durch den Befehl `print(Objekt)`, also beispielsweise `print(stift1)` ausgegeben werden kann.

Zudem möchten wir die `__add__`-Methode definieren, die uns schlichtweg die Summe der Preise zweier Stifte liefern soll. Das Argument, das den Preis des zweiten Stifts bezeichnet, wird hier üblicherweise `other` genannt.

### Codebeispiel #8

```

1  class Stift:
2      """Erstellt das Objekt Stift für einen Schreibwarenladen."""
3      def __init__(self, marke, artnr, preis):
4          """
5              Initialisiert ein neues Objekt Stift
6
7              Argumente:
8                  * marke (string): Marke des Stifts
9                  * artikelnummer (int): Artikelnummer des Stifts
10                 * preis (float): Verkaufspreis des Stifts
11
12
13             self.marke = marke
14             self.artikelnummer = artnr
15             self.preis = preis
16
17         def __add__(self, other):
18             return self.preis + other.preis
19
20         def __str__(self):
21             return "Stiftdaten: Marke %s, Artikelnummer %s, Preis %g €" \
22             % (self.marke, self.artikelnummer, self.preis)
23
24     # Hauptprogramm
25     stift1 = Stift("Pelikan", 206532329, 2.99)
26     stift2 = Stift("Lamy", 101789478, 8.99)
27
28     print(stift1)
29     print(stift1 + stift2)

```

## 11.4 Methoden: Funktionen für Objekte

```

Kapitel 11 [C:\Users\Sarah\PycharmProjects\Kapitel 11] - ...\\beispiel_8.py [Kapitel 11] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 11 beispiel_8.py
Project 1 Project 2 Favorites 2: Studie
beispiel_7.py
beispiel_8.py
beispiel_9.py
beispiel_10.py
beispiel_11.py
beispiel_12.py
beispiel_13.py
beispiel_14.py
beispiel_15.py
beispiel_16.py
Übung_1.py
Run: beispiel_8
"C:\Users\Sarah\PycharmProjects\Kapitel 11\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 11/beispiel_8.py"
Stiftdaten: Marke Pelikan, Artikelnummer 206532329, Preis 2.99 €
Stiftdaten: Marke Lamy, Artikelnummer 101785478, Preis 8.99 €
11.98
Event Log
IDE and Plugin Updates: PyCharm is ready to update. (today 9:21 PM) 30:1 CRLF: UTF-8: 4 spaces: Python 3.7 (Kapitel 11)

```

Abb. 11.10 Magische Methoden in Klassen verwenden

Beachten Sie insbesondere, wie wir die letzten beiden Zeilen des Hauptprogramms formuliert haben. Ohne diese magischen Methoden innerhalb der Klassendefinition hätten wir für diese beiden Befehle nicht den gewünschten Output erhalten. Ein Befehl der Form `stift1 + stift2` hätte sogar zu einer Fehlermeldung geführt. Durch die Verwendung von `__add__` als Instanzmethode lassen sich nun im Hauptprogramm ganz leicht die Preise zweier Stifte addieren.

Auch, wenn wir in Python eine Addition zweier Zahlen, etwa `7 + 2`, vornehmen, passiert im Grunde nichts anderes, als dass im Hintergrund ebenfalls die `__add__`-Methode aufgerufen wird. Wie Sie wissen, lässt sich der `+`-Operator in Python für Zahlen, Zeichenketten, Listen und viele andere Datentypen einsetzen. Er sorgt dafür, dass mehrere Objekte eines Typs addiert bzw. zu einem neuen Objekt desselben Typs konkateniert werden können. Dabei bleibt der ursprüngliche Datentyp (bzw. der Objekttyp) erhalten. Man spricht in diesem Zusammenhang auch von einer *Überladung* des Additionsoperators (`+`). Auch Vergleichsoperatoren wie `<` und `>` sowie binäre und viele weitere Operatoren lassen sich in diesem Sinne überladen.

Die Funktionalität der magischen Methoden und die damit verbundene Möglichkeit der Operatorenüberladung ist in vielen Fällen äußerst praktisch und lässt sich in zahlreichen technischen Variationen verwenden. Hierbei handelt es sich um einen Spezialfall von *Polymorphie*, auf die wir in 11.6. nochmals zurückkommen werden.

## 11 Objektorientierte Programmierung

### 11.5 Datenkapselung

Nachdem Sie nun wissen, was eine Methode ist, wollen wir weitere Aspekte untersuchen, die für Attribute und Methoden relevant sind. Attributwerte und Methoden einer Klasse lassen sich standardmäßig bei Bedarf von außerhalb der Klasse lesen und abändern. Betrachten Sie dazu den folgenden Code, in dem die Marke eines Stiftmodells nachträglich überschrieben wird:

#### Codebeispiel #9

```
1 stift1 = Stift("Pelikan", 206532329, 2.99)
2 stift1.marke = "edding"
3 print(stift1.marke)
```

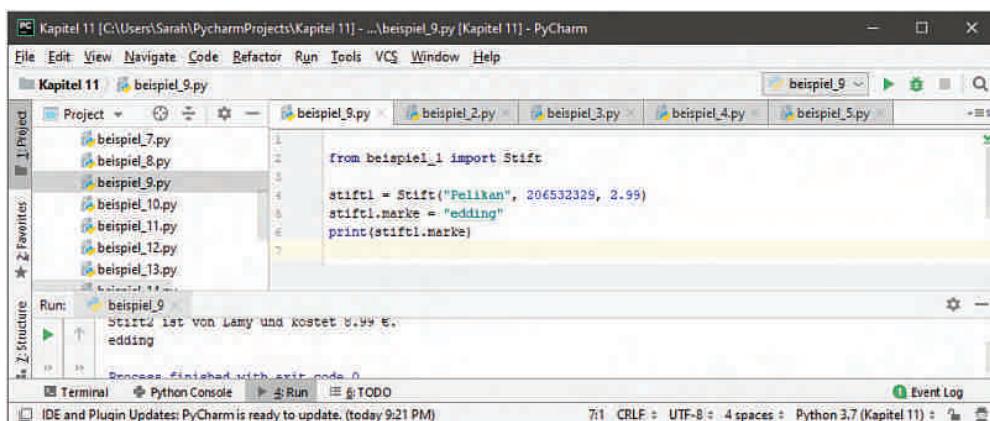


Abb. 11.11 Überschreiben von Attributen

Wie Sie sehen, lässt sich die Marke des Stifts auf diese Weise ganz einfach ändern. Manchmal möchte man jedoch nicht, dass Attribute änderbar oder überhaupt sichtbar sein sollen. So wäre es in einem Programm für ein Schreibwarengeschäft eigentlich eher unerwünscht, wenn sich feste Attributwerte wie die Marke eines bestimmten Artikels so einfach überschreiben ließen. Denkbar wäre hingegen, dass sich die Artikelnummer in seltenen Fällen ändert, wohingegen der Preis eines Modells mit größerer Wahrscheinlichkeit Änderungen unterliegen wird.

Python bietet die Option, vorab die gewünschten Zugriffsarten für Attribute festzulegen. Dadurch lassen sich etwa die Einsicht in geheime Attributwerte, nachträgliche Manipulationen oder versehentliche Änderungen von außen erschweren. Für die drei oben genannten Möglichkeiten – unerlaubte Änderungen, Änderungen bei besonderem Bedarf sowie unbeschränkte Änderungen – gibt es in Python drei verschiedene Kategorien zum Schutz der Attribute. Im ersten Fall unerlaubter Änderungen wird der direkte Zugriff auf die Daten von außen unterbunden. Sie sind dann weder sichtbar noch benutzbar – ein direkter Zugriff ist nur innerhalb der jeweiligen Klasse möglich.

Man nennt dieses Verhindern eines lesenden oder verändernden Zugriffs von außen *Datenkapselung*. Die drei Zugriffsarten unterscheiden sich formal in der Schreibweise der Variablen innerhalb der Konstruktormethode: Zwei dem Variablennamen vorangestellte Unterstriche bedeuten hierbei, dass es sich um ein sogenanntes *privates Member* handelt, ein Unterstrich bezeichnet ein *geschütztes Member* und kein Unterstrich gibt wie in den bisher behandelten Fällen an, dass wir es mit einem *öffentlichen Member* zu tun haben, das geändert werden darf. Ein *Member* (engl. für *Mitglied*) kann dabei entweder eine Methode oder ein Attribut innerhalb der Klasse bezeichnen. Im folgenden Code sind unsere drei gewählten Stiftattribute mit den drei unterschiedlichen Zugriffsoptionen versehen:

### Codebeispiel #10

```
1 class Stift:
2     schreibgeraet = True # dies ist hier optional
3     def __init__(self, marke, artnr, preis):
4         self.__marke = marke
5         self._artikelnummer = artnr
6         self.preis = preis
```

Wenn wir nun versuchen, außerhalb der Klasse auf `stift1` zuzugreifen, gibt Python einen sogenannten `AttributeError` für die Marke aus. Preis und Artikelnummer werden jedoch angezeigt:

```
1 stift1 = Stift("Pelikan", 206532329, 2.99)
2 print(stift1._artikelnummer, stift1.preis)
3 print(stift1.__marke)
```

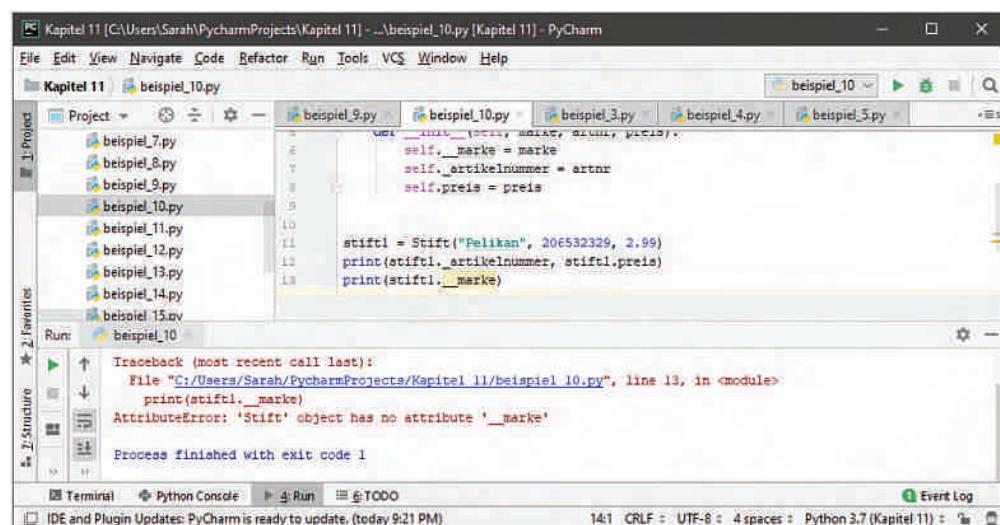


Abb. 11.12 Fehler bei der Ausgabe eines privaten Attributs

## 11 Objektorientierte Programmierung

Das private Attribut `stift1.__marke` ist nun vor externem Zugriff – also von außerhalb der Klasse – geschützt und nach außen hin nicht mehr sichtbar. Anhand der Fehleranzeige wird ersichtlich, dass der Interpreter mit dem Attribut so umgeht, als existiere es nicht. In den anderen beiden Fällen wird die entsprechende Information jedoch angezeigt. Diese unterscheiden sich in der Hinsicht, dass durch einen Unterstrich markierte, geschützte Member vielmehr eine Empfehlung an andere mit dem Code arbeitende Personen darstellen, dass von außen kein direkter Zugriff vorgenommen werden sollte. Öffentliche Member hingegen können von außen gelesen und überschrieben werden. Ein weiterer Vorteil der Kapselung durch einen oder zwei Unterstriche ist zudem, dass der Interpreter eine Fehlermeldung ausgibt, wenn etwa anstatt des Attributs `_marke` versehentlich das Attribut `marke` aufgerufen wird. Die Kapselung stellt also einen zusätzlichen Schutzmechanismus vor ungewollten Änderungen dar.

Auf diese Weise lassen sich übrigens auch Klassenattribute und Methoden – allgemein: alle Member – kennzeichnen. An dieser Stelle sei jedoch erwähnt, dass Python im Vergleich zu anderen objektorientierten Programmiersprachen wie etwa Java bezüglich Datenkapselung einer weniger restriktiven Philosophie folgt. Selbst im Falle privater Member gibt es Wege, von deren Existenz zu wissen, deren Werte sichtbar zu machen oder abzuändern. Als „privat“ gekennzeichnete Member werden somit vielmehr versteckt als gänzlich unzugänglich gemacht. Die Unterteilung in private, geschützte und öffentliche Member hat in Python daher eher den Charakter einer Empfehlung für andere Programmierende, als dass sie eine strikte Regel darstellt.

Wie nun bekannt sein dürfte, gibt es sowohl *lesenden* als auch *schreibenden* Zugriff auf Attributwerte. Standardmäßig wird in Python für diese zwei Zugriffsalternativen jeweils eine Methode definiert. Die Methode, die den Wert eines Attributs liest bzw. abfragt, wird *Getter* oder auch *Abfragemethode* genannt. Die Methode, mithilfe derer man den Wert überschreiben bzw. ändern kann, heißt *Setter* oder auch *Änderungsmethode*. Im Folgenden werden wir beide Methoden behandeln.

### 11.5.1 Getter und Setter

Da die Getter- und Setter-Methoden innerhalb einer Klasse stehen, haben sie vollen Zugriff auf die Daten innerhalb der Klasse – selbst, wenn diese gekapselt sind. Aus diesem Grund kann man die beiden Zugriffsoperationen wie ein Verbindungsstück zwischen den Attributen und den Programmteilen außerhalb der Klasse einsetzen, die optional einen kontrollierten Zugriff auf private Attribute ermöglichen können.

Mithilfe der Getter-Methode lässt sich der Wert eines Attributs abfragen. Obwohl der Methodenname wie üblich frei wählbar ist, wählt man hierfür in der Regel das Wort `get` (engl. für *holen*) gefolgt vom Attributnamen – in unserem Fall etwa `getmarke`. Diese Methode benötigt keinen Übergabewert, sondern – wie jede Instanzmethode

innerhalb einer Klasse – lediglich den Ausdruck `self`. Wir fügen also den folgenden Code in unsere Stift-Klassendefinition ein:

#### Codebeispiel #11

```
1 def getmarke(self):
2     """Gibt die Marke des Stifts zurück."""
3     return self.__marke
```

Nun können wir die Marke eines Stifts im Hauptprogramm mit `getmarke()` auslesen, obwohl wir den direkten lesenden Zugriff ursprünglich durch die Kapselung `__marke` unterbunden hatten:

```
1 stift2 = Stift("Lamy", 101789478, 8.99)
2 print(stift2.getmarke())
```

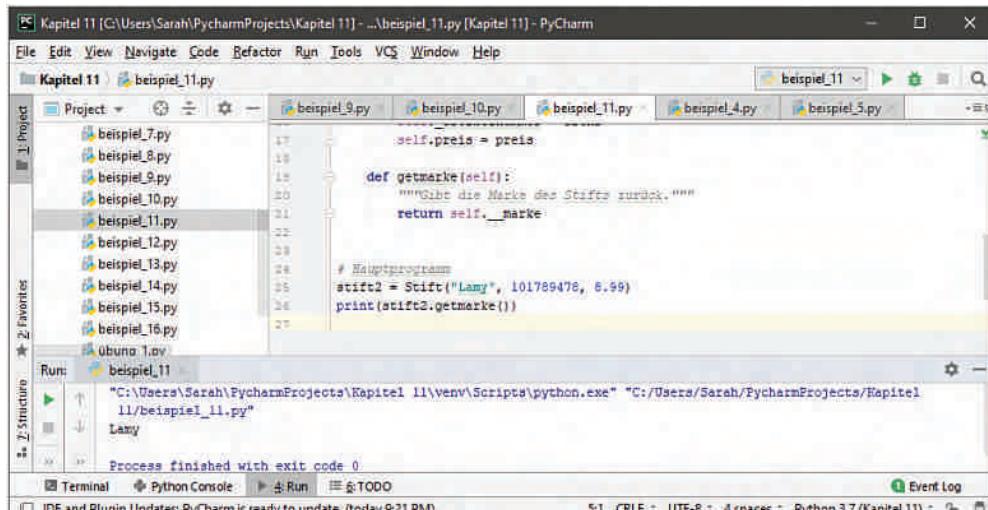


Abb. 11.13 Die Ausgabe des `__marke`-Werts durch `getmarke`

Nach diesem Schema lassen sich nach Wunsch `get`-Methoden für weitere Attribute aufsetzen.

Mit der Setter-Methode lässt sich der Wert eines Attributs verändern. Man wählt hier analog die Bezeichnung `set` (engl. für *setzen*) gefolgt von dem gewünschten Attributnamen. In unserem Fall wäre dies etwa `setpreis`, da wir möchten, dass der Preis eines Stifts leicht änderbar sein soll. Der neue Preis wird durch einen zusätzlichen Übergabewert, `preis_neu`, festgelegt. Wir möchten nun konkret den Preis von `stift1` um einen Euro auf 9,99 € ergänzen die Klassendefinition um den folgenden Code:

## 11 Objektorientierte Programmierung

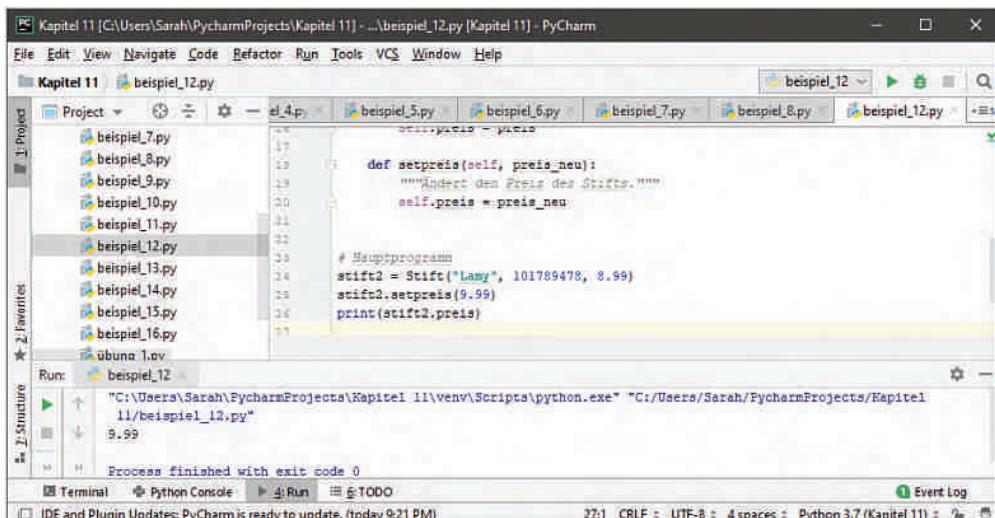
### Codebeispiel #12

```
1 def setpreis(self, preis_neu):
2     """Ändert den Preis des Stifts."""
3     self.preis = preis_neu
```

Anschließend schreiben wir in unserem Hauptprogramm:

```
1 stift2 = Stift("Lamy", 101789478, 8.99)
2 stift2.setpreis(9.99)
3 print(stift2.preis)
```

Der Preis wird somit geändert.



**Abb. 11.14** Die Änderung des `preis`-Werts durch `setpreis`

Der Vorteil der Setter- und Getter-Methoden als Zugriffsoperationen liegt hauptsächlich darin, dass sie größere Flexibilität und Gestaltungsmöglichkeiten beim Programmieren und Arbeiten mit Klassendaten erlauben. So besteht zwar allgemein die Option, Attribute bei Bedarf durch Ausdrücke wie `print(stift1.marke)` oder `stift1.marke = "Lamy"` zu lesen bzw. zu überschreiben. Da es sich bei Methoden aber um Funktionen handelt, lassen sich darin jedoch weitaus mehr Optionen für die Bearbeitung der Daten realisieren.

Haben Sie beispielsweise eine Idee, wie Sie mit den Getter-und-Setter-Methoden allzu große Preisänderungen eines Stifts verhindern können? Betrachten Sie hierzu den folgenden Code, in dem das soeben Gelernte nochmals zusammengefasst ist:

## Codebeispiel #13

```

1  class Stift:
2      schreibgeraet = True # dies ist hier optional
3      """Erstellt das Objekt Stift für einen Schreibwarenladen."""
4      def __init__(self, marke, artnr, preis):
5          """
6              Initialisiert ein neues Objekt Stift
7
8              Argumente:
9              * marke (string): Marke des Stifts
10             * artikelnummer (int): Artikelnummer des Stifts
11             * preis (float): Verkaufspreis des Stifts
12
13
14             self.__marke = marke
15             self.__artikelnummer = artnr
16             self.__preis = preis
17
18     def getpreis(self):
19         """
20             Gibt den Preis des Stifts zurück.
21         """
22         return self.__preis
23
24     def setpreis(self, preis_neu):
25         """
26             Ändert den Preis des Stifts.
27         """
28         if abs(self.__preis - preis_neu) < self.__preis*0.50:
29             self.__preis = preis_neu
30         else:
31             print("Die Abweichung zum vorherigen Preis ist sehr groß.")
32             bestaetigung = input("Sind Sie sicher, dass %g € als neuer"
33                                 " Preis gelten soll? (ja/nein) " % preis_neu)
34             if bestaetigung == "ja":
35                 self.__preis = preis_neu
36
37
38 # Hauptprogramm
39 stift1 = Stift("Pelikan", 206532329, 2.99)
40 x = float(input("Geben Sie den neuen Preis ein: "))
41 stift1.setpreis(x)
42 print("Neuer Preis [€]:", stift1.getpreis())

```

## 11 Objektorientierte Programmierung

The screenshot shows the PyCharm IDE interface with the following details:

- Title Bar:** Kapitel 11 [C:\Users\Sarah\PycharmProjects\Kapitel 11] - ...\\beispiel\_13.py [Kapitel 11] - PyCharm
- Menu Bar:** File Edit View Navigate Code Refactor Run Tools VCS Window Help
- Toolbar:** beispiel\_13
- Project View:** Shows files like beispiel\_7.py, beispiel\_8.py, etc., and a file named beispiel\_13.py which is currently selected.
- Code Editor:** Displays Python code for a pen price adjustment. It includes an if-block for confirmation and a main program loop for input and output.
- Run Tab:** Shows the command "C:\Users\Sarah\PycharmProjects\Kapitel 11\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 11/beispiel\_13.py". The output window shows user interaction and a result message.
- Status Bar:** IDE and Plugin Updates: PyCharm is ready to update. (today 9:21 PM)

Abb. 11.15 Spezifische Preisänderung und -ausgabe mit Getter- und Settermethoden

Wir haben hier zwei verschachtelte `if`-Schleifen verwendet, um innerhalb der `setpreis`-Methode mehrere Eingabemöglichkeiten zu berücksichtigen. Der `abs`-Befehl gibt dabei den Absolutbetrag der Abweichung des Neupreises zum bestehenden Wert aus. Preisabweichungen von über 50 Prozent geben in unserem Beispiel eine Sondermeldung aus.

Aufmerksame Lesende mögen hier zudem bemerkt haben, dass das Attribut `__preis` in diesem Codebeispiel zwar zu einem privaten gemacht wurde, die Änderung und Ausgabe über die `setpreis`- und `getpreis`-Methoden in den letzten beiden Zeilen jedoch dennoch möglich ist. Dies spielt insbesondere bei Programmiersprachen mit strikter Datenkapselung eine größere Rolle. Ein weiterer wichtiger Vorteil dieser beiden Zugriffsmethoden in Hinblick auf Datenkapselung besteht darin, dass man Objekte von innerhalb einer Klasse modifizieren kann, ohne dass dabei der externe Code im Hauptprogramm angepasst werden muss.

Getter- und Settermethoden werden zwar nicht nur bei gekapselten Daten verwendet, aus den genannten Gründen hier jedoch besonders gerne eingesetzt. Darüber hinaus gilt es allgemein als guter Programmierstil, jegliche Zugriffsoperationen auf Klassenattribute nicht direkt, sondern über Getter und Setter vorzunehmen.

Wenn Sie mehr über Datenkapselungsoptionen in Python erfahren möchten, können Sie zusätzlich über die `property`-Funktion nachlesen.

## 11.6 Vererbung: Ein wichtiges Prinzip der OOP

### 11.6 Vererbung: Ein wichtiges Prinzip der OOP

Ein wichtiges Prinzip beim objektorientierten Programmieren ist das der *Vererbung*. Hier werden Eigenschaften und Methoden einer sogenannten *Oberklasse* (auch: Eltern- oder Basisklasse) bei der Definition einer neuen *Unterklasse* (auch: abgeleitete, Kind- oder Subklasse) automatisch auf diese übertragen. Die abgeleitete Unterklasse *erbt* sozusagen die Struktur der übergeordneten Oberklasse. Nachdem eine Unterklasse angelegt wurde, lassen sich darin im Anschluss Attribute und Methoden der Oberklasse durch Überschreiben oder Hinzufügen leichter anpassen und gegebenenfalls erweitern.

Auf diese Weise können Sie sehr schnell und einfach neue Klassen erzeugen, ohne dass Sie dabei mit großem Aufwand die gesamte Klasse komplett neu schreiben müssen. Durch die Wiederverwendung des Codes werden also zusätzliche Arbeit und Codezeilen eingespart.

Verfügt die Schreibwarenhändlerin über ein sehr großes Stiftsortiment, kann es beispielsweise vorteilhaft sein, eine weitere Unterkategorie für unterschiedliche Stifttypen wie Kugelschreiber, Bleistifte, etc. anzulegen. Die Vererbung der Member aus der *Stift*-Klasse lässt sich hierbei realisieren, indem man die ursprüngliche Oberklasse hinter den Namen der neuen Klasse *Kugelschreiber* in Klammern schreibt:

```
1 class Kugelschreiber(Stift):
```

In der neuen *Kugelschreiber*-Unterklassie lassen sich anschließend zusätzlich zu den geerbten auch neue Attribute und Methoden angeben. Wir wollen dies testen, indem wir den *Kugelschreibern* das zusätzliche Attribut *farbe* zuschreiben. Dazu müssen wir zunächst wie bei jeder Klassendefinition die Konstruktormethode unserer Unterklassie unter den Definitionskopf schreiben. In der nächsten Zeile wiederholen wir die Konstruktormethode der Oberklasse und stellen ihr den Oberklassennamen mit einem Punkt voran. Alternativ zu dieser Schreibweise lässt sich auch die eingebaupte Funktion *super()* verwenden. Dabei wird automatisch nach der entsprechenden Oberklasse gesucht und diese verwendet. Unter die Konstruktorzeile für die *Stift*-Klasse schreiben wir unser neues Attribut *farbe*.

11

#### Codebeispiel #14

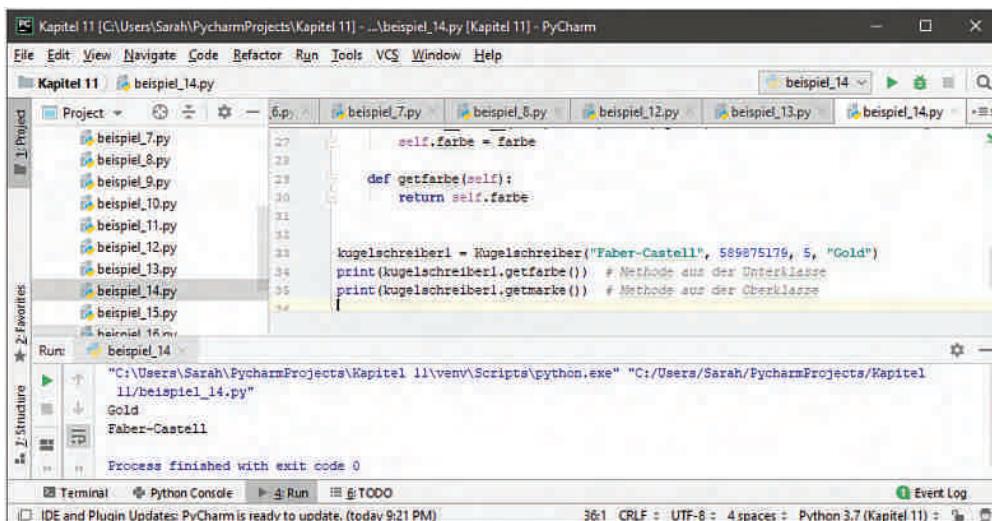
```
1 class Kugelschreiber(Stift):
2     """Erstellt das Objekt Kugelschreiber als Unterklasse von Stift."""
3     def __init__(self, marke, artnr, preis, farbe):
4         Stift.__init__(self, marke, artnr, preis) # Alternative zu Stift:
5         super()
6         self.farbe = farbe
7
8     def getfarbe(self):
9         return self.farbe
```

## 11 Objektorientierte Programmierung

```

10
11 kugelschreiber1 = Kugelschreiber("Faber-Castell", 589875179, 5, "Gold")
12 print(kugelschreiber1.getfarbe()) # Methode aus der Unterklasse
13 print(kugelschreiber1.getmarke()) # Methode aus der Oberklasse

```



**Abb. 11.16** Vererbung an die Stift-Unterklasse Kugelschreiber

Wie Sie sehen, bleiben bei diesem Ansatz insbesondere Änderungen an Ober- und Unterklassen übersichtlicher, als wenn wir sie separat definiert hätten. Dies ist umso nützlicher, je größer die Klassendefinition ist. Dabei wirken sich Änderungen in der Oberklasse ebenfalls direkt auf alle Unterklassen aus, während Änderungen innerhalb einer Unterklasse keinerlei Folgen für andere Klassen haben. Anhand der letzten Zeile wird deutlich, dass die Methode `getmarke` aus der Oberklasse automatisch verwendet wird, obwohl sie in der Definition der Unterklasse nicht explizit vorhanden ist.

Wenn Sie in einer Unterklasse eine Methode definieren, die denselben Namen hat wie eine Methode der Oberklasse, so gilt für alle Instanzen der Unterklasse automatisch nur die neu definierte Methode der Unterklasse. Ihr Inhalt muss dabei nicht identisch mit dem der Oberklassenmethode sein. Somit lassen sich Methoden der Oberklasse „überschreiben“ – genau genommen sucht Python zunächst immer die Unterklassen nach vorhandenen Methoden für Unterklassenobjekte ab. Gibt es keine, wird in der Oberklasse nach der entsprechenden Methode gesucht. Wir wollen dies anhand zweier gleichnamiger Methoden für die Ober- und Unterklasse verdeutlichen:

### Codebeispiel #15

```

1 # Methode für die Oberklasse Stift
2     def zeige_preis(self):
3         print("Der Preis des Stifts beträgt: %g €" % self.preis)
4
5 # Methode für die Unterklasse Kugelschreiber

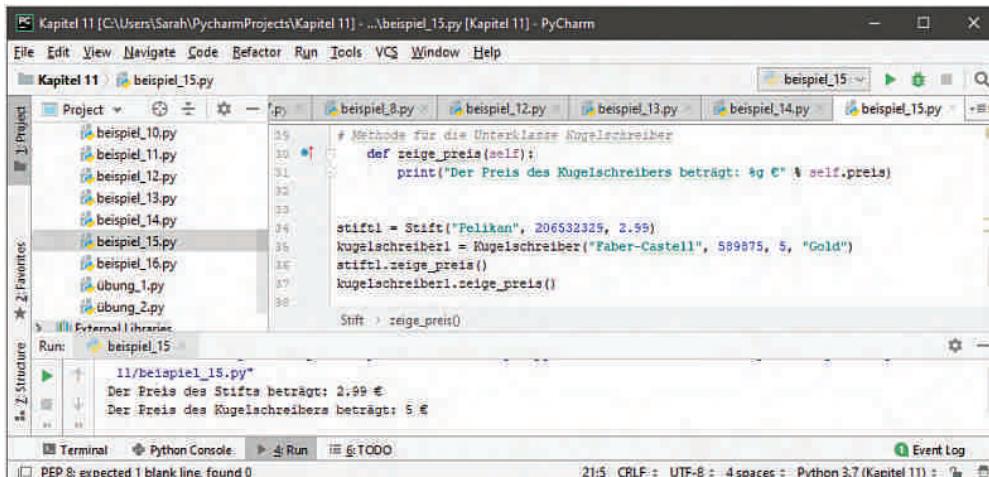
```

## 11.6 Vererbung: Ein wichtiges Prinzip der OOP

```

6     def zeige_preis(self):
7         print("Der Preis des Kugelschreibers beträgt: %g €" % self.preis)
8
9 # Hauptprogramm
10 stift1 = Stift("Pelikan", 206532329, 2.99)
11 kugelschreiber1 = Kugelschreiber("Faber-Castell", 589875179, 5, "Gold")
12 stift1.zeige_preis()
13 kugelschreiber1.zeige_preis()

```



**Abb. 11.17** Zwei gleichnamige Methoden für unterschiedliche Objekte

Beachten Sie dabei, dass Sie die beiden Methoden jeweils in der Ober- bzw. Unterklassendefinition angeben müssen. Anhand des Outputs sehen Sie, dass automatisch die passende `zeige_preis`-Methode für die beiden unterschiedlichen Objekte aus den beiden Klassen aufgerufen wird.

Diese Möglichkeit, ein und dieselbe Methode auf verschiedene Objekttypen anzuwenden, wird *Polymorphie* (auch: *Polymorphismus*) genannt. Dieser Begriff kommt aus dem Griechischen und bedeutet *Vielgestaltigkeit*. In der objektorientierten Programmierung ist damit gemeint, dass gleichnamige Operationen (bzw. Methoden) auf Objekte unterschiedlicher Klassen angewandt werden können. Auch in 11.4.4. hatten wir es beim Überladen von Operatoren mithilfe von magischen Methoden mit einer Form von Polymorphie zu tun.

11

Betrachten Sie nun das folgende interessante Beispiel für Polymorphismus innerhalb von Klassenmethoden, in dem deutlich wird, dass ein und derselbe Methodenaufruf automatisch in der richtigen Klasse erfolgt. Dabei definieren wir jeweils eine `Stift`- und eine `Kugelschreiber`-Instanz und legen diese gemeinsam in einer Liste an, auf deren Elementen wir `zeige_preis` anwenden:

```

1 s = stift1
2 k = kugelschreiber1
3 stiffe = [s, k]

```

## 11 Objektorientierte Programmierung

```
4   for stift in stifte:
5       stift.zeige_preis()
```

Der Output ist hierbei derselbe wie derjenige in der Abbildung des vorigen Beispiels.

In Python ist es möglich, an eine Unterklasse die Charakteristika mehrerer Klassen zu vererben. Hierzu müssen die einzelnen Oberklassen durch Kommata getrennt in der Klammer des Definitionskopfes aufgelistet werden. Für unser Beispiel wollen wir etwa annehmen, dass der Ladenbestand zusätzlich zu den Klassen Stift, Papier, Schulbedarf, etc. eine separate Klasse Saisonartikel für diejenigen Artikel im Sortiment enthält, die sich innerhalb eines bestimmten Zeitraums mit begrenzter Stückzahl im Angebot befinden. Die Schreibwarenhändlerin möchte nun alle Saisonartikel mit der Stift-Klasse zu einer neuen Klasse kombinieren, die die entsprechenden Angaben für saisonal erhältliche Bleistifte enthält.

Um diese *Mehrfachvererbung* auszudrücken, können Sie nach der Definition der Saisonartikel-Klasse die folgende Definition für die neue Unterklasse Bleistift verwenden:

### Codebeispiel #16

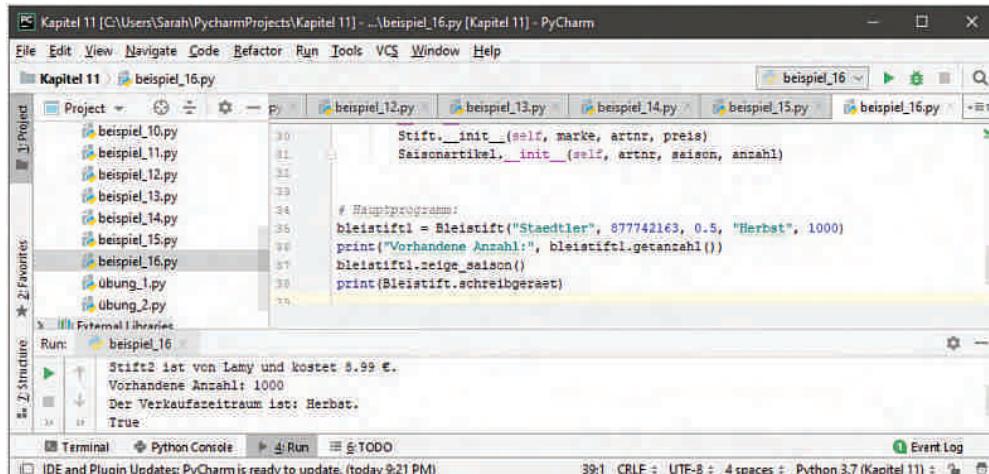
```
1  from beispiel_1 import Stift
2
3  class Saisonartikel:
4      """Erstellt das Objekt Saisonartikel für einen Schreibwarenladen."""
5      def __init__(self, artnr, saison, anzahl):
6          """
7              Initialisiert ein neues Objekt Saisonartikel
8
9              Argumente:
10                 * artnr (int): Artikelnummer des Saisonartikels
11                 * saison (string): Verkaufsperiode des Saisonartikels
12                 * anzahl (int): Bestandsanzahl des Saisonartikels
13
14
15                 self.artikelnummer = artnr
16                 self.saison = saison
17                 self.anzahl = anzahl
18
19             def getanzahl(self):
20                 return self.anzahl
21
22             def zeige_saison(self):
23                 print("Der Verkaufszeitraum ist: %s." % self.saison)
24
25 class Bleistift(Stift, Saisonartikel): # Hier geschieht die Mehrfachvererbung
26     def __init__(self, marke, artnr, preis, saison, anzahl):
27         Stift.__init__(self, marke, artnr, preis)
28         Saisonartikel.__init__(self, artnr, saison, anzahl)
```

## 11.6 Vererbung: Ein wichtiges Prinzip der OOP

```

30 # Hauptprogramm:
31 bleistift1 = Bleistift("Staedtler", 877742163, 0.5, "Herbst", 1000)
32 print("Vorhandene Anzahl:", bleistift1.getanzahl())
33 bleistift1.zeige_saison()
34 print(Bleistift.schreibgeraet)

```



**Abb. 11.18** Mehrfachvererbung aus zwei Klassen

Im Hauptprogramm werden nach dem Erzeugen von `bleistift1` die beiden Methoden aus `Saisonartikel` verwendet. Das Beispiel verdeutlicht zudem, wie sich Ausgaben durch `print`-Befehle entweder im Hauptprogramm oder innerhalb einer Methode realisieren lassen. Anhand der letzten Zeile wird ersichtlich, dass es sich bei dem Bleistift um ein Schreibgerät handelt. In der Tat gilt das Klassenattribut `schreibgeraet` der Klasse `Stift` auch für alle Bleistifte: Die Ausgabe ist `True`.

Ein interessanter Aspekt ist übrigens, dass man nicht nur aus selbst definierten Klassen, sondern auch aus Standardklassen wie `int`, `bool`, `list`, `dict`, etc. neue Unterklassen mit spezifischeren Eigenschaften und Methoden ableiten kann.

Wie in diesem Kapitel deutlich geworden sein dürfte, ist Objektorientierung das Herzstück des Programmierens mit Python. Wir hoffen, Ihnen hiermit einen verständlichen Überblick über die vielen Möglichkeiten der OOP vermittelt zu haben. Es konnten bei Weitem jedoch nicht alle Themen der objektorientierten Programmierung in Python behandelt werden. Mehr über Klassen finden Sie beispielsweise unter <https://docs.python.org/3/tutorial/classes.html>.

Nachdem wir das recht abstrakte Thema Objektorientierung hinter uns gelassen haben, möchten wir uns im nächsten Kapitel etwas Praktischerem widmen: Fehlern. Zunächst werden Sie jedoch in den folgenden Übungsaufgaben die Chance haben, Ihr neues Wissen über Klassen, Objekte und Methoden anzuwenden.

## 11 Objektorientierte Programmierung

### 11.7 Übung: Mit Objekten arbeiten

#### Übungsaufgaben

- Denken Sie an Kapitel 6 zurück. Dort hatten wir Datenstrukturen für eine Bibliothek erzeugt, die unter anderem die Angaben autor\_in, titel, verlag und standort aufwiesen. Legen Sie nun eine komplette Werk-Klasse mit diesen vier Angaben als Attribute an. Entscheiden Sie, welche der Attribute als privat ausgezeichnet werden sollten. Formulieren Sie jeweils eine Getter- und Settermethode für mindestens eines der Attribute.

Erzeugen Sie danach zwei werk-Objekte und lassen Sie sich die Bestandsanzahl durch ein Klassenattribut ausgeben. Ändern Sie den Standort eines Werks mithilfe der Setter-Methode und lassen Sie sich den neuen Standort anschließend durch die Getter-Methode ausgeben. Lassen Sie sich sodann alle Angaben zu dem Werk durch eine `__str__`-Methode anzeigen.

Erstellen Sie zuletzt eine aus Werk abgeleitete Klasse, die ein weiteres Attribut enthält. Lassen Sie sich schließlich erneut die Bestandsanzahl ausgeben. Machen Sie von einer Methode aus der Oberklasse Gebrauch und lassen Sie sich den Wert des neuen Attributs anzeigen.

- Das folgende Programm bildet Monty Pythons „Cheese“-Sketch in einem Programm nach. Versuchen Sie nachzuvollziehen, was der Code beschreibt.

**Info:** Hier wird die `format`-Methode der String-Klasse für formatierte String-Ausgaben verwendet. Falls Sie sich erinnern: Zuvor hatten wir hierfür vom Operator `%` mit entsprechenden Buchstaben Gebrauch gemacht, um innerhalb von Strings Werte einzufügen. Die `format`-Methode wird jedoch als der wahre pythonische Weg empfohlen. Versuchen Sie, ihre Funktionsweise zu verstehen.

Der Code enthält zudem fünf Fehler. Finden und korrigieren Sie diese.

```

1 class Käse:
2     """Erstellt das Objekt Käse für ein Käsegeschäft."""
3     def __init__(sorte, preis):
4         """
5             Initialisiert ein neues Objekt Käse
6
7             Argumente:
8                 * sorte (string): Käsesorte
9                 * preis (float): Preis der Käsesorte
10            """
11
12         self.sorte = käsesorte
13         self.preis = preis
14
15     käsel = Käse("Camembert", 3.99)

```

## 11.7 Übung: Mit Objekten arbeiten

```
16 print("Verkäufer:\tGuten Morgen!")
17 while True:
18     print("\t\t\tWas darf's sein?")
19     wunsch = str(input("Kunde:\t\tIch hätte gerne "))
20     if wunsch == käsel.sorte:
21         print("Verkäufer:\tJa, {} haben wir! "
22               "Er kostet {} €.\n".format(käsel.sorte, käsel.preis) +
23               "\t\t\tOh, die Katze hat ihn gegessen!")
24     else:
25         print("Verkäufer:\tTut mir leid, {} haben wir nicht!".format(wunsch))
```

## 11 Objektorientierte Programmierung

### Lösungen

1.

```

1  class Werk:
2      bestandwerk = True
3      """Erstellt das Objekt Werk für eine Bibliothek."""
4      anzahl = 0
5
6      def __init__(self, autor_in, titel, verlag, standort):
7          """
8              Initialisiert ein neues Objekt Werk
9
10             Argumente:
11             * autor_in (string): Autor/in des Werks
12             * titel (string): Titel des Werks
13             * verlag (string): Verlag des Werks
14             * standort (string): Standort des Werks
15
16
17             self.__autor_in = autor_in
18             self.__titel = titel
19             self.verlag = verlag
20             self.standort = standort
21
22             Werk.anzahl += 1
23
24     def setstandort(self, standort_neu):
25         """
26             Ändert den Standort des Werks.
27
28             self.standort = standort_neu
29
30     def getstandort(self):
31         """
32             Gibt den Standort des Werks zurück.
33
34             return self.standort
35
36     def __str__(self):
37         return "Das Werk '%s' von %s (Verlag: %s) hat den Standort %s." \
38             % (self.__titel, self.__autor_in, self.verlag, self.standort)
39
40 class Sachbuch(Werk):
41     """Erstellt das Objekt Sachbuch als Unterklasse von Werk."""
42     def __init__(self, autor_in, titel, verlag, standort, themengebiet):
43         Werk.__init__(self, autor_in, titel, verlag, standort)
44         self.thema = themengebiet
45
46 # Hauptprogramm
47 werk1 = Werk("Ferrante, Elena", "Meine geniale Freundin", "Suhrkamp Verlag",
48 "BE 1577")
49 werk2 = Werk("Knausgård, Karl Ove", "Lieben", "btb Verlag", "FE 8974")
50 print(Werk.anzahl)
51 werk2.setstandort("FE 1234")
52 print(werk2.getstandort())
53 print(werk2)

```

## 11.7 Übung: Mit Objekten arbeiten

```

54
55 sachbuch1 = Sachbuch("Schmitt, Sarah", "Python 3", "BMU Verlag", "ST 1341",
56 "Informatik")
57 print(Werk.anzahl) # Klassenattribut aus der Oberklasse
58 print(sachbuch1) # Methode __str__ aus der Oberklasse
59 print(sachbuch1.thema)

```

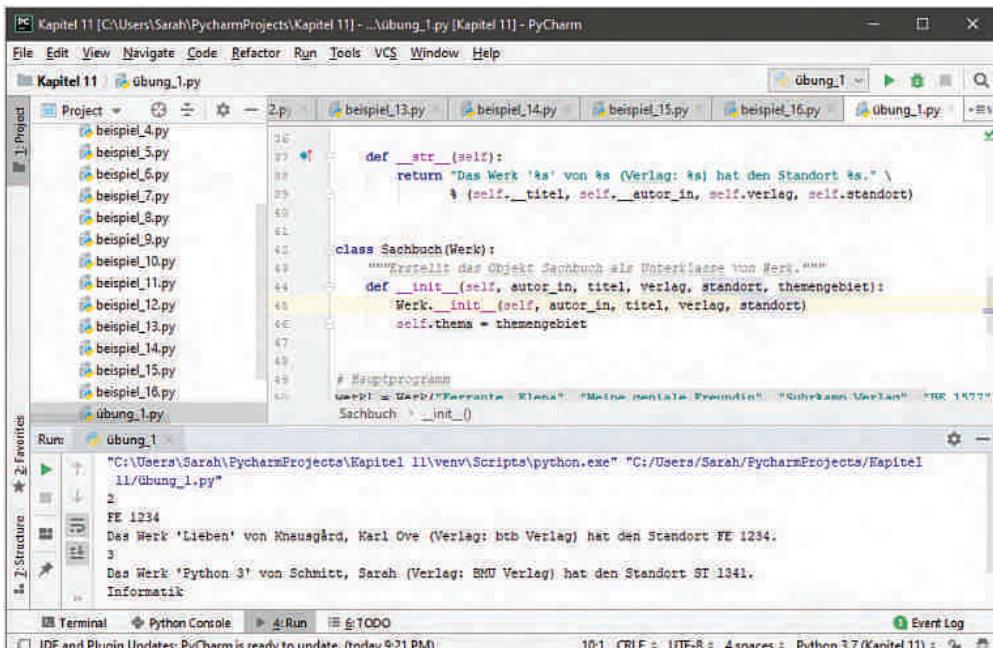


Abb. 11.19 Ein Verwaltungsprogramm für eine Bibliothek erstellen

2. Fehler: 2. Zeile (Docstring " "), 3. Zeile (`self`), 12. Zeile (`sorte`), 21. Zeile (`{ 0 }`), 22. Zeile (`{ 1 }`).

Das Hauptprogramm (Verkäufer) fragt die Anwender (Käufer) durch eine Eingabeaufforderung, welche Käsesorte diese gerne hätten. In der Klasse Käse gibt es jedoch nur ein Objekt käsel von der Sorte Camembert für 3,99 € (Attribute sorte und preis). Nach der Anwendereingabe prüft das Programm, ob der Wunsch des Käufers mit der Sorte von käsel übereinstimmt. Außert der Käufer einen anderen Wunsch als Camembert, erhält er die Auskunft, dass die Sorte nicht vorrätig ist. Bei Camembert erfolgt eine andere Ausgabe. Durch while True stellt der Verkäufer seine Frage immer wieder erneut.

```

1 class Käse:
2     """Erstellt das Objekt Käse für ein Käsegeschäft."""
3     def __init__(self, sorte, preis):
4         """
5             Initialisiert ein neues Objekt Käse
6
7             Argumente:
8                 * sorte (string): Käsesorte

```

## 11 Objektorientierte Programmierung

```

9         * preis (float): Preis der Käsesorte
10        """
11
12        self.sorte = sorte
13        self.preis = preis
14
15
16    käsel = Käse("Camembert", 3.99)
17    print("Verkäufer:\tGuten Morgen!")
18    while True:
19        print("\t\t\tWas darf's sein?")
20        wunsch = str(input("Kunde:\tIch hätte gerne "))
21        if wunsch == käsel.sorte:
22            print("Verkäufer:\tJa, {} haben wir! "
23                  "Er kostet {} €.\n".format(käsel.sorte, käsel.preis) +
24                  "\t\t\tOh, die Katze hat ihn gegessen!")
25        else:
26            print("Verkäufer:\tTut mir leid, {} haben wir nicht!".format(wunsch))

```

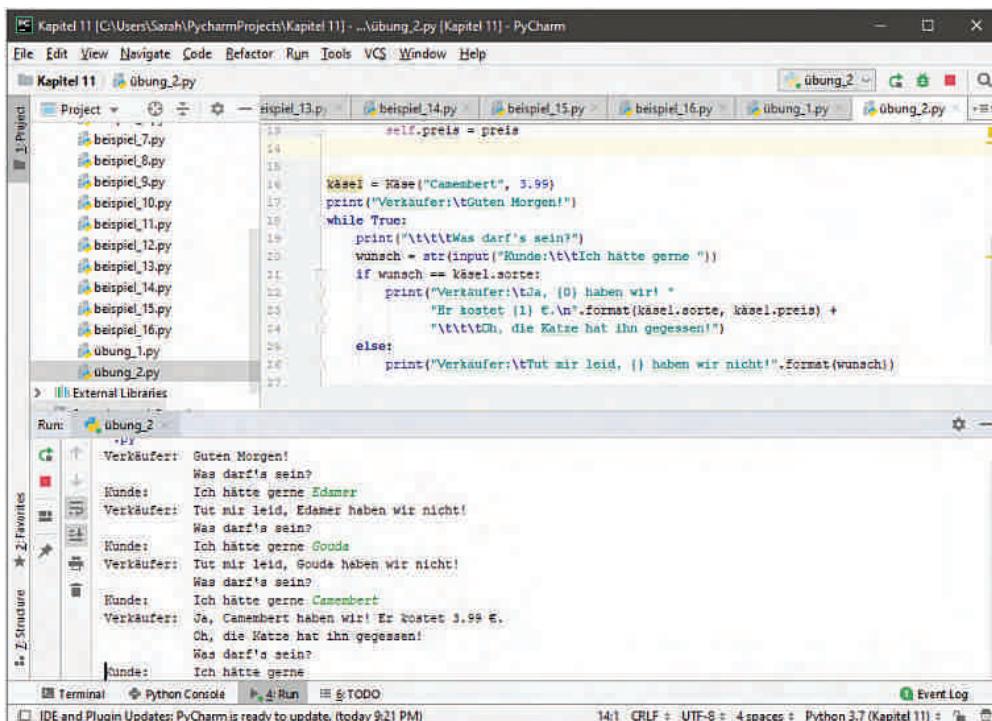


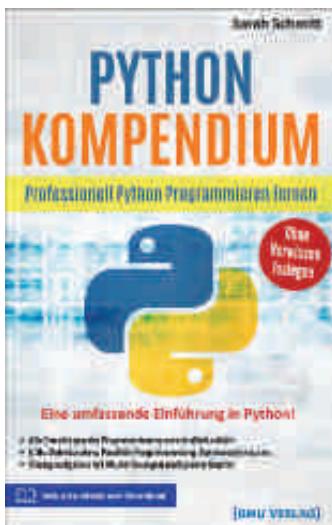
Abb. 11.20 Monty Python's Käse-Sketch als Klasse

Zur Formatierung: Der Befehl \t fügt einen Tabulator ein.

Fortan werden wir vorzugsweise die `format`-Methode zum Formatieren von Strings verwenden. In den folgenden Kapiteln werden wir ihre Funktionsweise in weiteren Anwendungen genauer kennen lernen.

Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 12

# Fehler und Ausnahmen behandeln

Manchmal ist die Sache klar: Sie schreiben ein Programm, das ausnahmslos genau das tut, was Sie möchten. Wie in der menschlichen Kommunikation kann es jedoch auch beim Arbeiten mit Programmiersprachen passieren, dass Ihr Gegenüber nicht versteht, was Sie von ihm wollen. Als Folge können Missverständnisse, Probleme oder gar Fehler auftreten – die dann wiederum zu unerwarteten Programmabrüchen oder sogar -abstürzen führen können.

Programmabrüche, die durch Programmierfehler verursacht wurden, haben oftmals verheerende Folgen. Beispielsweise können dabei wichtige Daten verloren gehen oder – im Falle größerer Programme – andere wirtschaftliche oder gesellschaftliche Schäden entstehen. Für Unternehmen können Prozesse zur nachträglichen Fehlerbehebung und Schadensbegrenzung mit viel Zeitaufwand und dementsprechend hohen Kosten verbunden sein.

Um Fehler im Programm zu vermeiden und robusten Code zu schreiben, sollten Sie sich daher vorher genau überlegen, was Ihr Programm machen soll und wie Sie dieses Ziel am einfachsten erreichen können. Seien Sie sich dessen bewusst, dass Fehler stets passieren und auch noch nach mehreren Überprüfungen unerkannt bleiben können. Perfekter Code existiert nicht – Ausnahmen oder Fehler können jederzeit auftreten. Versuchen Sie deswegen, diese so früh wie möglich zu unterbinden. Ein eindeutiger Code, in dem alle möglichen Fälle und Verzweigungen bedacht wurden und klar formuliert sind, ist hier zielführend.

In den folgenden Abschnitten werden wir genauer untersuchen, welche programmiertechnischen Vorkehrungen Sie im Vorfeld treffen können, um Programmabbrüche zu verhindern. Zunächst wollen wir hierfür einige mögliche Fehlerquellen innerhalb von Programmen betrachten.

### 12.1 Was sind Fehler und Ausnahmen in Python?

Im Bereich der Softwareentwicklung werden Fehler meist *Ausnahmen* (engl. *exceptions*) genannt. Damit wird ausgedrückt, dass sie eben nicht die Regel, sondern die Ausnahme darstellen und nur in seltenen Fällen auftreten. Je umfangreicher und komplexer ein Programm ist, umso höher ist jedoch die Wahrscheinlichkeit, dass sich beim Programmieren unvorhergesehene Fehler einschleichen. Entdeckt der Interpreter einen Fehler im Programm, sagt man, dass eine Ausnahme ausgelöst oder geworfen wird (engl. *throw an exception*). Bei *exceptions* handelt es sich übrigens ebenfalls um

## 12.1 Was sind Fehler und Ausnahmen in Python?

Objekte bestimmter Klassen, die mit Attributen und Methoden ausgestattet sind, die der Klassifizierung und Behandlung von Ausnahmen dienen. Die Klasse `Exception` ist hierbei die Oberklasse, von der alle Unterklassen unterschiedlicher Ausnahmetypen erben.

Um eine Fehlerquelle finden und beheben zu können, muss man sich überlegen, was genau zu einer bestimmten Ausnahme geführt hat und wie sich das Auftreten dieses besonderen Falls verhindern lässt. Dafür muss man den Fehler verstehen. Die Ursachen für Programmfehler können dabei ganz unterschiedlicher Art sein. Fehler können sich etwa gleich am Anfang durch eine entsprechende Meldung bemerkbar machen oder auch längere Zeit unerkannt bleiben – und zum Beispiel erst aufgrund falscher Ergebnisse in einem Berechnungsalgorithmus entdeckt werden.

Die unterschiedlichen Fehlertypen lassen sich meist leicht verschiedenen Kategorien zuordnen. Je besser Sie über die möglichen Fehlertypen Bescheid wissen, umso einfacher wird es sein, Fehler souverän zu handhaben.

- ▶ Lexikalische Fehler sind Zeichenketten, die der Python-Interpreter nicht auswerten kann. Dies können etwa falsch geschriebene Variablen, Befehle oder Funktionen sein, die einen undefinierten Bezeichner darstellen.
- ▶ Syntaxfehler entstehen, wenn die Grammatik der Programmiersprache nicht eingehalten wird – zum Beispiel, wenn Klammern oder Doppelpunkte vergessen werden oder wenn für eine Funktion nicht die richtige Parameteranzahl verwendet wird.

Fehler dieser beiden Kategorien werden meist früh entdeckt, da das Programm ohne deren Behebung oftmals nicht laufen kann. In diesem Zusammenhang spricht man auch von *Kompilierungsfehlern*. In den Übungen von Kapitel 9 und 11 hatten wir etwa lexikalische und syntaktische Fehler kennen gelernt.

Doch selbst programmiertechnisch einwandfreier Code kann während der Ausführung des Programms zu Fehlern führen. Fehler von dieser Art werden *Laufzeitfehler* genannt. Sie können unter anderem durch eine bestimmte Betriebssystemversion oder falsche Eingabedaten ausgelöst werden. Zum Beispiel ist es möglich, dass Nutzer unerwartete Daten in das Programm eingeben, die der Python-Interpreter dann versehentlich weiterverarbeitet, da er keine weiteren Anweisungen zu den gewünschten Datentypen erhalten hat. Denkbar ist auch, dass eine zur Programmausführung benötigte Datei gelöscht wurde, was ebenfalls zu einem Laufzeitfehler, in diesem Fall zu einem sogenannten `IOError`, führt.

12

Laufzeitfehler machen sich in der Regel durch eine Programmunterbrechung, ein entsprechendes Dialogfenster oder falsche Ergebnisse bemerkbar. Man unterscheidet sie in semantische, logische oder Designfehler:

## 12 Fehler und Ausnahmen behandeln

- ▶ Ein semantischer Fehler liegt dann vor, wenn das Programm zwar syntaktisch einwandfrei läuft, aber dennoch inhaltliche Fehler enthält. Eine falsche Parameterreihenfolge oder Schleifen mit verspätetem Abbruch sind etwa Beispiele für semantische Fehler.
- ▶ Logische Fehler bedeuten, dass im Problemlösungsansatz ein Denkfehler vorliegt. Dieser kann beispielsweise darin begründet sein, dass für eine Fallunterscheidung das Zeichen > anstatt < eingegeben wurde. Logische Fehler können jedoch auch auf komplexere Hintergründe wie Fehlschlüsse oder andere fehlerhafte Fallunterscheidungen hinweisen.
- ▶ Im Falle von Designfehlern wurde das Grundkonzept eines Programms nicht korrekt entwickelt. Dies kann zum Beispiel an einem auf falschen Annahmen beruhenden Algorithmus liegen, an Fehlern in der Anforderungsdefinition an das Programm oder auch daran, dass das Programm auf der Grundlage mangelhaften Wissens entwickelt wurde.

Beim Auftreten eines Laufzeitfehlers wird das Programm in der Regel komplett angehalten; Codezeilen, die auf die fehlerhafte Zeile folgen, werden nicht mehr ausgeführt. Untersuchen Sie hierfür einmal, was geschieht, wenn bei einer Nutzereingabe, die eine ganze Zahl erwartet, ein String eingegeben wird:

### Codebeispiel #1

```
1 eingabe = int(input("Geben Sie eine ganze Zahl ein: "))
2 print("Vielen Dank für Ihre Eingabe.")
```

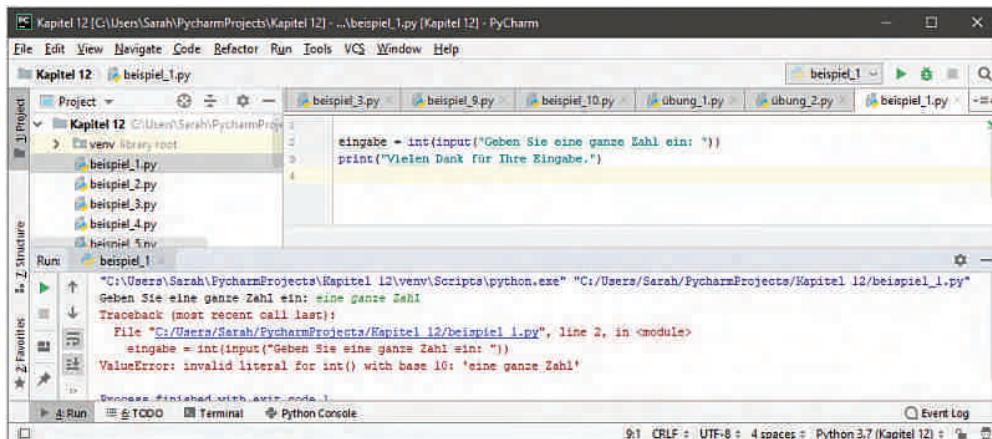


Abb. 12.1 Programmabbruch bei falscher Nutzereingabe

Wie Sie sehen, bricht das Programm beim Auftreten des Fehlers sofort ab. Die nachfolgende `print`-Ausbgabe wird nicht mehr ausgeführt. In der Fehlermeldung ist zudem

## 12.1 Was sind Fehler und Ausnahmen in Python?

angegeben, dass es sich bei diesem Fehler um einen Fehler vom Typ `ValueError` handelt.

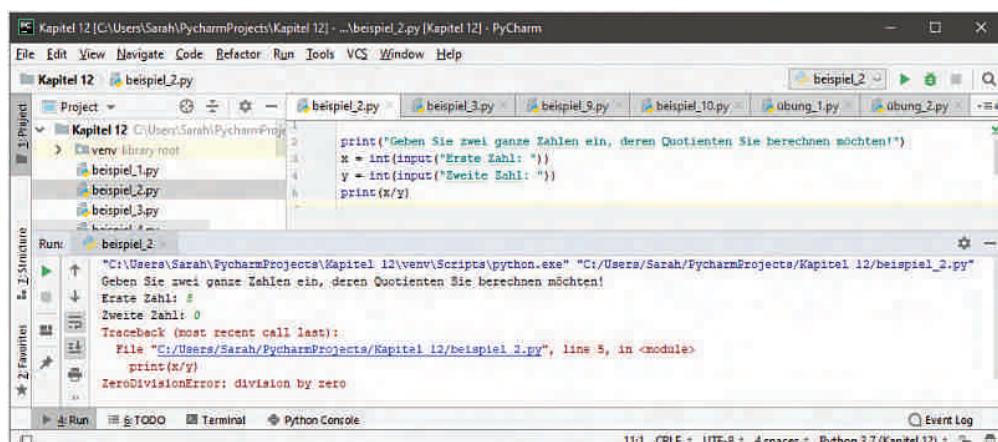
Auch die Operation im folgenden Code liefert eine Fehlermeldung, wenn die Nutzer-  
eingabe für die Variable `y` eine Null ist:

### Codebeispiel #2

```

1 print("Geben Sie zwei ganze Zahlen ein, deren Quotienten Sie berechnen
2 möchten!")
3 x = int(input("Erste Zahl: "))
4 y = int(input("Zweite Zahl: "))
5 print(x/y)

```



**Abb. 12.2** Programmabbruch beim Teilen durch Null

Da die Division durch Null mathematisch nicht möglich ist, kann Python die Berechnung nicht durchführen und gibt zusammen mit dem fehlerhaften Befehl und der entsprechenden Zeilenummer auch die Fehlermeldung `ZeroDivisionError` aus.

Selbstverständlich ist es meistens eher unpraktisch, wenn das gesamte Programm angehalten wird. Besser wäre es, entsprechende Vorkehrungen für unerwünschte Programmabläufe zu treffen, indem man an denjenigen Stellen im Programm, an denen Laufzeitfehler antizipiert werden, stattdessen im Voraus angibt, wie sich das Programm im konkreten Fall verhalten soll. So lassen sich Laufzeitfehler und Fehlermeldungen vermeiden und das Programm kann fortgesetzt werden.

Python verfügt über zwei Kontrollstrukturen, über die sich Programmabbrüche handhaben lassen: `try ... except` und `try ... finally`. Im Folgenden werden wir sie genauer betrachten und untersuchen, wie sich die unterschiedlichen Fehlertypen damit behandeln lassen.

## 12 Fehler und Ausnahmen behandeln

Eine Liste über mögliche Ausnahmen, die bereits in Python eingebaut sind, finden Sie unter <https://docs.python.org/3/library/exceptions.html>.

### 12.2 try ... except: Ausnahmen behandeln

Um eine Ausnahmebehandlung vorzunehmen, muss Python die entsprechende Anweisung hierfür erhalten. In Kapitel 7 hatten Sie bereits eine Möglichkeit kennen gelernt, wie sich verschiedene Verzweigungen im Programm durch `if`, `else` und `elif` angeben lassen. Unsere Fallunterscheidungen galten dabei meist verschiedenen Zahlenwertbereichen. Spezifische Ausnahmefälle, die zu einer Fehlermeldung führen, lassen sich hierdurch nicht handhaben. So kann man mit `if`, `else` oder `elif` etwa keine Nutzereingabe behandeln, die eine ganze Zahl erwartet, stattdessen aber eine Zeichenkette erhält. Python hält jedoch andere Tools bereit, mithilfe derer sich Fallunterscheidungen auf die Behandlung derartiger Ausnahmen ausweiten lassen.

Mit `try ... except` lassen sich unvorhergesehene Programmabbrüche für verschiedene Ausnahmefälle verhindern. Operationen im Code, die nur unter bestimmten Bedingungen durchführbar sind, werden hierbei nur versuchsweise (von engl. *try*: versuchen) durchlaufen. Sollten die Operationen tatsächlich nicht ausführbar sein, wird das Programm nicht abgebrochen. Stattdessen lässt sich innerhalb einer `except`-Klausel festlegen, wie in diesem Fall fortgefahrene werden soll. Eine `try ... except`-Anweisung hat die allgemeine Form:

```
1  try:
2      Anweisungsblock1
3  except [Ausnahmetyp]:
4      Anweisungsblock2
```

Die Angabe des Ausnahmetyps ist hier optional. Ein einfaches Codebeispiel mit einer `try ... except`-Anweisung ohne Angabe des Ausnahmetyps ist etwa:

#### Codebeispiel #3

```
1  try:
2      zahl = int(input("Bitte geben Sie eine ganze Zahl ein: "))
3      print("Danke, Sie haben die Zahl {} eingegeben!".format(zahl))
4  except:
5      print("Entschuldigung, Ihre Eingabe war nicht richtig.")
```

## 12.2 try ... except: Ausnahmen behandeln

```

Kapitel 12 (C:/Users/Sarah/PycharmProjects/Kapitel 12) - .../beispiel_3.py [Kapitel 12] - PyCharm
File Edit View Navigate Code Refactor Run Tools VCS Window Help
Kapitel 12 beispiel_3.py
Project 1 Project 2 Structure 2 Favorites
Kapitel 12 C:/Users/Sarah/PycharmProjects/Kapitel 12/venv/library root
  beispiel_1.py
  beispiel_2.py
  beispiel_3.py
  beispiel_4.py
  beispiel_5.py
  beispiel_6.py
  beispiel_7.py
  beispiel_8.py
  beispiel_9.py
  beispiel_10.py
  übung_1.py
  Übung_2.py
  ...
Run: beispiel_3
  C:/Users/Sarah/PycharmProjects/Kapitel 12/venv/Scripts/python.exe "C:/Users/Sarah/PycharmProjects/Kapitel 12/beispiel_3.py"
  Bitte geben Sie eine ganze Zahl ein: 5.5
  Entschuldigung, Ihre Eingabe war nicht richtig.

Process finished with exit code 0

```

Abb. 12.3 Ausnahmebehandlung bei falscher Nutzereingabe

Der zweite Anweisungsblock unter `except:` wird dabei nur in dem Fall ausgeführt, dass keine ganze Zahl eingegeben wurde und der `try`-Anweisungsblock somit zu einem Laufzeitfehler geführt hat. Konnte der `try`-Anweisungsblock fehlerfrei durchlaufen werden, wird die `except`-Klausel übersprungen.

Im obigen Beispiel müssen Sie in der `except`-Klausel nicht unbedingt spezifizieren, welchen Ausnahmetyp Sie behandeln möchten. Der Befehl kann auch ohne weitere Angaben erfolgen und fängt dann alle Ausnahmearten auf einmal ab. Derartige generische `except`-Klauseln können in einigen besonderen Fällen vorteilhaft sein – etwa, um einen unerwarteten Programmabsturz oder Datenverluste zu verhindern. Es ist dennoch ratsam, stets den spezifischen Ausnahmetyp anzugeben, da der Code dadurch leichter nachvollziehbar ist und genauer auf bestimmte Fälle eingehen kann. Zudem können nicht bedachte Ausnahmen durch eine spezifische Ausnahmebehandlung sichtbar gemacht werden. Zusätzlich zu den spezifischen `except`-Klauseln kann anschließend ein generischer `except`-Ausnahmeblock eingefügt werden, der alle unbekannten Ausnahmen handhabt.

Die spezifische Ausnahmebehandlung lässt sich auf zweierlei Weisen realisieren: Einerseits können Sie alle möglichen Ausnahmefälle als Tupel innerhalb einer einzigen `except`-Klausel festlegen, die dann jeweils dieselbe Ausnahmebehandlung zur Folge haben. Alternativ hierzu können Sie bei mehreren möglichen Ausnahmetypen jeweils eine `except`-Klausel pro Ausnahme innerhalb ein- und desselben `try`-Blocks vorsehen.

Wir wollen beide Varianten anhand des Codebeispiels aus dem vorigen Unterkapitel untersuchen, in dem wir die Division zweier eingegebener ganzer Zahlen vorgenommen hatten. Wir behandeln dabei sowohl die Ausnahme, dass keine ganze Zahl

## 12 Fehler und Ausnahmen behandeln

(`ValueError`) eingegeben wurde, als auch die Ausnahme, dass die zweite Eingabe eine Null (`ZeroDivisionError`) ist.

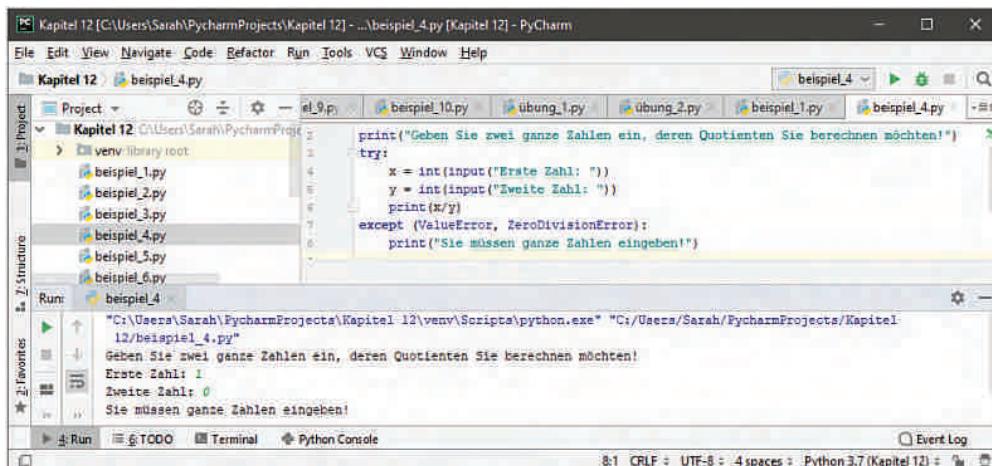
Die erste Variante lässt sich folgendermaßen ausdrücken:

### Codebeispiel #4

```

1 print("Geben Sie zwei ganze Zahlen ein, deren Quotienten Sie berechnen
2 möchten!")
3 try:
4     x = int(input("Erste Zahl: "))
5     y = int(input("Zweite Zahl: "))
6     print(x/y)
7 except (ValueError, ZeroDivisionError):
8     print("Sie müssen ganze Zahlen eingeben!")

```



**Abb. 12.4** Gemeinsame Ausnahmebehandlung bei zwei Ausnahmetypen

Wie Sie sehen, ist die Meldung des Programms im Falle einer unerwarteten Nutzereingabe hier nicht sehr spezifisch, da es sich auch bei der eingegebenen Zahl Null um eine ganze Zahl handelt, die jedoch für die Variable `y` zu einer ungültigen Operation führt.

Eine genauere und meist bevorzugte Ausnahmebehandlung wäre dadurch gegeben, dass mehrere Ausnahmefälle zu unterschiedlichen Rückmeldungen des Programms führen:

### Codebeispiel #5

```

1 print("Geben Sie zwei ganze Zahlen ein, deren Quotienten Sie berechnen
2 möchten!")
3 try:

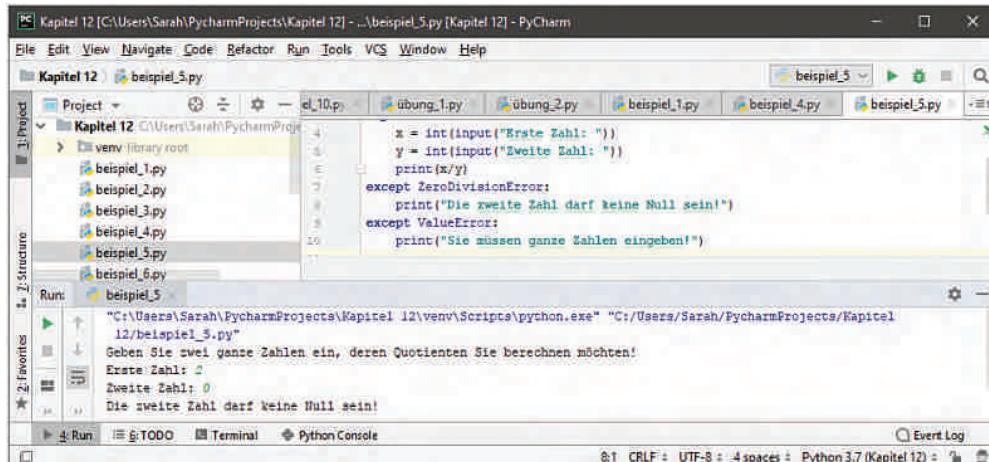
```

## 12.2 try ... except: Ausnahmen behandeln

```

4     x = int(input("Erste Zahl: "))
5     y = int(input("Zweite Zahl: "))
6     print(x/y)
7 except ZeroDivisionError:
8     print("Die zweite Zahl darf keine Null sein!")
9 except ValueError:
10    print("Sie müssen ganze Zahlen eingeben!")

```



**Abb. 12.5** Getrennte Ausnahmebehandlung bei zwei Ausnahmetypen

Es wird stets höchstens eine der möglichen Klauseln ausgeführt. Die Ausnahmehierarchie verläuft dabei von spezifischen hin zu weniger spezifischen Ausnahmen – ansonsten kann es passieren, dass Python beim konsekutiven Abarbeiten der `except`-Klauseln versehentlich eine allgemeinere Ausnahme feststellt, für die ebenfalls eine spezifischere Ausnahmebehandlung vorgesehen war. So könnte der Python-Interpreter etwa zunächst eine allgemeinere `ArithmeticError`-Ausnahme behandeln, obwohl es sich genauer gesagt um eine spezifischere `ZeroDivisionError`-Ausnahme handelt, wenn der Code der `ZeroDivisionError`-Ausnahme erst nach dem der `ArithmeticError`-Ausnahme steht.

Sind Ihnen alle möglichen Ausnahmen innerhalb einer Anwendung bekannt, sollten Sie diese also stets ihrer Hierarchie entsprechend angeben und so genau wie möglich behandeln. Die Ausnahmehierarchie finden Sie ebenfalls unter dem oben genannten Link. Beispielsweise können Sie der Hierarchien-Liste entnehmen, dass sich `ZeroDivisionError` und `OverflowError` beide von der Oberklasse `ArithmeticError` ableiten lassen. In unserem vorigen Divisions-Beispiel wäre es daher denkbar gewesen, die Ausnahme `ZeroDivisionError` durch `ArithmeticError` zu ersetzen.

Wir wollen nun eine weitere typische Variante der Ausnahmebehandlung betrachten, in der zusätzlich das Schlüsselwort `else` Anwendung findet. Hierfür werden wir das

## 12 Fehler und Ausnahmen behandeln

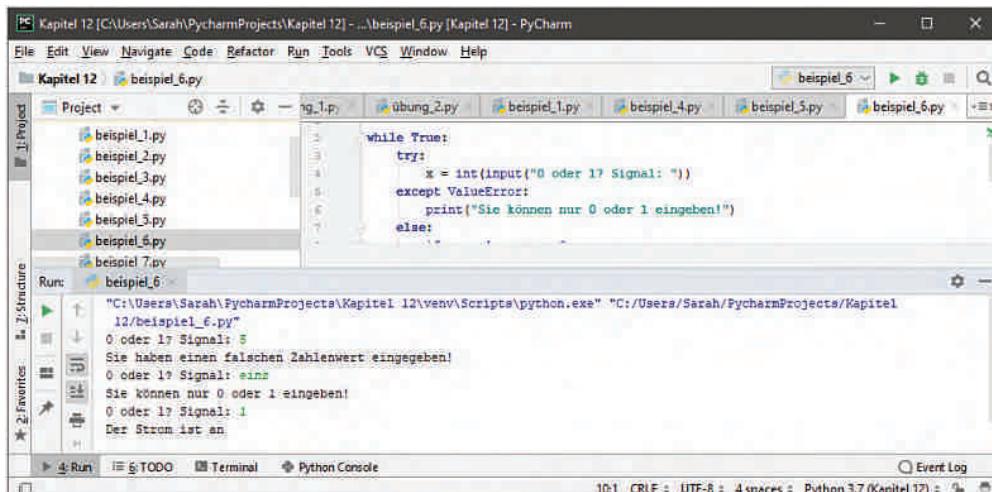
Strom-Beispiel aus Kapitel 7 heranziehen, das wir mit einer while-Schleife und einer break-Anweisung ausgeschmückt haben:

### Codebeispiel #6

```

1  while True:
2      try:
3          x = int(input("0 oder 1? Signal: "))
4      except ValueError:
5          print("Sie können nur 0 oder 1 eingeben!")
6      else:
7          if x == 1 or x == 0:
8              strom = ("an" if x == 1 else "aus")
9              print("Der Strom ist", strom)
10             break
11         else:
12             print("Sie haben einen falschen Zahlenwert eingegeben!")

```



**Abb. 12.6** Ausnahmen mit einem else-Block

Durch `try ... except` werden hier zu Anfang alle unerwarteten Eingaben abgefangen, die keine ganze Zahl darstellen wie durch `int(input())` gefordert. Gibt man hier etwa `eins` oder eine Fließkommazahl ein, erfolgt durch `except ValueError:` eine entsprechende Meldung, dass nur die Zahlen 0 oder 1 eingegeben werden können. Unser ursprüngliches Programm, das ohne diese Ausnahmehandlung durch `try ... except` lief, hätte in diesem Fall eine Fehlermeldung ausgegeben. `ValueError` steht hier für diejenige spezifische Ausnahme, dass die Nutzereingabe einen unerwarteten Wert enthält. Sie finden eine genauere Beschreibung dieses Ausnahmetyps unter dem oben angegebenen Link der Python-Dokumentation.

Der `else`-Block wird nur dann ausgeführt, wenn der `try`-Block keine Ausnahme ausgelöst hat – also genau dann, wenn tatsächlich eine ganze Zahl eingegeben wurde. In

## 12.2 try ... except: Ausnahmen behandeln

diesem Fall wird im `else`-Block überprüft, ob die Eingabe eine 0 oder eine 1 war – es erfolgen entsprechende `print`-Ausgaben. Allgemein lassen sich bei einer Konstruktion mit `else` innerhalb des `else`-Blocks beispielsweise Erfolgsmeldungen oder weitere Berechnungen mit im `try`-Block ermittelten Werten durchführen. Insbesondere bei längeren Programmen vermeidet man allzu lange Formulierungen im `try`-Block: Hier steht nur derjenige Code, der überprüft werden soll. Alles Weitere folgt im `else`-Block.

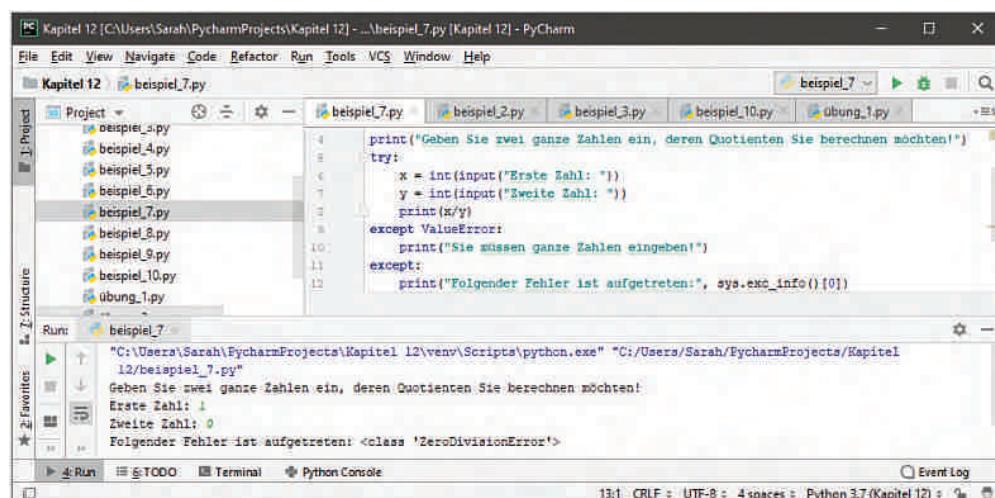
Abschließend möchten wir Ihnen in diesem Abschnitt vorstellen, wie sich unerwartete, unbekannte Ausnahmen handhaben lassen. So kann es mitunter passieren, dass ein Ausnahmefall auftritt, der in keine der angegebenen Kategorien der vorhandenen `except`-Blöcke fällt. Hierfür betrachten wir nochmals unser Divisions-Beispiel, nehmen jedoch an, dass die Ausnahme „Division durch Null“ im Code nicht bedacht wurde. Der letzte `except`-Block fängt hier zur Sicherheit alle Ausnahmen ab, die der vorige `except`-Block mit `ValueError` nicht behandelt:

### Codebeispiel #7

```

1 import sys
2
3 print("Geben Sie zwei ganze Zahlen ein, deren Quotienten Sie berechnen"
4 möchten!")
5 try:
6     x = int(input("Erste Zahl: "))
7     y = int(input("Zweite Zahl: "))
8     print(x/y)
9 except ValueError:
10    print("Sie müssen ganze Zahlen eingeben!")
11 except:
12    print("Folgender Fehler ist aufgetreten:", sys.exc_info()[0])

```



12

Abb. 12.7 Unerwartete Ausnahmen auffangen und anzeigen lassen

## 12 Fehler und Ausnahmen behandeln

Das `sys`-Modul enthält systemspezifische Parameter und Funktionen und muss zunächst importiert werden. Die Funktion `sys.exc_info` gibt ein Tupel mit drei Werten aus, die Auskunft über den konkreten Ausnahmetyp [0], den Ausnahmeparameter [1] sowie den Ausnahmeort [2] geben. Auf diese Weise lassen sich durch den letzten `except`-Block auch unbekannte Ausnahmetypen auffangen, ausweisen und dementsprechend behandeln, sodass es in keinem Fall zu einem Programmabbruch kommt.

### 12.3 finally: der Abschluss der Ausnahmebehandlung

Die `finally`-Anweisung definiert eine Anweisungsfolge, die in jedem Fall vor einem eventuellen Programmabbruch ausgeführt werden soll. Somit können etwa wichtige Dateien abgespeichert, Benutzer abgemeldet oder unnötige Daten gelöscht werden.

Diese Anweisung hat die folgende allgemeine Form:

```

1  try:
2      Anweisungsblock1
3  except [Ausnahmetyp]:
4      Anweisungsblock2
5  finally:
6      Anweisungsblock3

```

Wie bei `try ... except` wird auch hier zunächst der Anweisungsblock in der `try`-Klausel ausgeführt. Bei Bedarf folgt ein Anweisungsblock in der `except`-Klausel. Im Falle einer Ausnahme wird der Ausnahmetyp gespeichert und der Anwendungsblock der `finally`-Klausel ausgeführt. Anschließend wird das Programm abgebrochen und die Ausnahme angegeben. Doch auch in dem Fall, dass keine – oder eine unbekannte – Ausnahme auftritt, wird abschließend `finally` ausgeführt. Dieser Block wird somit in jedem Fall gelten. Sinnvoll wäre zum Beispiel, innerhalb eines `finally`-Blocks Dateien schließen zu lassen, sodass Daten bei einem Systemabsturz nicht verloren gehen. Nachfolgend sehen Sie ein Codebeispiel, in dem das Programm eine letzte Nachricht ausgibt, bevor es sich verabschiedet:

#### Codebeispiel #8

```

1  try:
2      zahl = int(input("Bitte geben Sie eine ganze Zahl ein: "))
3      print("Danke, Sie haben die Zahl {} eingegeben!".format(zahl))
4  except ValueError:
5      print("Entschuldigung, Ihre Eingabe war nicht richtig.")
6  finally:
7      print("Schönen Tag noch!")

```

## 12.4 Selbst definierte Ausnahmen festlegen

```

try:
    zahl = int(input("Bitte geben Sie eine ganze Zahl ein: "))
    print("Danke, Sie haben die Zahl {} eingegeben!".format(zahl))
except ValueError:
    print("Entschuldigung, Ihre Eingabe war nicht richtig.")
finally:
    print("Schönen Tag noch!")

```

The screenshot shows the PyCharm interface with the file `beispiel_8.py` open. The code above is displayed in the editor. In the run tab, the output of the script is shown:

```

"C:\Users\Sarah\PycharmProjects\Kapitel 12\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 12/beispiel_8.py"
Bitte geben Sie eine ganze Zahl ein: 2.E
Entschuldigung, Ihre Eingabe war nicht richtig.
Schönen Tag noch!

```

**Abb. 12.8** Die Ausführung der `finally`-Klausel

Wie Sie anhand des Screenshots erkennen können, wird der `print`-Befehl innerhalb des `finally`-Blocks trotz der falschen Eingabe ausgeführt. Ersetzen Sie im obigen Code einmal `ValueError` durch `SyntaxError` – somit existiert keine passende Ausnahmebehandlung für unerwartete Eingaben. Wenn Sie nun keine ganze Zahl eingeben, verabschiedet sich das Programm ebenfalls höflich durch die `finally`-Klausel, ehe es die Traceback-Ausgabe mit dem `ValueError` hinterlässt.

Die optionale `finally`-Klausel darf – genauso wie die optionale `else`-Klausel – nur einmal pro `try ... except`-Anweisung vorkommen. Sie steht immer ganz am Ende der entsprechenden Verzweigungen. Ist ein `else`-Zweig vorhanden, steht `finally` erst nach diesem.

## 12.4 Selbst definierte Ausnahmen festlegen

Mitunter ist es eine gute Idee, selbst Ausnahmen zu definieren, wenn keine der in Python standardmäßig integrierten Ausnahmen zur gewünschten Situation passt. Das könnte etwa auch dann der Fall sein, wenn man eine Ausnahme mit einem treffenderen Namen beschreiben möchte. Oder denken Sie an unser Strom-Beispiel zurück: Hier wäre es sehr praktisch, eine Ausnahme zu erstellen, die für alle Eingaben außer 0 und 1 gilt.

12

Um für dieses Beispiel eine eigene Ausnahme zu definieren, müssen wir eine neue Ausnahmeklasse erzeugen, die wir von einer bereits integrierten Klasse ableiten. Als Oberklasse können Sie hier einfach die allgemeine Klasse `Exception` wählen, für die wir weitere Einschränkungen vornehmen, sodass lediglich die Eingabewerte 0 und 1 keine Ausnahme auslösen. In Bezug auf die Bezeichnung von Ausnahmen ist es üb-

## 12 Fehler und Ausnahmen behandeln

lich, englische Wörter zu verwenden, die jeweils mit `Error` enden. Dadurch ist ein mit den bereits integrierten Ausnahmen einheitliches Benennungsschema gewährleistet. Wir wählen für unsere benutzerdefinierte Ausnahme – unsere selbst definierte Unterklasse – den Namen `ZeroOneError`.

Die neue Unterklasse `ZeroOneError` erbt von der Oberklasse `Exception` nach demselben Schema, wie wir es in Kapitel 11.6. gelernt haben. Die Ausnahme soll lediglich den Wert aufnehmen, der ihr durch die Nutzereingabe übergeben wird. Hierfür überschreiben wir die `__init__`-Methode, indem wir sie in der Unterklasse neu definieren. Die Unterklasse für unsere neu definierte Ausnahme `ZeroOneError` sieht nun wie folgt aus:

```
1 class ZeroOneError(Exception):
2     def __init__(self, wert):
3         self.Wert = wert
```

Selbst definierte Ausnahmen werden nicht eigenständig ausgelöst, sondern müssen innerhalb des Programms explizit geworfen werden. Hierfür dient das Schlüsselwort `raise`, auf das im Anschluss der Name der Ausnahme und in Klammern der Wert folgt, der an die Ausnahme übergeben werden soll. Sie können `raise` immer dann verwenden, wenn Sie eine bestimmte Ausnahme im Programm explizit behandeln möchten. Es muss dabei nicht immer eine Klammer mit einem Wert angegeben werden. Ebenso wenig muss ein konkreter Ausnahmefall vorliegen. Bevor wir uns unserem Strom-Beispiel widmen, soll uns ein stark vereinfachtes Beispiel zur kurzen Illustration von `raise` dienen:

```
1 try:
2     raise SyntaxError
3 except SyntaxError:
4     print("Die Syntax stimmt nicht!")
```

Hier wird der Syntax-Error gleich durch den `except`-Block abgefangen. Sie können jedoch auch Ausnahmen auslösen, ohne eine Ausnahmebehandlung vorzunehmen und dabei gleichzeitig die Traceback-Meldung anpassen. Geben Sie zum Beispiel einmal `raise SyntaxError("Fehlerhafte Syntax!")` ein und betrachten Sie die letzte Zeile der Traceback-Meldung.

In unserem Strom-Beispiel entscheidet die Nutzereingabe darüber, ob der Strom aus oder an ist, oder ob eine Ausnahme ausgelöst wird. Die Überprüfung, zu welchem Fall die Eingabe führt, lässt sich mithilfe einer `if-elif-else`-Abfrage vornehmen. Danach kann man die Ausnahme wie gewohnt durch `except` abfangen.

```
1 try:
2     x = int(input("0 oder 1? Signal: "))
3     if x == 1:
4         print("Der Strom ist an.")
```

## 12.4 Selbst definierte Ausnahmen festlegen

```

5     elif x == 0:
6         print("Der Strom ist aus.")
7     else:
8         raise ZeroOneError(x)
9
10    except ZeroOneError as fehler:
11        print("Die Eingabe {} ist keine 0 oder 1".format(fehler.Wert))

```

Beachten Sie dabei, dass die Variable `x` aufgrund von `int(input())` von vornherein nur ganze Zahlen als Werte annehmen kann. Die Eingabe einer Zeichenkette oder einer Fließkommazahl würde demnach nicht von unserer `ZeroOneError`-Ausnahme abgefangen werden. Um auch derartige Ausnahmen zu behandeln, können wir eine weniger spezifische Ausnahme, `ValueError`, innerhalb einer weiteren `except`-Klausel in unseren Code aufnehmen. Selbst definierte Ausnahmen lassen sich also problemlos mit integrierten Standardausnahmen kombinieren. Im folgenden Code sind alle für das vollständige Programm notwendigen Zeilen zusammengefasst:

### Codebeispiel #9

```

1  class ZeroOneError(Exception):
2      def __init__(self, wert):
3          self.Wert = wert
4
5      try:
6          x = int(input("0 oder 1? Signal: "))
7          if x == 1:
8              print("Der Strom ist an.")
9          elif x == 0:
10              print("Der Strom ist aus.")
11          else:
12              raise ZeroOneError(x)
13
14     except ZeroOneError as fehler:
15         print("Ihre Eingabe '{}' ist keine 0 oder 1!".format(fehler.Wert))
16     except ValueError:
17         print("Sie müssen eine 0 oder 1 eingeben!")
18
19     print("Vielen Dank.")

```

## 12 Fehler und Ausnahmen behandeln

**Abb. 12.9** Eigene Ausnahmen definieren

Innerhalb des ersten `except`-Blocks geschieht dabei Folgendes: Es wird eine Kopie der Ausnahme erstellt, indem dem Namen der Ausnahme der Begriff `as` und ein frei wählbarer Instanzname (hier: `fehler`) zugewiesen wird. Der eingegebene Zahlenwert lässt sich anschließend – wie bei Klassen üblich – über diesen Instanznamen gefolgt von einem Punkt und dem Attributnamen aufrufen, wie in Zeile 15 des Screenshots zu sehen ist. Dieser Zugriff ist möglich, da es sich bei konkreten Ausnahmen um nichts anderes als Instanzen bestimmter Klassen handelt. Wert ist in unserem Fall das entsprechende Attribut der Instanz `fehler`.

Nachdem die Ausnahmeinformationen durch die `as`-Klausel in die `fehler`-Instanz geschrieben wurden, lässt sich bei Bedarf darauf zugreifen. So können Sie mithilfe von Funktionen aus dem `sys`-Modul weitere Informationen über die Ausnahme erhalten, wie wir dies in 12.2. getan hatten. Das `sys`-Modul verfügt darüber hinaus über weitere Funktionen zur Fehlerbehandlung.

Wir werden uns im letzten Unterkapitel mit der Verwendung des PyCharm-Debuggers beschäftigen. Wenn Sie mehr über die Behandlung von Ausnahmen lernen möchten, können Sie sich beispielsweise mit verschachtelten Ausnahmen befassen.

## 12.5 Den PyCharm-Debugger verwenden

Auch wenn Sie glauben, innerhalb Ihres Programms alle Ausnahmen bedacht und vorsorglich behandelt zu haben, kann es dennoch zu Fehlern ganz unterschiedlicher Art kommen. Zum Lokalisieren und Beheben von Programmierfehlern (auch *Bugs* genannt) kommt daher zusätzlich ein sogenannter *Debugger* zum Einsatz. Dies ist nichts anderes als ein besonderes Werkzeug, mithilfe dessen Sie Ihre Anwendung

## 12.5 Den PyCharm-Debugger verwenden

schrittweise anhalten und den aktuellen Programmstatus auf Fehler untersuchen können, wodurch sich der Aufwand bei der Fehlerbehebung erheblich reduziert.

Ein Debugger realisiert üblicherweise Standard-Features wie das Setzen von Haltepunkten (engl. *breakpoints*), bedingtes Anhalten oder das Arbeiten mit Logdateien. Zudem ermöglicht er das Anzeigen aktueller Variablenwerte sowie deren Abändern während der Laufzeit. Während Debugger früher oftmals als unabhängige Software in Kombination mit einer IDE verwendet wurden, sind nunmehr die meisten IDEs – darunter auch PyCharm – standardmäßig mit einem Debugger ausgestattet.

Wir wollen sogleich anhand eines konkreten Codebeispiels in das Debuggen mit PyCharm einsteigen. Erinnern Sie sich an den Satz des Pythagoras zum Bestimmen der längsten Länge in einem rechtwinkligen Dreieck? Bei dieser Länge handelt sich um die sogenannte Hypotenuse: derjenigen Seite des Dreiecks, die dem rechten Winkel gegenüberliegt. Die Formel zur Berechnung der Hypotenuse  $c$  (bei bekannten Längen  $a$  und  $b$ ) lautet nach Pythagoras: . Wir werden nun einen Code betrachten, in dem zuerst die Hypotenuse  $c$  und anschließend der Gesamtumfang des Dreiecks berechnet wird. Um die Quadratwurzel-Funktion `sqrt()` verwenden zu können, müssen wir anfangs zudem das Modul `math` importieren. Im Code befindet sich ein Berechnungsfehler, den wir durch den Debugger-Vorgang finden möchten.

### Codebeispiel #10

```

1 import math
2
3 a = int(input("Länge Seite a [cm]: "))           # Haltepunkt
4 b = int(input("Länge Seite b [cm]: "))
5 c = math.sqrt(a**2 + b**2)                         # Haltepunkt
6 print("Länge Seite c: {:.2f} cm".format(c))
7
8 def umfang(x, y, z):                                # Haltepunkt
9     summe = x + y + z
10    return summe
11 print("Der Umfang ist: {:.2f} cm".format(umfang(a, b, c))) # Haltepunkt

```

Der Ausdruck `{:.2f}` in den Zeilen 6 und 11 bedeutet, dass die Fließkommazahlen auf zwei Nachkommastellen gerundet angezeigt werden.

12

Wenn Sie für die Variablen `a` oder `b` einen String eingeben, liefert die Ausgabe die Fehlermeldung `ValueError` und gibt die Zeile 3 (bzw. Zeile 4 für `b`) als fehlerhafte Zeile an. Ausnahmen dieser Art – etwa auch das Teilen durch Null – lassen oftmals bereits anhand der Fehleranzeige erkennen, bis zu welcher Stelle das Programm problemfrei ausgeführt werden konnte. Diese Ausnahmen müssten Sie selbstverständlich im Vorfeld durch eine entsprechende Ausnahmebehandlung mit `try...except` behandeln (Übungen 12.6.).

## 12 Fehler und Ausnahmen behandeln

Wir wollen nun untersuchen, wie sich Berechnungsvorgänge mithilfe des Debuggers überprüfen lassen. Dabei ist es einerseits möglich, dass das Programm nur bis zu einer bestimmten Stelle ausführbar ist und danach eine Ausnahme ausgibt. Andererseits könnte es sein, dass das Programm zwar problemlos ausführbar ist, trotz Ausbleiben einer konkreten Fehlermeldung jedoch ein implausibles Ergebnis liefert. In beiden Fällen sollten Sie als Erstes einen Blick auf den Code werfen und versuchen auszumachen, in welchem Bereich oder in welcher Zeile der Fehler aufgetreten sein könnte. Insbesondere bei kürzeren Anwendungen lässt sich der Fehler oftmals bereits auf diese Weise ausfindig machen.

Konnte man keine eindeutig fehlerhaften Stellen identifizieren, führt man typischerweise eine Debugging-Sitzung durch, indem man zunächst in mehreren Zeilen Haltepunkte setzt. Das Programm wird dadurch schrittweise jeweils nur bis zum nächsten Haltepunkt ausgeführt. Die Zeile mit dem Haltepunkt wird dabei nicht durchlaufen. Während der Programmunterbrechung lässt sich sodann der aktuelle Programmstatus detaillierter untersuchen. Die Anwendung bleibt solange angehalten, bis sie den Befehl bekommt, fortzufahren.

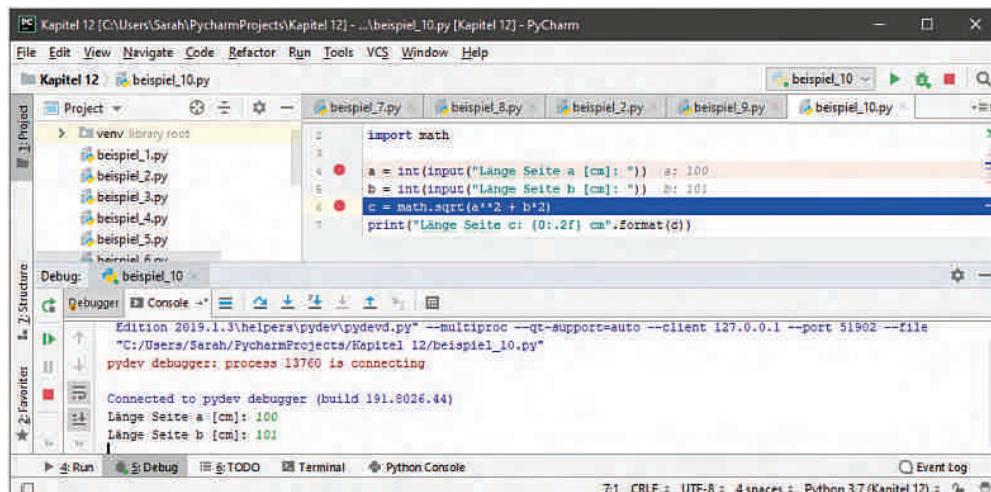
Die einfachste Möglichkeit, im PyCharm-Debugger einen Haltepunkt zu setzen, ist, direkt in den Bereich links neben der gewünschten Codezeile zu klicken. An dieser Stelle erscheint dann ein roter Kreis (●), der den Haltepunkt markiert. Alternativ hierzu können Sie entweder in die entsprechende Zeile klicken und im Hauptmenü unter **Run → Toggle Line Breakpoint** auswählen, oder auch das Tastatursymbol **Strg + F8** (bzw. **cmd + F8** für Mac OS X) verwenden. Um Haltepunkte zu entfernen, wiederholen Sie den Vorgang einfach. Übrigens müssen Sie nicht das gesamte Programm anhalten, um Haltepunkte zu setzen – dies kann auch während der Ausführung vorgenommen werden, was insbesondere bei größeren Programmen von Vorteil ist. Setzen Sie nun die Haltepunkte wie oben in den Kommentaren angegeben in Zeilen 3, 5, 8 und 11.

Nachdem die Haltepunkte gesetzt sind, lässt sich das Programm im Debug-Modus starten. Am einfachsten veranlassen Sie dies, indem Sie rechts oberhalb des geöffneten Programms auf das grüne Käfersymbol (🐞) klicken. Dabei muss sichergestellt werden, dass im Drop-down-Fenster links daneben die gewünschte Datei ausgewählt ist. Alternativ hierzu können Sie links im Projekt-Navigator mit der rechten Maustaste auf die entsprechende Datei klicken und die Option **Debug [Dateiname]** auswählen. Die Datei können Sie auch oben im Hauptmenü unter **Run → Debug...** auswählen. Möglicherweise müssen Sie das Programm bereits einmal gestartet haben, damit es hier als Auswahloption erscheint.

Anschließend wird das Programm bis zur Zeile mit dem ersten Haltepunkt ausgeführt, die blau hinterlegt dargestellt wird. Wie Sie sehen, erscheint im **Console**-Fenster unter dem Code keinerlei Output, da der Debugger die Zeile 3 mit dem ersten Haltepunkt noch nicht behandelt hat. Um das Programm bis zum nächsten Haltepunkt

## 12.5 Den PyCharm-Debugger verwenden

auszuführen, drücken Sie **F9** oder den grünen Pfeil ( ) links neben dem Debugger-Fenster.



**Abb. 12.10** Den Debugger mit Haltepunkten starten

Nun können Sie für `a` und `b` etwa die Werte 100 und 101 eingeben. Vielleicht fällt Ihnen hierbei auf, dass die beiden aktuellen Variablenwerte auf Ihre Eingabe hin ebenfalls in der jeweiligen Zeile im Editorfenster – in grauer Kursivschrift – angezeigt werden. So lässt sich beim Debugging im ersten Schritt immer überprüfen, ob die Werte aller Variablen stimmen. Wenn Sie mit dem Cursor in Zeile 3 über `a` fahren, wird zudem dessen aktueller Wert sowie der Datentyp eingeblendet. Auch unten im **Variables**-Fenster der **Debugger**-Ansicht sind diese Informationen nun sichtbar. Diese Möglichkeiten, Variablenwerte und Datentypen einzusehen, ist insbesondere bei längeren Programmen sehr nützlich.

Sie können Variablenwerte während des Debuggens sehr einfach aus dem **Console**-Fenster heraus ändern und somit versuchsweise unterschiedliche Programmverläufe und Ergebnisse testen. Bei angehaltenem Programm klicken Sie dazu auf das kleine Prompt-Ikon ( ) direkt am linken Rand des Debugger-Fensters. Es öffnet sich ein interaktives Konsolenfenster, in dem Sie die gewünschten Variablenwerte eingeben können, zum Beispiel `a = 1` und `b = 1`. Wenn Sie das Programm nun fortführen ( ), werden diese neuen Werte verwendet: Die berechnete Länge der Seite `c` wird im Konsolenfenster, in Zeile 5 sowie im **Debugger**-Fenster unter **Variables** als 1,73 angezeigt.

Aktuelle Berechnungswerte können Sie sich übrigens ebenfalls in einem externen Fenster anzeigen lassen. Dies lässt sich mit dem Befehl **Evaluate Expression** ( ) oder **Alt + F8**) bewerkstelligen, mit dem sich ganze Ausdrücke innerhalb des Codes auswerten lassen. Lassen Sie hier einmal separat den Ausdruck `math.sqrt(a**2 + b**2)` auswerten.

## 12 Fehler und Ausnahmen behandeln

Mit dem Befehl **Step Over** (Pfeil über dem Debugger-Fenster oder **F8**) wird die jeweilige Zeile ausgeführt und das Programm wiederum in der nächsten Zeile angehalten. Funktionen werden dabei jedoch übersprungen. Klicken Sie nun einmal auf diesen Button: Der Debugger springt über die `umfang`-Funktion zur letzten Zeile. Mit **F9** (bzw. ) wird auch diese ausgeführt und der Umfang wird angezeigt.

Vielleicht haben Sie anhand des errechneten Ergebnisses bemerkt, dass in unserem Code ein Formelfehler verborgen ist, obwohl der Debugger diesen nicht finden konnte: Die Quadratwurzel von 2 beträgt gerundet etwa 1,41 und nicht 1,73. In der Formel wurde ein \*-Operator vergessen, sodass versehentlich die Multiplikation mit 2 anstatt der zweiten Potenz berechnet wurde. Dieser Fehler hat im Folgecode ebenfalls Auswirkungen auf die Berechnung des Umfangs. Wenn Sie einen weiteren \*-Operator einfügen, stimmt die Rechnung.

Mit **Step Over** werden übrigens nicht nur Funktionen, sondern alle Einrückungen übersprungen. Mit **Step into** ( oder **F7**) lassen sich auch eingerückte Zeilen, zum Beispiel Funktionen, untersuchen. Sie können die beiden Optionen einmal selbst für die `umfang`-Funktion überprüfen.

Beim Debuggen von Codezeilen müssen Sie nicht zwingend Haltepunkte setzen. Sie können auch direkt auf die entsprechende Zeile klicken, bis zu der das Programm laufen soll. Mit **Run to Cursor** ( oder **Alt + F9**) können Sie den Code danach direkt bis zu dieser Stelle laufen lassen. Wenn Sie ein fehlerhaftes Programm ganz ohne Haltepunkte im Debug-Modus starten, wird es zudem meist automatisch in der fehlerhaften Zeile stehen bleiben.

Der Debugging-Prozess mag Ihnen im Falle dieses einfachen Beispiels vielleicht überflüssig erscheinen. Sobald Sie es aber mit längeren, komplexeren Programmen zu tun haben, werden Sie sehr schnell merken, dass der Debugger ein sehr hilfreiches Werkzeug ist. Wir empfehlen Ihnen an dieser Stelle, das Debugging-Tool mit Ihren eigenen Programmen zu verwenden und dabei weitere Funktionen selbstständig auszuprobieren. Eine ausführlichere Beschreibung des PyCharm-Debuggers finden Sie unter [www.jetbrains.com](http://www.jetbrains.com).

## 12.6 Übung: Ausnahmen im Programm behandeln

### 12.6 Übung: Ausnahmen im Programm behandeln

#### Übungsaufgaben

1. Korrigieren Sie den Code aus 12.5. für die Berechnung der Hypotenuse, sodass die Formel stimmt. Sie können die umfang-Funktion der Einfachheit halber weglassen. Erweitern Sie anschließend die möglichen Eingabezahlen auf Fließkommazahlen und sehen Sie eine Ausnahme für die Eingabe von Zeichenketten mit entsprechender Meldung vor. Der Code soll außerdem eine generische except-Klausel für andere unbekannte Ausnahmen beinhalten und die Art der Ausnahme anzeigen.
2. Erweitern Sie den Code aus Aufgabe 1 folgendermaßen: Definieren Sie eine eigene Ausnahme für Längenangaben mit negativen Zahlenwerten. Fügen Sie zuletzt eine passende finally-Klausel hinzu.

## 12 Fehler und Ausnahmen behandeln

### Lösungen

1.

```

1 import math
2 import sys
3
4 try:
5     a = float(input("Länge Seite a [cm]: "))
6     b = float(input("Länge Seite b [cm]: "))
7 except ValueError:
8     print("Sie müssen eine Zahl eingeben!")
9 except:
10    print("Folgende Ausnahme ist aufgetreten:", sys.exc_info()[0])
11 else:
12    c = math.sqrt(a**2 + b**2)
13    print("Länge Seite c: {:.2f} cm".format(c))

```

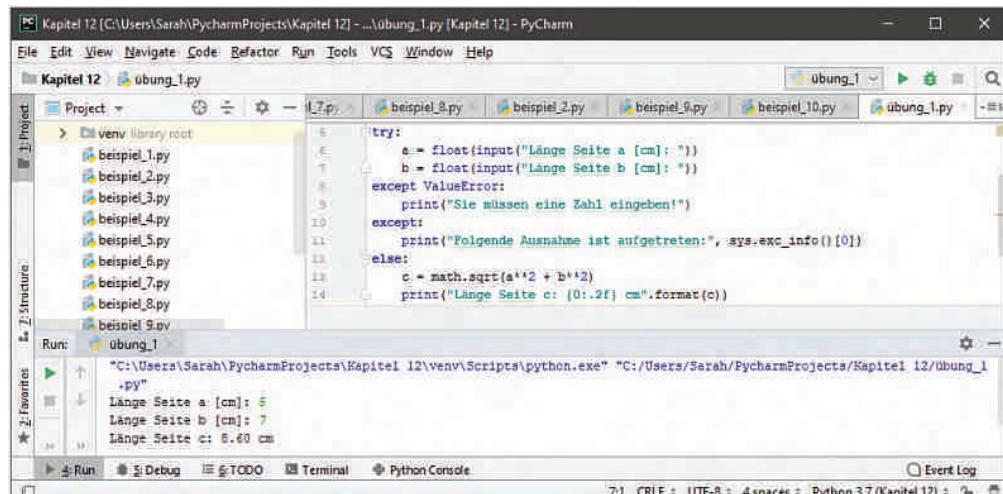


Abb. 12.11 Die Berechnung der Länge  $c$  mit Pythagoras

2.

```

1 import math
2 import sys
3
4 class NegativeNumberError(Exception):
5     def __init__(self, wert1, wert2):
6         self.Wert1 = wert1
7         self.Wert2 = wert2
8
9     try:
10        a = float(input("Länge Seite a [cm]: "))
11        b = float(input("Länge Seite b [cm]: "))
12        if a <= 0 or b <= 0:
13            raise NegativeNumberError(a, b)
14
15    except NegativeNumberError as fehler:

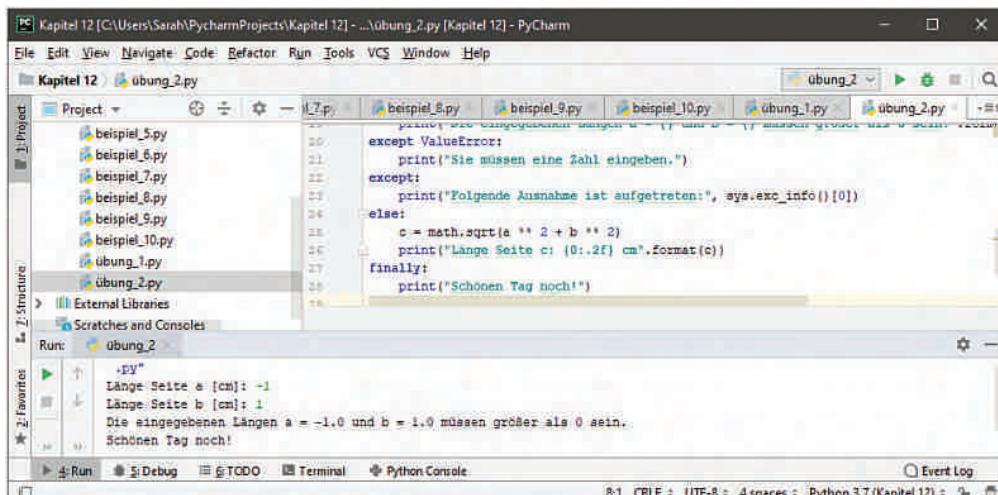
```

## 12.6 Übung: Ausnahmen im Programm behandeln

```

16     print("Die eingegebenen Längen a = {} und b = {} müssen größer als 0
17     sein.".format(fehler.Wert1, fehler.Wert2))
18 except ValueError:
19     print("Sie müssen eine Zahl eingeben.")
20 except:
21     print("Folgende Ausnahme ist aufgetreten:", sys.exc_info()[0])
22 else:
23     c = math.sqrt(a ** 2 + b ** 2)
24     print("Länge Seite c: {:.2f} cm".format(c))
25 finally:
26     print("Schönen Tag noch!")

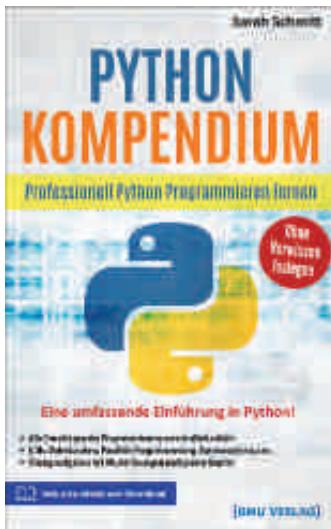
```



**Abb. 12.12** Eine selbst definierte Ausnahme verwenden

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 13

# Textdateien für die Datenspeicherung verwenden

Unsere bisherigen Programme verwendeten Daten, die wir direkt ins Programm eingaben und die nach dem Schließen der Anwendung wieder verloren gingen. Da eine solche Funktionsweise in reellen Anwendungen denkbar unpraktisch wäre, werden Sie in den nächsten beiden Kapiteln lernen, wie die permanente Datenspeicherung in Python funktioniert, sodass Sie umfangreichere, praxistaugliche Programme erstellen können.

Ein Python-Programm kann sowohl eingehende Daten lesen – etwa durch eine Tastatureingabe oder aus auf dem Computer vorhandenen Dateien – als auch ausgehende Daten in eine Datei schreiben oder auf dem Bildschirm ausgeben. Es gibt verschiedene Möglichkeiten, diese Operationen zu realisieren. In diesem Kapitel werden wir betrachten, wie sich Datenspeicherungsoperationen mit Textdateien ausführen lassen. Im nächsten Kapitel werden wir uns sodann dem Thema Datenbanken und weiteren Möglichkeiten der Datenspeicherung widmen.

Um Ihnen sogleich einen praktischen Mehrwert zu geben, werden wir die Möglichkeiten des Datenlesens und -speicherns von Textdateien in diesem Kapitel anhand von zwei Beispielen – einer deutschen Großstadt sowie eines Gedichts – untersuchen. Die Dateien werden hierbei als Objekte aus der Klasse `File` verwendet, über die die jeweiligen Ein- und Ausgabeoperationen laufen.

### 13.1 Daten aus einer Textdatei auslesen

Zunächst möchten wir betrachten, wie wir Daten, die in einer Textdatei gespeichert sind, für den Gebrauch innerhalb einer Anwendung aus dieser Datei auslesen können. Angenommen, Sie haben auf Ihrem Computer eine Datei namens *berlin.txt* gespeichert, die die Bevölkerungszahlen für fünf Berliner Bezirke für ein bestimmtes Jahr enthält. Die Angaben zu jedem Bezirk liegen in der *.txt*-Datei nach dem Schema

1	Berlin
2	Mitte
3	2018
4	383,457

## 13 Textdateien für die Datenspeicherung verwenden

vor, wobei die erste Zeile die Stadt, die zweite den Bezirk, die dritte das Jahr und die vierte die Bevölkerungsanzahl (in englischer bzw. Python-Schreibweise mit Komma) bezeichnet. Wir möchten diese Daten in unserem Programm nun dahingehend aufbereiten, dass wir darauf zugreifen und damit weitere Berechnungen anstellen können. Um die Datei *berlin.txt* mit PyCharm öffnen zu können, muss sie sich im aktuellen PyCharm-Projektordner befinden. Ansonsten gibt der Interpreter die Ausnahme `FileNotFoundException` aus. In den Übungsaufgaben werden Sie die Gelegenheit haben, eine Ausnahmebehandlung für einen solchen Fall zu formulieren.

Der Befehl zum Lesen unserer Datei *berlin.txt* lautet:

```
1 f = open("berlin.txt", "r")
```

Die `open`-Funktion enthält als ersten Parameter den Dateinamen als String. Sollte sich Ihre Datei nicht im aktuellen PyCharm-Projektordner befinden, müssen Sie hier stattdessen den gesamten Dateipfad angeben, zum Beispiel: `C:\Programme\Meine-Daten\berlin.txt`. Somit lassen sich auch Dateien öffnen, die an anderen Orten auf dem Rechner gespeichert sind. Der zweite Parameter, ebenfalls ein String, gibt an, dass die Datei im Lesemodus geöffnet werden soll (`r` steht für engl. *read*). An dieser Stelle sind als Einträge unter anderem auch `w`, `a` und `+` möglich, die wir in einem späteren Abschnitt dieses Kapitels kennenlernen werden.

Der `open`-Befehl erzeugt ein neues Dateiobjekt `f`, auf das wir unter der gewählten Bezeichnung für die weitere Datenbearbeitung zugreifen können. Der Name des Datenobjekts ist wie immer frei wählbar – meist sieht man hier jedoch die Bezeichnungen `fobj` oder einfach `f` für `file` (engl. für *Datei*). Um das Auslesen von Daten aus dieser Datei zu beenden und die Daten abzuspeichern, sollten Sie am Ende zudem immer die Methode `close()` verwenden.

Um unsere Bevölkerungsdaten nun zeilenweise auszulesen, führen wir ganz einfach eine Iteration mithilfe einer `for`-Schleife durch:

### Codebeispiel #1

```
1 f = open("berlin.txt", "r")
2 for zeile in f:
3     zeile = zeile.strip()
4     print(zeile)
5 f.close()
```

## 13.1 Daten aus einer Textdatei auslesen

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists 'Kapitel 13' and 'beispiel\_1.py'. Below the bar, the code editor displays a script named 'beispiel\_1.py' which reads data from a file named 'berlin.txt'. The 'Run' tool window at the bottom shows the output of the script, listing several entries from the file.

```

1 f = open("berlin.txt", "r")
2 print(f.read(300))

```

The 'berlin.txt' file contains the following data:

```

Berlin
Mitte
2018
383457
Berlin
Friedrichshain-Kreuzberg
2018
289120
Berlin
Tempelhof-Schöneberg
2018
351429
Berlin
Neukölln

```

**Abb. 13.1** Bevölkerungsdaten in einem Programm anzeigen lassen

Die Daten werden nun ähnlich wie in der .txt-Datei zeilenweise untereinander aufgelistet. Die `strip`-Funktion sorgt dafür, dass unnötige Leerzeichen, Tabulatorzeichen oder Zeilenumbrüche nicht dargestellt werden. Beachten Sie hierbei, dass `open` alle Zeichen aus der Datei standardmäßig als Strings ausliest. Wenn Sie nun beispielsweise mit einer Bevölkerungszahl weitere Berechnungen durchführen möchten, müssen Sie diese zunächst mit `int()` in eine ganze Zahl umwandeln.

Das Auslesen von Daten aus einer Textdatei ermöglicht sehr viele weitere Funktionalitätsoptionen und ist in erster Linie für das Bearbeiten von Texten geeignet. Beispielsweise können Sie sich den kompletten Inhalt einer Datei mithilfe der Methode `read()` anzeigen lassen. Standardmäßig gibt diese Methode den gesamten Inhalt der Datei aus. Wenn Sie der Klammer jedoch eine Zahl übergeben, können Sie sich lediglich eine gewünschte Zeichenanzahl anzeigen lassen. Beispielsweise können wir – sofern die Datei im entsprechenden Ordner existiert – die ersten 300 Zeichen des Gedichts „Das Ideal“ von Kurt Tucholsky folgendermaßen auslesen:

13

### Codebeispiel #2

```

1 f = open("Das Ideal - Kurt Tucholsky.txt", "r")
2 print(f.read(300))

```

## 13 Textdateien für die Datenspeicherung verwenden

The screenshot shows the PyCharm IDE interface. The project 'Kapitel 13' is open, and the file 'beispiel\_2.py' is selected in the Project tool window. The code in 'beispiel\_2.py' is:

```
f = open("Das Ideal - Kurt Tucholsky.txt", "r")
print(f.read(300))
```

In the Run tool window, the command run is:

```
"C:\Users\Sarah\PycharmProjects\Kapitel 13\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 13/beispiel_2.py"
```

The output pane shows the first 300 characters of the file 'Das Ideal - Kurt Tucholsky.txt'. The output is:

```
Das Ideal - Kurt Tucholsky
Ja, das möchte ich
Eine Villa im Grünen mit großer Terrasse,
vorn die Ostsee, hinten die Friedrichstraße;
mit schöner Aussicht, ländlich-mondän,
vom Badezimmer ist die Zugspitze zu sehn -
aber abends zum Kino hast du nicht weit.

Das Ganze schlicht, voller Bescheidenheit:
```

**Abb. 13.2** Die Ausgabe der ersten 300 Zeichen eines Gedichts

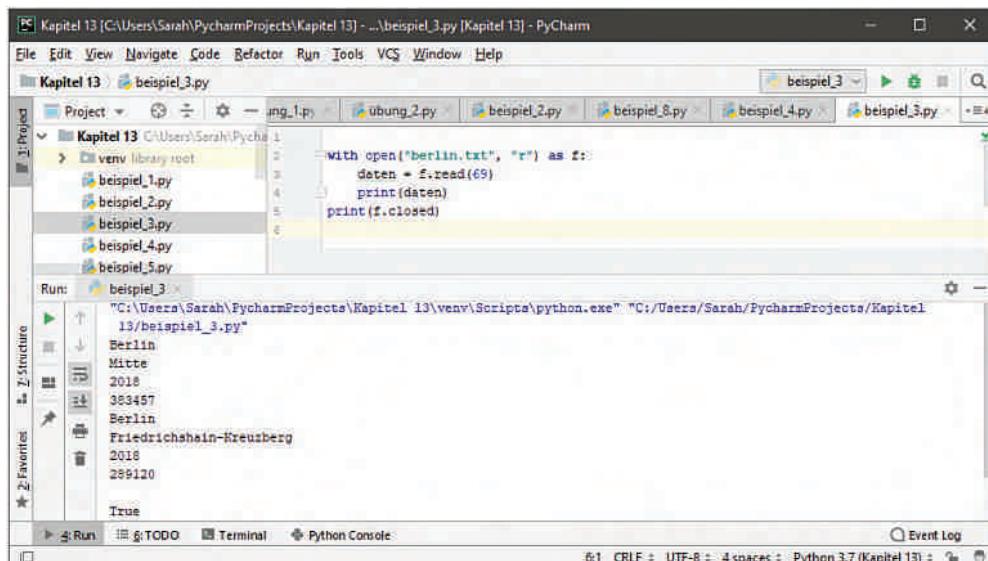
Mit der `readline`-Methode können Sie sich nach demselben Schema die erste Zeile der Datei ausgeben lassen. Fügen Sie im Code einen weiteren `readline`-Befehl hinzu, wird die zweite Zeile ausgegeben, bei einem weiteren die dritte usw. Diese Option ist sehr hilfreich, wenn Sie Informationen Zeile für Zeile für den weiteren Gebrauch in einer Anwendung auslesen möchten.

Um im Falle von Ausnahmen in jedem Fall zu gewährleisten, dass die Datei richtig geschlossen wird und keine Daten verloren gehen, kommt beim Arbeiten mit Dateien oftmals der Begriff `with` zum Einsatz. Dabei wird die Datei automatisch geschlossen, nachdem der `with`-Block durchlaufen wurde. Somit kann auf einen `try ... except ... finally`-Block zum Öffnen und Schließen der Datei bzw. auf die `close`-Methode am Ende des Codes verzichtet werden, was der Output True der letzten Zeile verdeutlicht:

### Codebeispiel #3

```
1 with open("berlin.txt", "r") as f:
2     daten = f.read(69)
3     print(daten)
4 print(f.closed)
```

## 13.1 Daten aus einer Textdatei auslesen

Abb. 13.3 Öffnen einer Datei mit `with`

Zum Abschluss dieses Abschnitts wollen wir die Daten unserer `berlin.txt`-Datei direkt in Objekte einer neuen Klasse `Bevoelkerung` umwandeln, um damit im Programm weitere Berechnungen anzustellen. Hierzu definieren wir zunächst unsere neue Klasse `Bevoelkerung` mit den gewünschten vier Attributen `stadt`, `bezirk`, `jahr` und `einwohner`, die in unserer Datei für vier verschiedene Bezirke – also vier verschiedene Klassenobjekte – vorliegen.

## Codebeispiel #4

```

1 class Bevoelkerung:
2     """Erstellt das Objekt Bevoelkerung für eine Stadt."""
3     def __init__(self, stadt, bezirk, jahr, einwohner):
4         """
5             Initialisiert ein neues Objekt Bevoelkerung
6
7             Argumente:
8             * stadt (string): die untersuchte Stadt
9             * bezirk (string): der jeweilige Stadtbezirk
10            * jahr (int): das Analysejahr
11            * einwohner (int): Einwohnerzahl des Bezirks
12        """
13
14        self.stadt = stadt
15        self.bezirk = bezirk
16        self.jahr = jahr
17        self.einwohner = einwohner
18
19    def geteinwohner(self):
20        """

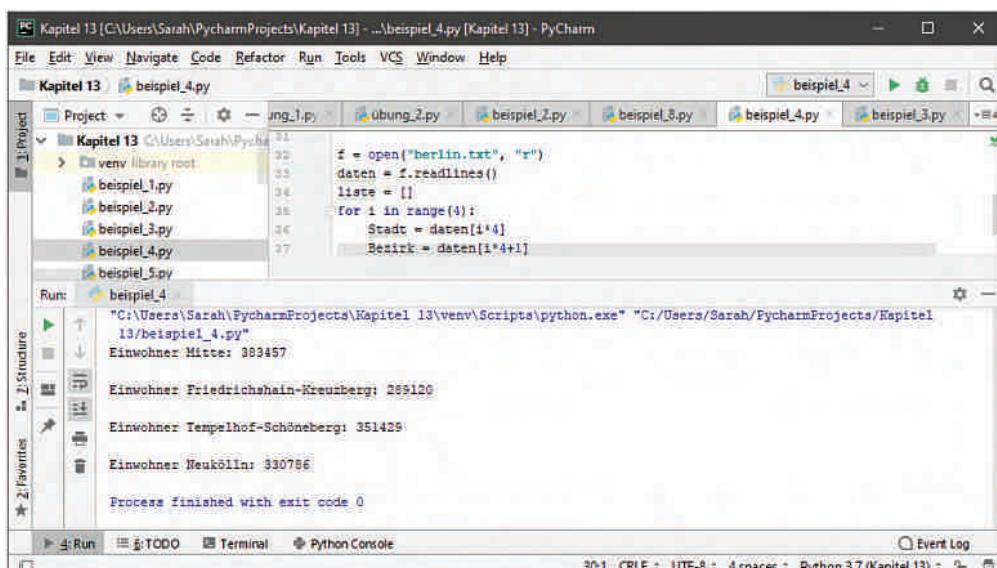
```

## 13 Textdateien für die Datenspeicherung verwenden

```

21         Gibt die Einwohnerzahl eines Bezirks zurück.
22         """
23         return self.einwohner
24
25     def seteinwohner(self, einwohner_neu):
26         """
27             Ändert die Einwohnerzahl des Bezirks.
28         """
29         self.einwohner = einwohner_neu
30
31     f = open("berlin.txt", "r")
32     daten = f.readlines()
33     liste = []
34     for i in range(4):
35         Stadt = daten[i*4]
36         Bezirk = daten[i*4+1]
37         Jahr = daten[i*4+2]
38         Einwohner = daten[i*4+3]
39         liste += [Bevoelkerung(Stadt, Bezirk, Jahr, Einwohner)]
40     for i in range(4):
41         print("Einwohner {0}: {1}".format(liste[i].bezirk.strip(), liste[i].
42             geteinwohner()))
43     f.close()

```



**Abb. 13.4** Daten aus einer Datei in einer Klasse anlegen

In der Klassendefinition ist ebenfalls eine Setter- und Gettermethode für die Einwohnerzahl vorgesehen. Während die zuvor erwähnte Methode `readline` lediglich zeilenweise Informationen aus einer Datei ausliest, legt die `readlines`-Methode jede Zeile der Datei als Element einer Liste an. Im Code wird diese Liste zunächst unter `daten` abgespeichert. Danach wird eine leere Liste, `liste`, erzeugt.

## 13.2 Daten in eine Textdatei schreiben

In der ersten `for`-Schleife wird `liste` sodann mit unseren vier Bevölkerungsobjekten mit den entsprechenden vier Attributen `stadt`, `bezirk`, `jahr` und `einwohner` gefüllt, die aus der ursprünglichen `daten`-Liste stammen. Um eine solche schematische Abfrage vornehmen zu können, muss Ihnen der Inhalt Ihrer Datei selbstverständlich genau bekannt sein und zudem in einheitlicher Form vorliegen. In der zweiten `for`-Schleife wird schließlich die Einwohnerzahl jedes Bezirks durch die Gettermethode `geteinwohner` angezeigt.

### 13.2 Daten in eine Textdatei schreiben

Wie im vorigen Kapitel bereits erwähnt wurde, lassen sich Dateien mithilfe des `open`-Befehls nicht nur zum Lesen, sondern auch umgekehrt zum Schreiben öffnen. Hierfür kommt als zweites Argument im `open`-Befehl der Buchstabe `w` zum Einsatz (engl. für `write`):

#### *Codebeispiel #5*

```
1 f = open("berlin2.txt", "w")
```

Existiert an dem besagten Ort bereits eine Datei mit dem angegebenen Namen, so wird ihr Inhalt gelöscht, wenn Sie sie nach diesem Schema zum Schreiben öffnen. Ansonsten wird automatisch eine neue Datei erstellt, die anschließend mit Inhalt ausgefüllt werden kann. Beispielsweise könnten wir unsere leere Textdatei `berlin2.txt` nun umgekehrt mit den Bevölkerungsdaten von Berlin-Mitte füllen wollen. Diesmal wählen wir als Datenstruktur ein Dictionary mit den entsprechenden Schlüssel-Wert-Paaren:

```
1 bev_berlin = {"Stadt": "Berlin", "Bezirk": "Mitte", "Jahr": 2018,
2 "Bevölkerung": 383457}
```

Nachdem das Dictionary angelegt wurde, können wir unsere Datei folgendermaßen mit dem Inhalt des Dictionaries füllen:

```
1 f = open("berlin2.txt", "w")
2 for eintrag in bev_berlin:
3     f.write("{} {}\n".format(eintrag, bev_berlin[eintrag]))
4 f.close()
```

## 13 Textdateien für die Datenspeicherung verwenden

The screenshot shows the PyCharm IDE interface. In the top navigation bar, the file 'beispiel\_5.py' is selected. The code editor displays the following Python script:

```

1 f = open("berlin2.txt", "w")
2 bev_berlin = {"Stadt": "Berlin", "Bezirk": "Mitte", "Jahr": 2018, "Bevölkerung": 383457}
3 f = open("berlin2.txt", "w")
4 for eintrag in bev_berlin:
5     f.write("{0} {1}\n".format(eintrag, bev_berlin[eintrag]))
6 f.close()

```

In the bottom right corner, a terminal window titled 'berlin2 - Notepad' shows the output of the script:

```

Stadt Berlin
Bezirk Mitte
Jahr 2018
Bevölkerung 383457

```

**Abb. 13.5** Die Informationen wurden in die Datei geschrieben

Wenn Sie nun in dem entsprechenden PyCharm-Ordner nach der Datei *berlin2.txt* suchen, werden Sie sehen, dass diese dort mit dem gewünschten Inhalt erstellt wurde. Die Verwendung der `format`-Funktion sollte Ihnen mittlerweile soweit bekannt sein, dass wir sie hier nicht weiter erklären werden. Mit `\n` wird im Ausgabestring ein Zeilensprung veranlasst.

Analog zu `f.read` für Leseprozesse können Sie für das Schreiben von Strings in eine Datei also die Methode `f.write` nutzen. Dabei können Sie nicht nur in *.txt*-Dateien, sondern auch in andere Dateiformate, zum Beispiel DOC-Dateien, schreiben. Im folgenden Beispiel wird zunächst die Word-Datei angelegt und die erste Gedichtstrophe in einem String abgespeichert, der sodann mit `write` in die Datei geschrieben wird. Die Ausgabe des dargestellten `print`-Befehls ist die Anzahl der Zeichen im angegebenen String:

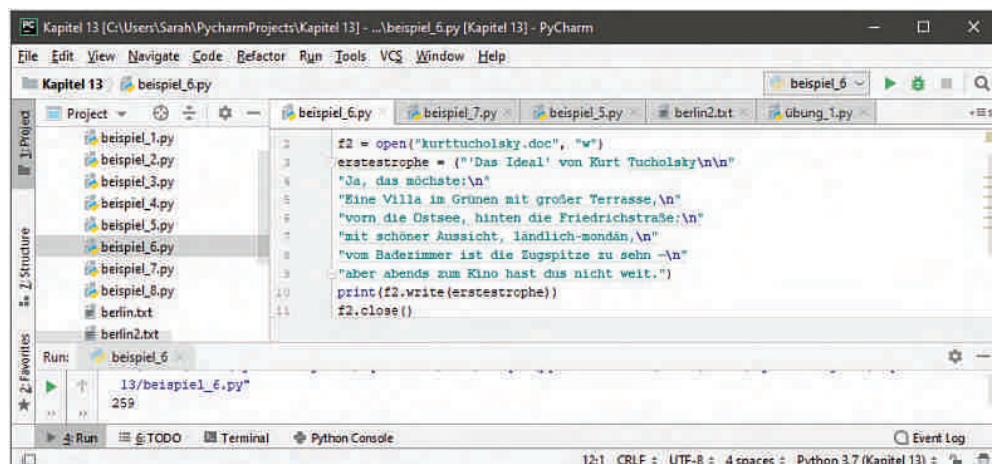
### Codebeispiel #6

```

1 f2 = open("kurttucholsky.doc", "w")
2 erstestrophe = ("'Das Ideal' von Kurt Tucholsky\n\n"
3 "Ja, das möchste:\n"
4 "Eine Villa im Grünen mit großer Terrasse,\n"
5 "vorn die Ostsee, hinten die Friedrichstraße;\n"
6 "mit schöner Aussicht, ländlich-mondän,\n"
7 "vom Badezimmer ist die Zugspitze zu sehn -\n"
8 "aber abends zum Kino hast dus nicht weit.")
9 print(f2.write(eriestrophe))
10 f2.close()

```

## 13.2 Daten in eine Textdatei schreiben



The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_1.py' through 'beispiel\_6.py', 'berlin.txt', and 'berlin2.txt'. Below it, the code editor displays 'beispiel\_6.py' containing Python code that reads a poem from a Word document and writes it to 'berlin2.txt'. The 'Run' tab at the bottom shows the command '13/beispiel\_6.py' and the result '259'. The status bar at the bottom right indicates the time as 12:11, encoding as CRLF, and Python version as 3.7 (Kapitel 13).

```

1 f2 = open("kurttucholsky.doc", "w")
2 erstestrofe = ("Das Ideal" von Kurt Tucholsky\n\n"
3 "Ja, das möchte:\n"
4 "Eine Villa im Grünen mit großer Terrasse,\n"
5 "vorn die Ostsee, hinten die Friedrichstraße:\n"
6 "mit schöner Aussicht, ländlich-mondän,\n"
7 "vom Baderimmer ist die Zugspitze zu sehn -\n"
8 "aber abends zum Kino hast du nicht weit.")
9 print(f2.write(ertestestrofe))
10 f2.close()

```

**Abb. 13.6** Einen String mit write in eine Datei schreiben

Wenn Sie die soeben angelegte Word-Datei öffnen, werden Sie sehen, dass die erste Strophe darin enthalten ist.

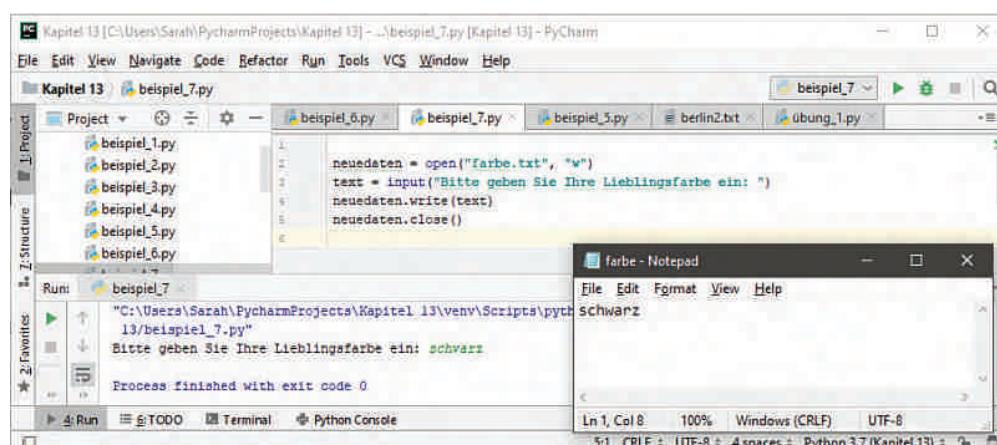
Eine interessante Möglichkeit, wie Sie eine Tastatureingabe direkt in einer Datei speichern können, ist im folgenden Codebeispiel illustriert:

### Codebeispiel #7

```

1 neuedaten = open("farbe.txt", "w")
2 text = input("Bitte geben Sie Ihre Lieblingsfarbe ein: ")
3 neuedaten.write(text)
4 neuedaten.close()

```



The screenshot shows the PyCharm IDE. The code editor contains 'beispiel\_7.py' with the provided script. The 'Run' tab shows the command '13/beispiel\_7.py' and the user input 'Bitte geben Sie Ihre Lieblingsfarbe ein: schwarz'. The terminal output shows the process finished with exit code 0. To the right, a Notepad window titled 'farbe - Notepad' displays the word 'schwarz'. The status bar at the bottom right shows the file is 51 bytes long, encoded in CRLF, and uses 4 spaces for Python 3.7 (Kapitel 13).

**Abb. 13.7** Dateien durch eine Tastatureingabe mit Informationen füllen

## 13 Textdateien für die Datenspeicherung verwenden

### 13.3 Weitere Lese- und Schreibeoptionen

In den vorigen beiden Teilkapiteln haben Sie gesehen, dass die `open`-Funktion jeweils zwei Argumente aufnimmt: den Dateinamen sowie eine Angabe darüber, welche Lese- oder Schreiboperation mit der Datei ausgeführt werden soll. Beide Angaben stehen im String-Format. Das zweite Argument ist dabei optional: Wird hier nichts angegeben, verwendet der Interpreter automatisch den Lesemodus ("r") als Voreinstellung. Außer den Werten "r" und "w" lassen sich hier auch die Buchstaben "x" und "a" sowie das Zusatzzeichen "+" verwenden. Der folgende tabellarische Überblick fasst die verschiedenen Optionen zusammen:

Dateimodus	Beschreibung
"r"	Standardmodus. Die Datei wird zum Lesen geöffnet. Kann sie nicht gefunden werden, erfolgt eine Fehlermeldung.
"w"	Schreibmodus. Existiert die Datei nicht, wird eine neue Datei angelegt. Existiert die Datei bereits, wird sie geleert und mit dem neuen Inhalt beschrieben.
"x"	Erstellt eine neue Datei. Ist die Datei bereits vorhanden, wird die Fehlermeldung <code>FileExistsError</code> ausgegeben.
"a"	Die Datei wird im append-Modus geöffnet; neuer Inhalt wird am Ende der Datei hinzugefügt. Ist keine Datei vorhanden, wird eine neue Datei angelegt.
"+"	Datei wird sowohl zum Lesen als auch zum Schreiben geöffnet.

**Tab. 13.1** Verschiedene Möglichkeiten für Operationen mit Dateien

Standardmäßig werden Dateien im Textmodus ("t") geöffnet, sodass dieser nicht spezifiziert werden muss. Möchten Sie jedoch stattdessen das Binärformat – etwa für Bilddateien – verwenden, so kommt hierfür der String "b" zum Einsatz. Wenn Sie den Binärmodus explizit auswählen möchten, so müssen Sie diesen String direkt hinter den Dateimodus schreiben, also beispielsweise "wb" oder "r+b".

Angenommen, wir möchten nun die Bevölkerungsdaten von Berlin, die bereits in der Datei `berlin.txt` vorliegen, um die Angaben eines weiteren Bezirks erweitern. Hierzu können wir die `append`-Option folgendermaßen einsetzen:

#### Codebeispiel #8

```

1 f = open("berlin.txt", "a+")
2 f.write("\nBerlin\nCharlottenburg-Wilmersdorf\n2018\n341,327")
3 print(f.read())
4 f.close()

```

### 13.3 Weitere Lese- und Schreibeoptionen



**Abb. 13.8** Einer Datei nachträglich Informationen hinzufügen

Der zusätzliche `+`-Operator sorgt dabei dafür, dass die Datei nach dem Hinzufügen der Informationen durch den bereits angegebenen `open`-Befehl direkt ausgelesen werden kann. Ohne diesen Operator würden Sie beim Ausleseversuch die Fehlermeldung erhalten, dass die Datei nicht lesbar ist.

Um vorhandene Dateien zu löschen, können Sie das `os`-Modul importieren und die darin enthaltene `remove`-Funktion verwenden. Hierfür müssen Sie jedoch sicherstellen, dass die genannte Datei tatsächlich existiert. Auf den `remove`-Befehl hin wird die Datei sodann aus dem Ordner entfernt:

```

1 import os
2
3 os.remove("kurttucholsky.txt")

```

In diesem Kapitel haben wir primär untersucht, wie sich Daten aus `.txt`-Dateien in einer Anwendung ausgeben bzw. wie sich Daten in eine `.txt`-Datei schreiben lassen. Möchten Sie etwa Daten aus einer Excel- oder einer `.csv`-Datei importieren, können Sie hierfür die `pandas`-Bibliothek verwenden. Damit lassen sich umfassendere statistische Analysen durchführen, die wir in Kapitel 18 über Datenanalyse kennen lernen werden. Eine andere Option der Datenspeicherung und -verarbeitung sind Datenbanken. Diese werden Inhalt des nächsten Kapitels sein.

## 13 Textdateien für die Datenspeicherung verwenden

### 13.4 Übung: Mit Dateien für die Datenspeicherung arbeiten

#### Übungsaufgaben

1. Wie in Abschnitt 13.1. erwähnt, sollten Sie für den Fall, dass auf Ihrem Rechner keine Datei mit dem im `open`-Befehl angegebenen Namen existiert, sicherheitsshalber eine Ausnahmebehandlung für einen `FileNotFoundException` vorsehen. Nehmen Sie dies nun durch eine `try ... except`-Klausel vor und testen Sie den Code für den Fall, dass keine Datei mit dem entsprechenden Namen gefunden werden kann. Legen Sie dafür innerhalb der `except`-Klausel die entsprechende Datei an. Überlegen Sie sich auch, wie Sie gewährleisten können, dass die Datei in jedem Fall wieder geschlossen wird. Sie können ebenfalls eine weitere, allgemeine `except`-Klausel für unbekannte Ausnahmen vorsehen und sich den Ausnahmetyp ausgeben lassen.
2. Fügen Sie der Datei `berlin.txt` weitere Bevölkerungsdaten eines Berliner Bezirks hinzu. Dieser Vorgang soll durch eine Tastatureingabe erfolgen, in der die Nutzer und Nutzerinnen aufgefordert werden, die entsprechenden Informationen zu den vier gespeicherten Punkten (Stadt, Bezirk, Jahr und Bevölkerung) selbst einzugeben. Achten Sie darauf, dass die Ausgabezeilen korrekt untereinanderstehen.

## 13.4 Übung: Mit Dateien für die Datenspeicherung arbeiten

## Lösungen

1.

```

1  try:
2      f = open("readme.txt", "r")
3  except FileNotFoundError:
4      f = open("readme.txt", "w")
5      print("Datei existiert nicht, wird angelegt.")
6  except Exception as fehler: # die Klasse Exception enthält alle Ausnahmen
7      print("Ein unbekannter Fehler ist aufgetreten: {}".format(fehler))
8  else:
9      daten = f.read()
10     print("Datei erfolgreich geöffnet.")
11 finally:
12     f.close()

```

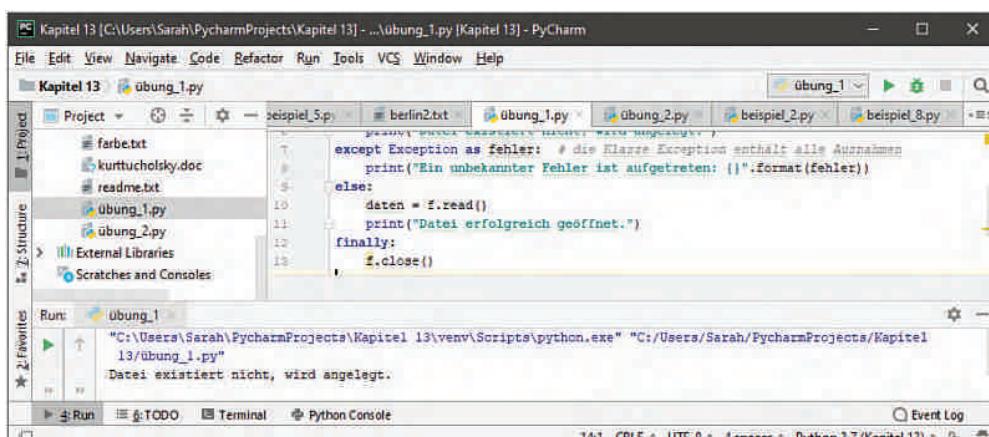


Abb. 13.9 Ausnahmebehandlung beim Öffnen einer Datei

2.

```

1  f = open("berlin.txt", "a+")
2  stadt = input("Geben Sie die Stadt ein: ")
3  bezirk = input("Geben Sie den Stadtbezirk ein: ")
4  jahr = input("Geben Sie das Analysejahr ein: ")
5  bevoelkerung = input("Geben Sie die Bevölkerungszahl ein: ")
6  f.write(stadt+"\n")
7  f.write(bezirk+"\n")
8  f.write(jahr+"\n")
9  f.write(bevoelkerung+"\n")
10 f.close()

```

## 13 Textdateien für die Datenspeicherung verwenden

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists 'Kapitel 13' and 'Übung\_2.py'. The code editor window displays the following Python script:

```
f = open("berlin.txt", "a+")
stadt = input("Geben Sie die Stadt ein: ")
bezirk = input("Geben Sie den Stadtbezirk ein: ")
jahr = input("Geben Sie das Analysejahr ein: ")
bevoelkerung = input("Geben Sie die Bevoelkerungszahl ein: ")
f.write("\n"+stadt+"\n")
f.write(bezirk+"\n")
f.write(jahr+"\n")
f.write(bevoelkerung+"\n")
f.close()
```

The 'Run' tab in the bottom left shows the output of running the script:

```
Geben Sie das Analysejahr ein: 2018
Geben Sie die Bevoelkerungszahl ein: 305697
Process finished with exit code 0
```

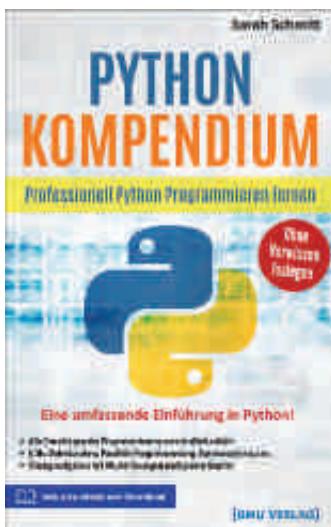
A separate Notepad window titled 'berlin - Notepad' shows the contents of the file 'berlin.txt' after execution:

Content
2018 341,327 Berlin Pankow 2018 305697

Abb. 13.10 Einer Datei nachträglich Informationen hinzufügen

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 14

# Mit Datenbanken arbeiten

Im vorigen Kapitel haben Sie bereits die vier wesentlichen Datenoperationen *Erzeugen, Lesen, Aktualisieren* und *Löschen* kennen gelernt, für die häufig auch die Abkürzung *CRUD* (engl. für *Create, Read, Update* und *Delete*) verwendet wird. Obwohl zwar jedes Programm, das Daten permanent abspeichert, diese vier Operationen realisieren können muss, sind sie vor allem in Bezug auf fortgeschrittenere Anforderungen an die Datenspeicherung wichtig.

Um die sichere Ausführung der *CRUD*-Operationen zu gewährleisten und Datenfehler zu verhindern, ist insbesondere bei größeren Datenmengen ein umfassenderes System zur Handhabung der Daten sinnvoll. Hier erleichtern Datenbanken die Arbeit mit Daten erheblich. Während Textdateien unstrukturierte Daten enthalten, liegen die Informationen in einer Datenbank in strukturierter Form vor. Das bedeutet, dass der Inhalt und die Anzahl der einzelnen Spalten sowie die Reihenfolge, in der die jeweiligen Informationen vorliegen müssen, genau vorgeschrieben sind.

Im Vergleich zur Datenspeicherung in Textdateien sind Datenbanken darüber hinaus in vielerlei Hinsicht vorteilhaft. So besteht beim Arbeiten mit Datenbanken die Option, diese durch ein Passwort vor unerwünschtem Zugriff zu schützen. Noch dazu können mehrere Anwenderinnen und Anwender simultan auf die Daten zugreifen, sodass ihnen jederzeit alle aktuellen Informationen aus den Datensätzen zur Verfügung stehen.

Im Gegensatz zu einfachen Textdateien weisen Datenbanken also eine besondere Struktur auf, über die wir Ihnen in diesem Kapitel einen ersten Überblick geben werden. Wenn Sie professionell mit Datenspeicherung arbeiten möchten, kommen Sie meist nicht umhin, sich hierfür mit Datenbanken zu befassen.

### 14.1 Was ist eine Datenbank?

Datenbanken dienen allgemein der Verwaltung von Daten in elektronischer Form. Dies umfasst die effiziente, permanente Speicherung der Daten sowie deren Bereitstellung bei Bedarf seitens der Nutzer oder im Rahmen einer Anwendung. Zu einem Datenbanksystem gehören einerseits das Verwaltungssystem – das sogenannte *Datenbankmanagementsystem (DBMS)* – sowie andererseits die Datenbank an sich, die die Gesamtheit aller Daten enthält. Zum Verwalten und Abfragen der Daten wird eine Datenbanksprache eingesetzt.

## 14.1 Was ist eine Datenbank?

Es gibt verschiedene Möglichkeiten, Datenbanken zu strukturieren. Am häufigsten kommen dabei sogenannte relationale Datenbanken zum Einsatz, bei denen die Relation zwischen Tupeln und Attributen in Tabellenform – also durch Zeilen und Spalten – ausgedrückt wird. Die Attribute sind hierbei die verschiedenen Spaltenköpfe, während die einzelnen Spalteneinträge die spezifischen Attributwerte enthalten. Jede Zeile der Tabelle ist ein Tupel und repräsentiert einen eigenen Datensatz. So könnte ein Buch in unserem Bibliotheksbeispiel aus Kapitel 6 etwa durch die jeweiligen Attributwerte der Spalten *Titel*, *Autor\_in*, *Erscheinungsjahr* und *Verlag* definiert werden. Dies mag zunächst an die Struktur eines Dictionaries oder einer Klasse mit unterschiedlichen Objekten erinnern. In relationalen Datenbanken ist jeder Datensatz – in unserem Fall: jedes Buch – jedoch zusätzlich mit mindestens einem unveränderlichen Schlüssel versehen, durch den dieser sich eindeutig identifizieren lässt. Ein kompletter Buch-Datensatz hätte somit beispielweise die allgemeine Spaltenform (*ISBN-Nummer*, *Titel*, *Autor\_in*, *Erscheinungsjahr*, *Verlag*) mit den jeweiligen konkreten Attributwerten, wobei die ISBN-Nummer jedes Buchs als dessen eindeutiger Schlüssel gewählt werden kann.

Relationale Datenbanken ermöglichen außerdem Verknüpfungen zwischen unterschiedlichen Tabellen. So könnten wir zusätzlich zu unserer oben definierten „Buch“-Tabelle die beiden weiteren Tabellen „Nutzer\_in“ und „Ausleihen“ anlegen, die einerseits die Daten aller Bibliotheksnutzerinnen und -nutzer sowie andererseits alle Informationen zu deren Buchausleihen nach ISBN-Nummer enthalten. In einem darauf aufbauenden Bibliothekssystem lassen sich sodann alle drei Tabellen durch die ISBN-Nummer und die Nutzer-ID praktisch verknüpfen, sodass die Daten weiter aufbereitet und präsentiert werden können.

Die am weitesten verbreitete Datenbanksprache ist *SQL*. Die Abkürzung steht für *Structured Query Language* (zu Deutsch: „strukturierte Abfragesprache“) und bezeichnet eine Datenbanksprache, mithilfe derer sich Datenstrukturen in relationalen Datenbanken definieren und darauf basierende Datenbestände abfragen und bearbeiten lassen. SQL wurde Anfang der 70er Jahre, als erstmals größere Datenspeicher zum Einsatz kamen, von IBM entwickelt. Die Einfachheit, Stabilität und Effizienz dieser Sprache ließen sie schnell zu einem großen Erfolg werden, und so gilt SQL auch weiterhin als die Standardsprache für Anwendungen, in denen Datenmengen behandelt werden müssen. Obwohl es mittlerweile – vor allem seit Aufkommen von *Big Data* und dem *Web 2.0* – andere Optionen zur Bearbeitung von Datenbanken gibt, die wir in diesem Kapitel ebenfalls kurz vorstellen werden, stellt SQL für sehr viele Anwendungen immer noch die beste Wahl dar.

Für SQL gibt es je nach technischen Anforderungen, Betriebssystem und verwendeter Programmiersprache unterschiedliche DBMS-Versionen, die wie Dialekte auf der Standardsprache SQL basieren. Nicht alle dieser Varianten sind quellenoffen und kostenlos.

## 14 Mit Datenbanken arbeiten

*SQLite* ist ein einfaches relationales Datenbankmanagementsystem (kurz: *RDBMS* für *Relational Database Management System*), das vorzugsweise in kleineren Anwendungen zum Einsatz kommt. *SQLite* ist standardmäßig in Python 3 integriert, sodass keine externen Bibliotheken erforderlich sind. Das Modul lässt sich einfach durch den Befehl `import sqlite3` verwenden. *SQLite* speichert die gesamte Datenbank in einer einzigen Datei auf der Festplatte ab und verwendet keinen separaten Datenbankserver. Aufgrund seiner reduzierten Funktionalität kann auf eine *SQLite*-Datenbank jedoch jeweils nur von einem Computer aus zugegriffen werden, weswegen es für Webanwendungen ungeeignet ist.

*MySQL* ist ein quelloffenes RDBMS, das sich in Kombination mit verschiedenen Programmiersprachen, darunter auch Python, C++, PHP oder C#, einsetzen lässt. Die Internetdienste Facebook, Twitter und YouTube verwenden beispielsweise *MySQL*. Diese Datenbanksprache ist anfangs zwar einfach zu erlernen, bietet im Vergleich zu vielen anderen Systemen jedoch weniger Features.

Eine weitere Open-Source-Implementierung von SQL ist *PostgreSQL*. Dieses relationale Datenbankmanagementsystem eignet sich ebenfalls sehr gut für Webanwendungen und wird unter anderem von Uber, Netflix und Spotify eingesetzt. Aufgrund seiner fortgeschrittenen Funktionalität erfreut sich *PostgreSQL* allgemein großer Beliebtheit und stellt meist die erste Wahl in Kombination mit umfangreicheren Python-Anwendungen dar.

Um Ihnen einen ersten Einblick in die Funktionsweise von relationalen Datenbanken zu geben, werden wir uns im Folgenden der Einfachheit halber ausschließlich mit *SQLite* beschäftigen.

### 14.2 SQLite-Datenbanken erstellen und bearbeiten

Wir werden nun die grundlegenden Befehle zum Erstellen und Bearbeiten einer *SQLite*-Datenbank mit Python kennen lernen. Um *SQLite* verwenden zu können, müssen Sie als Erstes das Modul `sqlite3` importieren. Danach wird eine Verbindung zur Datenbank hergestellt. Beide Schritte können Sie für die Datenbank-Datei `buecher.db` mit dem folgenden Code veranlassen:

```
1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
```

Die Dateiendung des Datenbanknamens ist frei wählbar: Die Endungen `.csv`, `.doc` oder `.xls` sind hier etwa auch möglich. Achten Sie darauf, dass sich die gewünschte Datenbank im selben Verzeichnis wie Ihr Python-Programm befindet. Andernfalls wird dort eine neue, leere Datei angelegt. Die `connect`-Funktion nimmt die Datenbank

## 14.2 SQLite-Datenbanken erstellen und bearbeiten

auf, liest diese ein und verbindet sie mit unserem Connection-Objekt `verbindung`, dessen Name ebenfalls frei wählbar ist.

Da wir nun eine leere Datenbankdatei angelegt haben, möchten wir sie als Nächstes mit dem Inhalt unserer drei Tabellen *Buch*, *Nutzer\_in* und *Ausleihen* füllen. Die drei Tabellen sollen die folgenden Einträge enthalten:

<b>ISBN-Nummer</b>	<b>Titel</b>	<b>Autor_in</b>	<b>Erschienen</b>	<b>Verlag</b>
3518469304	Meine geniale Freundin	Ferrante, Elena	2018	Suhrkamp Verlag
3596129974	Verbrechen und Strafe	Dostojewskij, Fjodor M.	1996	FISCHER Taschenbuch
3596163587	Die Brüder Karamasow	Dostojewskij, Fjodor M.	2006	FISCHER Taschenbuch
3499228548	Sehr blaue Augen	Morrison, Toni	1979	Rowohlt Taschenbuch
1234567890	Python 3	Schmitt, Sarah	2020	BMU Verlag

**Tab. 14.1** Inhalt der Tabelle *Buch*

<b>Nutzer-ID</b>	<b>Vorname</b>	<b>Nachname</b>
1	Amro	Mustermann
5	Cassidy	Hatcher
7	Christiane	Müller

**Tab. 14.2** Inhalt der Tabelle *Nutzer\_in*

<b>Nutzer-ID</b>	<b>ISBN-Nummer</b>
1	1234567890
5	3499228548
5	3518469304
7	3596163587

**Tab. 14.3** Inhalt der Tabelle *Ausleihen*

Da die Nutzer-ID „5“ in der Tabelle *Ausleihen* doppelt vorkommt, kann sie in dieser Tabelle nicht als eindeutiger Schlüssel gewählt werden – diese Funktion erfüllt hier die ISBN-Nummer.

Auf die drei Tabellen wollen wir die CRUD-Operationen anwenden. Außerdem möchten wir wie im vorigen Abschnitt erwähnt Verknüpfungen zwischen den drei Tabellen vornehmen können – zum Beispiel möchten wir wissen, welche Bücher die Nutzerin mit der Nutzer-ID 7 ausgeliehen hat oder ob ein bestimmtes Werk von Dostojewskij entliehen ist. Hierfür müssen wir einzelne Datensätze auswählen und innerhalb unserer Python-Anwendung anzeigen lassen können.

## 14 Mit Datenbanken arbeiten

Mit dem Code

```
1 cursor = verbindung.cursor()  
2 cursor.execute()
```

erzeugen wir im ersten Schritt ein `cursor`-Objekt, das auf die aktuelle Bearbeitungsposition in der Datenbank verweist. Anschließend können wir mithilfe des `cursor`-Objekts SQLite-Operationen und -abfragen von Python aus tätigen, wofür die `execute`-Methode zum Einsatz kommt. Innerhalb ein und derselben Datenbank lassen sich beliebig viele Cursors setzen.

### 14.2.1 SQLite-Tabellen erstellen

Nun können wir durch den SQLite-Befehl `CREATE TABLE` eine neue Tabelle erstellen. Zur einfacheren Lesbarkeit und Unterscheidung vom übrigen Python-Code verwendet man für Codeelemente, die aus SQL stammen, vorzugsweise Großbuchstaben. Dies ist jedoch nicht zwingend erforderlich.

Der SQLite-Befehl zum Erzeugen unserer neuen *Buch*-Tabelle mit den gewünschten Attributen sieht folgendermaßen aus:

```
1 CREATE TABLE buch(isbn INTEGER, titel TEXT, autor_in TEXT, erscheinungsjahr  
2 INTEGER, verlag TEXT)
```

Diese gesamte Codezeile wird nun durch die oben erwähnte `execute`-Methode an die SQLite-Datenbank geschickt. Der SQLite-Befehl steht dabei innerhalb von drei Anführungszeichen, um ihn leichter als SQLite-Code zu kennzeichnen. Es ist jedoch auch möglich, lediglich nur jeweils ein Anführungszeichen zu verwenden.

```
1 cursor.execute("""CREATE TABLE buch(isbn INTEGER, titel TEXT, autor_in TEXT,  
2 erscheinungsjahr INTEGER, verlag TEXT)""")
```

Wie Sie womöglich bereits erkannt haben, geben die Wörter hinter den Attributnamen den jeweiligen Datentyp der Spalte an. Die Bezeichnungen der Datentypen in SQLite unterscheiden sich von denjenigen in Python und müssen nach dem obigen Schema festgelegt werden, um umgewandelt werden zu können. Die folgende Tabelle gibt einen Überblick über die möglichen Python-Datentypen und deren Entsprechungen in SQLite:

## 14.2 SQLite-Datenbanken erstellen und bearbeiten

Python-Datentyp	SQLite-Datentyp
None	NULL
int	INTEGER
float	REAL
str	TEXT
bytes	BLOB

**Tab. 14.4** Umwandlung von Python- in SQLite-Datentypen

Selbstverständlich lassen sich auch andere Python-Datentypen wie Tupel, Listen oder selbst definierte Klassen in SQLite überführen. Hierbei werden die vorhandenen SQLite-Datentypen verwendet, um die entsprechenden Informationen abzubilden. Obwohl wir in diesem Buch nicht genauer auf die Konversionsmethodik eingehen werden, sollten Sie diese Möglichkeit für fortgeschrittenere Arbeitsvorhaben im Hinterkopf behalten.

Nachdem wir die erste Tabelle angelegt haben, wiederholen wir den Vorgang für unsere beiden anderen Tabellen *Nutzer\_in* und *Ausleihen*:

```
1 cursor.execute("""CREATE TABLE nutzer_in(id INTEGER PRIMARY KEY, vorname TEXT,
2 nachname TEXT)""")
3 cursor.execute("""CREATE TABLE ausleihen(id INTEGER, isbn INTEGER)""")
```

In der Tabelle *Nutzer\_in* können wir durch den optionalen PRIMARY KEY zusätzlich angeben, dass es sich bei *id* um den eindeutigen Schlüssel zur Identifikation der Bibliotheksmitglieder handelt. In dieser Spalte dürfen somit keine zwei Einträge mit demselben Wert vorkommen.

Als Nächstes möchten wir die Spalten mit den gewünschten Attributwerten füllen. Dazu wollen wir zunächst die erste Zeile in unsere *Buch*-Tabelle aufnehmen. Das geschieht durch den Befehl

```
1 cursor.execute("""INSERT INTO buch VALUES(3518469304, "Meine geniale
2 Freundin", "Ferrante, Elena", 2018, "Suhrkamp Verlag")""")
```

Nach dem Einfügen von Daten in die Tabelle sollten Sie diese jedes Mal durch den Befehl

```
1 verbindung.commit()
```

dauerhaft abspeichern, da sie ansonsten nicht direkt in unsere Datenbank *buecher.db* geschrieben und stattdessen im Arbeitsspeicher liegen bleiben würden. Auch weitere

## 14 Mit Datenbanken arbeiten

Daten, die Sie anschließend einfügen, werden nur nach Angabe eines neuen `commit`-Befehls gespeichert.

Es lassen sich auch mehrere Zeilen mit ein und derselben `INSERT INTO`-Abfrage behandeln. Um die restlichen vier Zeilen unserer `Buch`-Tabelle in die Datenbank aufzunehmen, speichern wir die Informationen zunächst als Liste mit vier Tupeln ab, die die Informationen aus den vier Zeilen enthalten:

```
1 weitere_zeilen = [(3596129974, "Verbrechen und Strafe", "Dostojewskij,
2 Fjodor M.", 1996, "FISCHER Taschenbuch"), (3596163587, "Die Brüder Karamasow",
3 "Dostojewskij, Fjodor M.", 2006, "FISCHER Taschenbuch"), (3499228548,
4 "Sehr blaue Augen",
5 "Morrison, Toni", 1979, "Rowohlt Taschenbuch"),
6 (1234567890, "Python 3", "Schmitt, Sarah", 2020, "BMU Verlag")]
```

Die gesamte Liste fügen wir nun mithilfe der Methode `executemany` nach dem folgenden Schema in die Tabelle ein:

```
1 cursor.executemany("INSERT INTO buch VALUES(?, ?, ?, ?, ?)", weitere_zeilen)
2 verbindung.commit()
```

Die Anzahl der Fragezeichen hinter `VALUES` muss dabei der Spaltenanzahl unserer Tabelle entsprechen.

Wir gehen nach demselben Schema vor, um unsere beiden anderen Tabellen anzulegen. Der komplette Code hat bis zu diesem Punkt die folgende Form:

### Codebeispiel #1

```
1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5
6 # Tabellen anlegen
7 cursor.execute("""CREATE TABLE IF NOT EXISTS buch(isbn INTEGER, titel TEXT,
8 autor_in TEXT, erscheinungsjahr INTEGER, verlag TEXT)""")
9 cursor.execute("""CREATE TABLE IF NOT EXISTS nutzer_in(id INTEGER, vorname
10 TEXT, nachname TEXT)""")
11 cursor.execute("""CREATE TABLE IF NOT EXISTS ausleihen(id INTEGER, isbn
12 INTEGER)""")
13
14 # Inhalt Tabelle 1
15 cursor.execute("""INSERT INTO buch VALUES(3518469304, "Meine geniale
16 Freundin", "Ferrante, Elena", 2018, "Suhrkamp Verlag")""")
17 verbindung.commit()
18 weitere_zeilen = [(3596129974, "Verbrechen und Strafe", "Dostojewskij,
19 Fjodor M.", 1996, "FISCHER Taschenbuch"), (3596163587, "Die Brüder Karamasow",
20 "Dostojewskij, Fjodor M.", 2006, "FISCHER Taschenbuch"), (3499228548, "Sehr
21 blaue Augen", "Morrison, Toni", 1979, "Rowohlt Taschenbuch"), (1234567890,
```

## 14.2 SQLite-Datenbanken erstellen und bearbeiten

```

22 "Python 3", "Schmitt, Sarah", 2020, "BMU Verlag")]
23 cursor.executemany("INSERT INTO buch VALUES (?,?,?,?,?)", weitere_zeilen)
24 verbindung.commit()
25
26 # Inhalt Tabelle 2
27 nutzer_innen = [(1, "Amro", "Mustermann"), (5, "Cassidy", "Hatcher"), (7,
28 "Christiane", "Müller")]
29 cursor.executemany("INSERT INTO nutzer_in VALUES (?,?,?)", nutzer_innen)
30
31 # Inhalt Tabelle 3
32 ausleihdaten = [(1, 1234567890), (5, 3499228548), (5, 3518469304), (7,
33 3596163587)]
34 cursor.executemany("INSERT INTO ausleihen VALUES (?,?)", ausleihdaten)
35 verbindung.commit()
36 verbindung.close()

```

Wenn die Tabellen richtig erstellt wurden, werden Sie in PyCharm keinerlei Output erhalten. Sie können in Ihrem Arbeitsordner überprüfen, dass dort nun eine Datei namens *buecher.db* existiert.

Beachten Sie hierbei, dass der Python-Interpreter eine Fehlermeldung ausgibt, wenn das Programm nach dem erstmaligen Anlegen einer Tabelle durch CREATE TABLE erneut ausgeführt wird. Um dies zu vermeiden, haben wir dem Befehl CREATE TABLE den Zusatz IF NOT EXISTS hinzugefügt.

Genauso wie wir unsere Daten im vorigen Kapitel stets mit `close()` abgespeichert hatten um beim Beenden des Programms unerwünschte Datenverluste zu vermeiden, sollten Sie diese Praxis ebenfalls beim Arbeiten mit SQLite beherzigen. Stellen Sie dabei jedoch sicher, dass Sie eventuelle Änderungen zuvor durch den `commit`-Befehl gespeichert haben.

### 14.2.2 Zeilen aus einer Tabelle auslesen

Nachdem wir unsere drei Tabellen erstellt haben, möchten wir daraus bestimmte Zeilen anzeigen lassen. Dabei sind vor allem die drei Schlüsselbegriffe `SELECT`, `FROM` und `WHERE` von Bedeutung. Mit diesen drei Befehlen lassen sich die wesentlichsten Operationen für Datenbankabfragen durchführen. Durch `SELECT` können Sie aus den vorliegenden Zeilen eine oder mehrere Spalten auswählen. Mehrfachnennungen werden jeweils durch Kommata voneinander getrennt. Folgt ein `*`-Zeichen auf `SELECT`, werden hingegen alle Spalten angezeigt. Mit `FROM` wird angegeben, auf welche Tabelle die Datenabfrage verweisen soll. Mit dem optionalen Schlüsselwort `WHERE` lassen sich durch die Angabe einer Bedingung bestimmte Datensätze innerhalb einer Datenbank auswählen und anzeigen.

## 14 Mit Datenbanken arbeiten

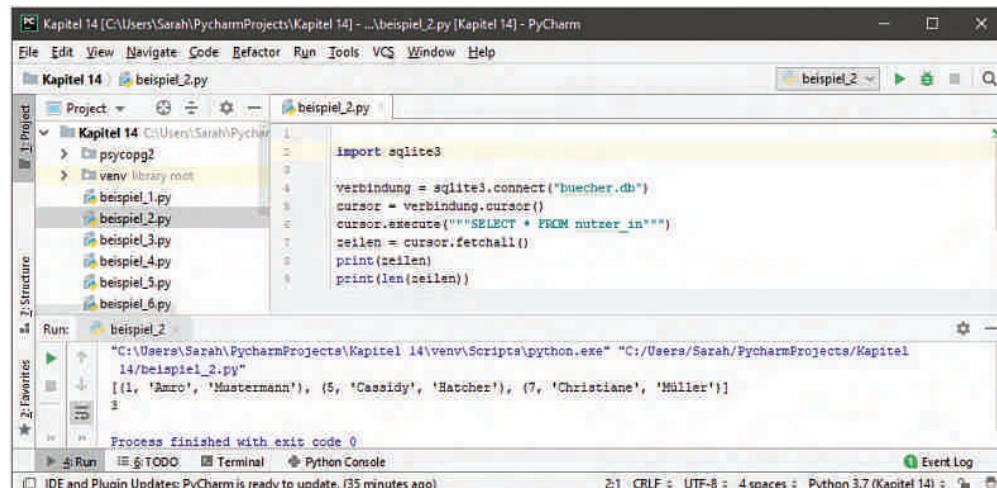
Der folgende Code gibt beispielsweise alle Datensätze aus der Tabelle `nutzer_in` aus und zählt in der letzten Zeile zudem ab, aus wie vielen Datensätzen die Tabelle besteht:

### Codebeispiel #2

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 cursor.execute("""SELECT * FROM nutzer_in""")
6 zeilen = cursor.fetchall()
7 print(zeilen)
8 print(len(zeilen))

```



**Abb. 14.1** Tabelleninhalte anzeigen lassen

Durch `cursor.fetchall` werden alle Datensätze, die unsere Abfrage ermittelt hat, auf einmal ausgegeben. Wie Sie am Output erkennen können, ist das Ergebnis eine Liste, innerhalb jeder Datensatz wiederum in einem Tupel steht. Insbesondere bei größeren Datenmengen kann diese Ausgabemethode jedoch sehr viel Speicherplatz beanspruchen, weshalb sie in diesem Fall ungeeignet ist. Bei großen Datenmengen können Sie die Daten stattdessen nach der folgenden Methode, in der jeder Datensatz als eigenes Tupel ausgegeben wird, zeilenweise abfragen:

### Codebeispiel #3

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()

```

## 14.2 SQLite-Datenbanken erstellen und bearbeiten

```

5     cursor.execute("""SELECT * FROM nutzer_in""")
6     for zeile in cursor:
7         print(zeile)

```

The screenshot shows the PyCharm IDE interface. The project is named 'Kapitel 14'. The 'Run' tab is active, showing the command: "C:\Users\Sarah\PycharmProjects\Kapitel\_14\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel\_14/beispiel\_3.py". The output window displays the following data:

```

1, 'Amro', 'Mustermann'
5, 'Cassidy', 'Hatcher'
7, 'Christiane', 'Müller'

```

**Abb. 14.2** Tabelleninhalte zeilenweise auslesen

Im nächsten Beispiel wird durch SELECT die ID-Nummer sowie der Vorname ausgewählt, während WHERE die Auswahl auf diejenigen Zeilen mit dem Nachnamen „Hatcher“ begrenzt:

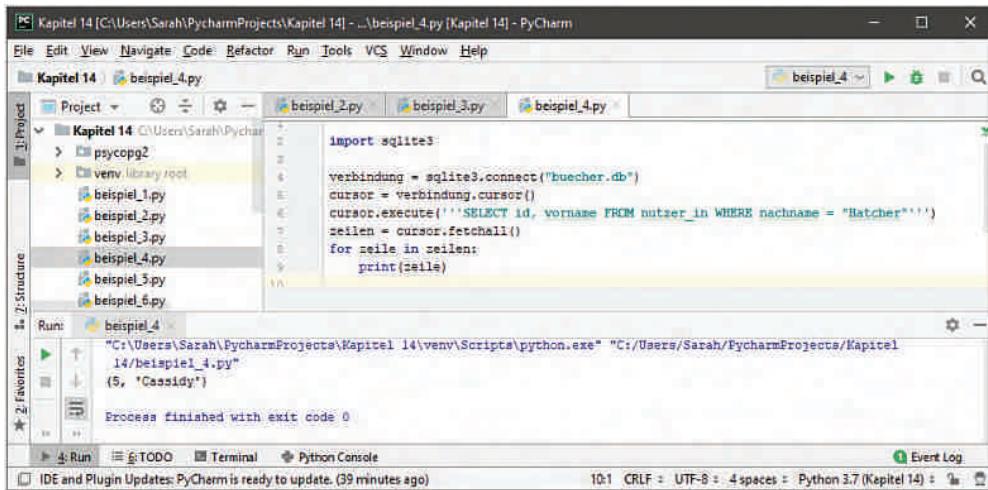
### Codebeispiel #4

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 cursor.execute('''SELECT id, vorname FROM nutzer_in WHERE nachname =
6 "Hatcher"''')
7 zeilen = cursor.fetchall()
8 for zeile in zeilen:
9     print(zeile)

```

## 14 Mit Datenbanken arbeiten



**Abb. 14.3** Bedingungen für eine Ausgabe mit WHERE angeben

Wir verwenden in der vierten Codezeile einfache Anführungszeichen, um die Unterscheidung zwischen dem Attributwert und dem SQLite-Befehl deutlicher zu machen.

Eine wichtige und sehr nützliche Funktionalität von Datenbanken ist die Möglichkeit, Verknüpfungen zwischen Tabellen – sogenannte *Joins* – vorzunehmen, was Abfragen über mehrere Tabellen ermöglicht. Dadurch können wir etwa herausfinden, welche Titel aus unserer *Buch*-Tabelle derzeit entliehen sind. Die Identifikation erfolgt in diesem Beispiel über die ISBN-Nummer:

### Codebeispiel #5

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 cursor.execute("""SELECT ausleihen.isbn, buch.titel, buch.autor_in FROM buch,
6 ausleihen WHERE buch.isbn=ausleihen.isbn""")
7 zeilen = cursor.fetchall()
8 print("Die folgenden Titel sind entliehen:")
9 for zeile in zeilen:
10     print(zeile)
11 verbindung.close()

```

## 14.2 SQLite-Datenbanken erstellen und bearbeiten

```
verbindung = sqlite3.connect('Buecher.db')
cursor = verbindung.cursor()
cursor.execute("""SELECT ausleihen.isbn, buch.titel, buch.autor_in FROM buch,
ausleihen WHERE buch.isbn=ausleihen.isbn""")
zeilen = cursor.fetchall()
print("Die folgenden Titel sind entliehen:")
for seile in zeilen:
    print(seile)
verbindung.close()
```

Run: beispiel\_5

```
Die folgenden Titel sind entliehen:
(3518465304, 'Meine geniale Freundin', 'Ferrante, Elena')
(3596163587, 'Die Brüder Karamasow', 'Dostojewskij, Fjodor M.')
(3499228548, 'Sehr blaue Augen', 'Morrison, Toni')
(1234567890, 'Python 3', 'Schmitt, Sarah')
```

**Abb. 14.4** Informationen aus mehreren Tabellen verknüpfen

Wenn Sie über mehrere Tabellen hinweg Daten abfragen möchten, müssen Sie hinter dem **FROM**-Begriff alle Tabellen auflisten, die von Ihrer Abfrage betroffen sind. Außerdem muss für die Auswahl bestimmter Spalten nach **SELECT** oder **WHERE** der entsprechende Tabellenname ebenfalls vor den gewünschten Spaltenbezeichnungen angegeben werden.

Um sich alle in einer Datenbank enthaltenen Tabellen anzeigen zu lassen, kommt der Ausdruck `sqlite_master` innerhalb eines SQLite-Befehls mit `SELECT`, `FROM` und `WHERE` zum Einsatz:

## *Codebeispiel #6*

```
1 import sqlite3  
2  
3 verbindung = sqlite3.connect("buecher.db")  
4 cursor = verbindung.cursor()  
5 cursor.execute(''':SELECT name FROM sqlite_master WHERE type= "table"'''')  
6 print(cursor.fetchall())
```

Nach diesem festen Schema können Sie sich stets alle Datenbank-Tabellen ausgeben lassen. Weiterhin besteht auch die Möglichkeit, zu überprüfen, ob eine bestimmte Tabelle existiert. Ist dies der Fall, wird der Name der Tabelle angezeigt, andernfalls erscheint eine leere Liste:

```
1 cursor.execute(''SELECT name FROM sqlite_master WHERE type= "table" AND name
2 = "buch"''')
3 print(cursor.fetchall())
```

## 14 Mit Datenbanken arbeiten

The screenshot shows the PyCharm IDE interface. The project is named 'Kapitel 14'. The code editor displays a file named 'beispiel\_6.py' containing SQLite queries to check for tables. The run tab shows the output of the script, which includes the creation of tables 'nutzer\_in' and 'ausleihen', and a check for table 'buch'. The terminal tab shows the command run: "C:/Users/Sarah/PycharmProjects/Kapitel 14/beispiel\_6.py". The output in the terminal is: [('buch'), ('nutzer\_in'), ('ausleihen')]. The status bar at the bottom indicates Python 3.7 (Kapitel 14).

```

import sqlite3
verbindung = sqlite3.connect("buecher.db")
cursor = verbindung.cursor()
cursor.execute(''':SELECT name FROM sqlite_master WHERE type= "table"''')
print(cursor.fetchall())

cursor.execute(''':SELECT name FROM sqlite_master WHERE type= "table" AND name = "buch"'''')
print(cursor.fetchall())

```

**Abb. 14.5** Anzeigen aller Tabellen und Überprüfen auf Existenz bestimmter Tabellen

Sie können die logischen Operatoren `AND` und `OR` verwenden, um darin mehrere Bedingungen für Datenbankabfragen zusammenzufassen.

### 14.2.3 SQLite-Tabellen aktualisieren

Sie wissen nun, wie man die ersten beiden *CRUD*-Operationen – *Erzeugen* und *Lesen* – für eine SQLite-Datenbank vornimmt. Als Nächstes möchten wir untersuchen, wie sich bereits erstellte Tabelleninhalte aktualisieren lassen. Hierfür kommt das Schlüsselwort `UPDATE` zum Einsatz. Im untenstehenden Code wird mit `UPDATE`, `SET` und `WHERE` die ISBN-Nummer des Buchs „Python 3“ abgeändert:

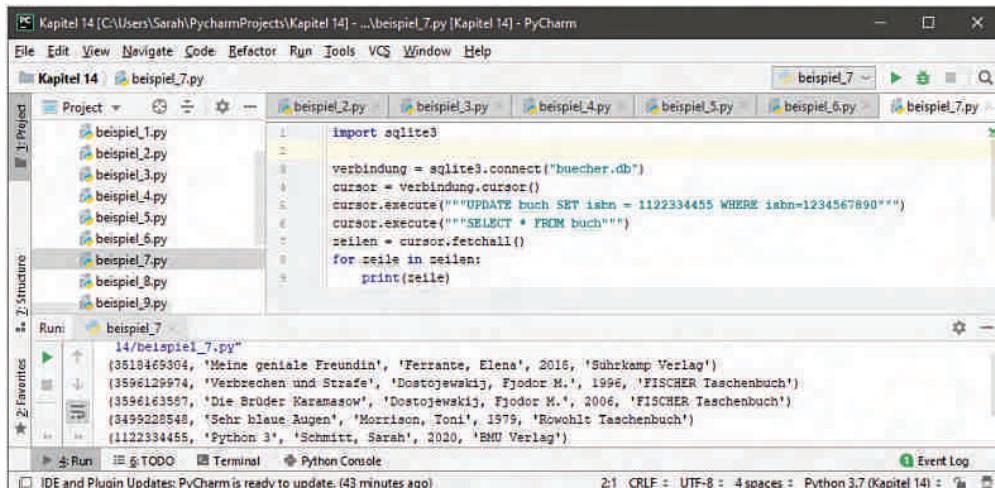
#### Codebeispiel #7

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 cursor.execute("""UPDATE buch SET isbn = 1122334455 WHERE isbn=1234567890""")
6 cursor.execute("""SELECT * FROM buch""")
7 zeilen = cursor.fetchall()
8 for zeile in zeilen:
9     print(zeile)
10 verbindung.commit()
11 verbindung.close()

```

## 14.2 SQLite-Datenbanken erstellen und bearbeiten



The screenshot shows the PyCharm IDE interface. The project is named 'Kapitel 14'. The 'beispiel\_7.py' file is open in the editor. The code uses Python's sqlite3 module to connect to a database named 'buecher.db', execute an UPDATE query to change the ISBN of a book, and then print all rows from the 'buch' table. The 'Run' tab shows the output of the script, which lists several books with their details. The bottom status bar indicates the script was run at 2:11, using CRLF line endings, UTF-8 encoding, 4 spaces indentation, and Python 3.7.

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 cursor.execute("""UPDATE buch SET isbn = 1122334455 WHERE isbn=1234567890""")
6 cursor.execute("""SELECT * FROM buch""")
7 zeilen = cursor.fetchall()
8 for zeile in zeilen:
9     print(zeile)

```

```

14/beispiel_7.py"
(3818465304, 'Meine geniale Freundin', 'Ferrante, Elena', 2016, 'Suhrkamp Verlag')
(3596120974, 'Verbrechen und Strafe', 'Dostojewskij, Fjodor M.', 1896, 'FISCHER Taschenbuch')
(3596163567, 'Die Brüder Karamasow', 'Dostojewskij, Fjodor M.', 2006, 'FISCHER Taschenbuch')
(3499228548, 'Sehr blaue Augen', 'Morrison, Toni', 1979, 'Rowohlt Taschenbuch')
(1122334455, 'Python 3', 'Schmitt, Sarah', 2020, 'BMU Verlag')

```

**Abb. 14.6** Tabelleninhalte aktualisieren

### 14.2.4 Tabelleninhalte löschen

Tabelleninhalte können durch den DELETE-Befehl gelöscht werden. Wenn Sie die Änderungen nach dem Löschkvorgang dauerhaft in der Datenbank speichern möchten, müssen Sie auch hier den commit-Befehl verwenden, nachdem Sie die gewünschten Tabelleninhalte gelöscht haben. Sie können einmal beide Varianten – durch Aus- und Einkommentieren des commit-Befehls in der vorletzten Zeile – ausprobieren um sich den Unterschied bei mehrmaligem Programmaufruf verdeutlichen zu lassen. Beachten Sie dazu die Ausgabe des folgenden Codes.

#### Codebeispiel #8

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 cursor.execute("""DELETE FROM ausleihen""")
6 print("Es wurden {} Zeilen gelöscht.".format(cursor.rowcount))
7 cursor.execute("""DELETE FROM ausleihen""")
8 print("Es wurden {} Zeilen gelöscht.".format(cursor.rowcount))
9 # verbindung.commit()
10 verbindung.close()

```

## 14 Mit Datenbanken arbeiten

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_1.py' through 'beispiel\_9.py'. Below it, the 'Project' tool window shows a tree structure with 'beispiel\_1.py' through 'beispiel\_9.py'. The main code editor window displays Python code for connecting to a SQLite database and deleting rows from a table named 'ausleihen'. The code uses the `execute` method with SQL commands like `DELETE FROM ausleihen` and `SELECT isbn FROM ausleihen`. The run tool window at the bottom shows the command to run the script and its output, which indicates 4 rows were deleted and 0 rows remain.

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 cursor.execute("""DELETE FROM ausleihen""")
6 print("Es wurden {} Zeilen gelöscht.".format(cursor.rowcount))
7 cursor.execute("""DELETE FROM ausleihen""")
8 print("Es wurden {} Zeilen gelöscht.".format(cursor.rowcount))
9 # verbindung.commit()

```

**Abb. 14.7** Tabelleninhalte löschen

Mit `rowcount` wird hierbei abgezählt, wie viele Zeilen der *Ausleihen*-Tabelle gelöscht wurden. Allgemein gibt `rowcount` die Anzahl derjenigen Zeilen aus, die mit dem letzten SQLite-Befehl bearbeitet wurden. Die einzige Ausnahme bildet hier der `SELECT`-Befehl, bei dem Sie wie in Abschnitt 14.2.2. beschrieben zunächst alle Daten mit `fetchall` abrufen müssen und sich die Zeilenanzahl anschließend mithilfe der `len`-Funktion anzeigen lassen können.

Einzelne Zeilen aus einer Tabelle lassen sich durch die zusätzliche Angabe von `WHERE` löschen. Zum Beispiel können Sie die Code-Zeile mit der `execute`-Methode folgendermaßen abändern, um lediglich die Zeile mit der ISBN-Nummer 3596163587 zu löschen:

```
1 cursor.execute("""DELETE FROM ausleihen WHERE isbn=3596163587""")
```

In den Übungsaufgaben werden Sie die Gelegenheit haben, dies selbst zu testen.

Mit dem Schlüsselwort `DROP` können Sie ganze Tabellen löschen:

### Codebeispiel #9

```

1 import sqlite3
2
3 import sys
4 verbindung = sqlite3.connect("buecher.db")
5 cursor = verbindung.cursor()
6 cursor.execute("""DROP table IF EXISTS ausleihen""")
7 try:
8     cursor.execute("""SELECT isbn FROM ausleihen""")
9 except Exception as e:

```

## 14.3 NoSQL-Datenbanken

```

10     print(e, sys.exc_info()[0])
11 verbindung.close()

```

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_7.py', 'beispiel\_8.py', 'beispiel\_9.py', 'buecher.db', 'Übung\_1.py', 'Übung\_2.py', and 'Übung\_3.py'. Below this is a code editor window containing Python code. The code attempts to drop a table named 'ausleihen' from a SQLite database and handle any errors. The code is:

```

10     print(e, sys.exc_info()[0])
11 verbindung.close()

```

When run, the terminal output shows:

```

"C:\Users\Sarah\PycharmProjects\Kapitel 14\venv\Scripts\python.exe" "C:/Users/Sarah/PycharmProjects/Kapitel 14/beispiel_9.py"
no such table: ausleihen <class 'sqlite3.OperationalError'>
Process finished with exit code 0

```

**Abb. 14.8** Eine gesamte Tabelle löschen

Bei kritischen Datenbankmanipulationen wie der obigen, in der versucht wurde, auf bereits gelöschte Inhalte zuzugreifen, ist es stets eine gute Idee, eine `try ... except`-Anweisung für unvorhergesehene Ausnahmen vorzusehen. Die möglichen Ausnahmeklassen für SQLite sind `DatabaseError`, `IntegrityError`, `ProgrammingError`, `OperationalError` sowie `NotSupportedError`. Durch das Importieren des `sys`-Moduls können wir uns im obigen Code durch die Methode `exc_info` ausgeben lassen, dass es sich bei der vorliegenden Ausnahme um einen `OperationalError` handelt.

## 14.3 NoSQL-Datenbanken

Die meisten Daten aus Webanwendungen werden in relationalen Datenbanken gespeichert. Es gibt jedoch auch andere, nicht-relationale Datenbanksysteme für die Datenspeicherung, die allgemein als NoSQL-Datenbankmodelle bezeichnet werden. Die Abkürzung NoSQL bedeutet „Not only SQL“ (engl. für „Nicht nur SQL“). NoSQL-Datenbanksysteme können ebenfalls SQL-ähnliche Sprachen unterstützen und lassen sich dementsprechend auch einsetzen, um die Funktionalität relationaler Datenbanken zu erweitern.

14

Der Bedarf an NoSQL-Datenbanken kam gegen Ende der 2000er Jahre verstärkt auf, als die Datenvolumen in vielen modernen Anwendungen des Web 2.0 zunahmen. Vor allem in datenintensiveren Cloud-, Big-Data- und Echtzeit-Anwendungen kommen NoSQL-Datenbanken sehr häufig zum Einsatz, da sich diese Anwendungen mit SQL meist nicht effizient genug handhaben lassen. NoSQL-Datenbanken können Objekt-

## 14 Mit Datenbanken arbeiten

orientierung unterstützen und ermöglichen insbesondere eine flexiblere Speicherung, da sich damit auch uneinheitliche, variierende Datenmengen mit semistrukturierten oder unstrukturierten Daten verarbeiten lassen.

NoSQL-Speicheroptionen umfassen unter anderem:

- ▶ Schlüssel-Werte-Datenbanken
- ▶ Dokumentenorientierte Datenbanken
- ▶ Spaltenorientierte Datenbanken
- ▶ Graphdatenbanken

Der Einsatz dieser NoSQL-Datenbanksysteme wirkt sich in der Regel insofern auf die Arbeitsleistung aus, dass sich beispielsweise bestimmte Lese- und Schreiboperationen effizienter gestalten lassen, während andere Vorgänge weniger effizient sind. Wir wollen Ihnen an dieser Stelle einen ersten Einblick in die unterschiedlichen Funktionsweisen dieser vier Speicheroptionen geben, ohne auf deren konkrete praktische Anwendung einzugehen.

### Schlüssel-Werte-Datenbanken

Diese einfachsten aller NoSQL-Datenbanken sind in Schlüssel-Wert-Paaren strukturiert. Ihre Werte können unter anderem Tupel, Strings oder Mengen sein und lassen sich durch einen eindeutigen Schlüssel identifizieren und abrufen. Aufgrund ihres einfachen Aufbaus bieten Schlüssel-Werte-Datenbanken eine effiziente und schnelle Datenverarbeitung. Sie werden gerne in webbasierten Anwendungen zur Handhabung von Sitzungsmanagement-Prozessen eingesetzt. Eine einfache Schlüssel-Wert-Datenbank für Python ist beispielsweise *pickleDB*.

### Dokumentenorientierte Datenbanken

In dokumentenorientierten Datenbanken ist jedem Schlüssel (bzw. Identifikator) ein eigenes Dokument zugewiesen. Die Dokumente können dabei unterschiedliche Formatierungen aufweisen und wiederum Schlüssel-Wert-Paare, Schlüssel-Listen-Paare oder eingebettete Dokumente mit semistrukturierten Daten enthalten. Dokumentenorientierte Datenbanken sind dadurch sehr viel flexibler als andere Datenbanksysteme. Sie werden vor allem für das Content-Management und zur Verarbeitung von Daten aus mobilen Anwendungen eingesetzt. In Kombination mit Python können Sie zum Beispiel *TinyDB*, eine kompakte, einfache dokumentenorientierte Datenbank, verwenden.

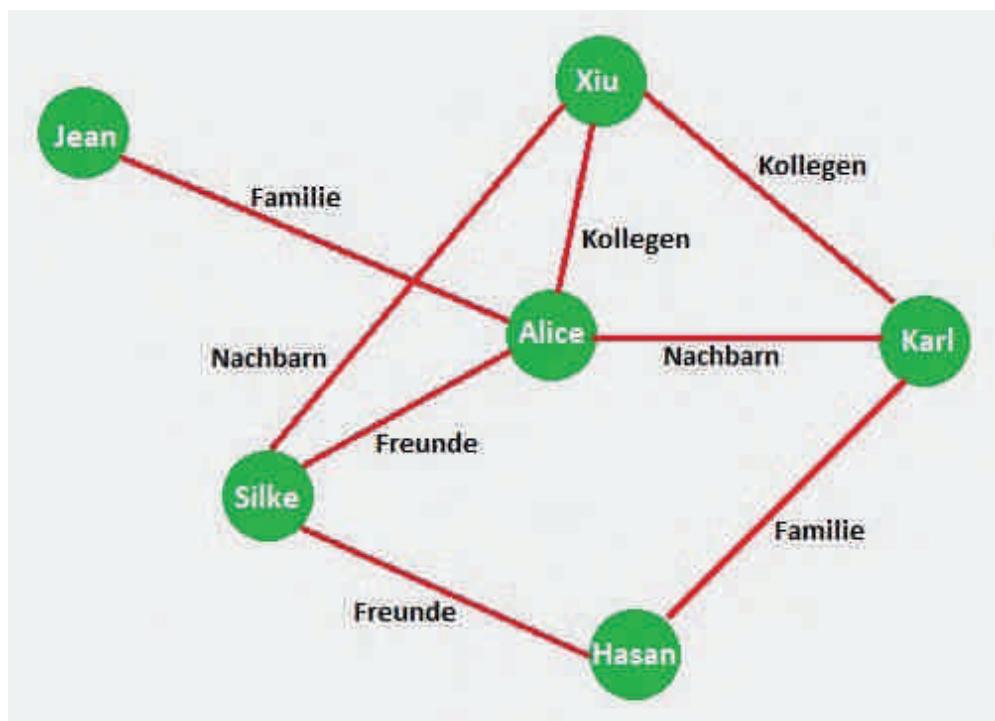
### Spaltenorientierte Datenbanken

Mit diesem Datenbanksystem lassen sich insbesondere große Datenvolumen schneller abfragen, was an dessen besonderem Speicherungstyp liegt: Im Gegensatz zu re-

lationalen Datenbanken werden die Daten in einer spaltenorientierten Datenbank nicht zeilen-, sondern spaltenweise gespeichert. Dies erleichtert oftmals den gezielten Zugriff auf gewünschte Inhalte, wodurch sich wiederum Datenanalysen beschleunigen lassen. Ein Beispiel einer spaltenorientierten Datenbank-Implementierung ist *MonetDB*.

### Graphdatenbanken

Graphdatenbanken enthalten Informationen über Datennetzwerke. Anwendungsbeispiele sind etwa das Nutzerdaten-Management und Reservierungssysteme. Die Graphen repräsentieren dabei Straßennetze, Transportwege oder soziale Netzwerke, und bestehen aus den Verbindungen bzw. Beziehungen zwischen den einzelnen Orten oder Individuen.



**Abb. 14.9** Beispiel eines einfachen Graphen

Die Orte bzw. Individuen sind dabei die Knoten des Graphen. Sie sind vergleichbar mit den Datensätzen relationaler Datenbanken. Die Kanten des Graphen sind die Verbindungen zwischen den Knoten und drücken deren Beziehungen zueinander aus. Somit lassen sich die Zusammenhänge zwischen den Daten veranschaulichen. Mit Python können Sie beispielsweise von der Graphdatenbank *Neo4j* Gebrauch machen.

## 14 Mit Datenbanken arbeiten

Um in Python mit NoSQL-Datenbanken arbeiten zu können, müssen Sie beispielsweise den beliebten *PyMongo*-Treiber sowie die *MongoDB*-Datenbank – eine dokumentenorientierte Datenbank – installieren. Anschließend lassen sich mit den Daten die gewünschten CRUD-Operationen durchführen.

### 14.4 XML-Datenbanken

XML steht für *eXtensible Markup Language*. Dabei handelt es sich um eine standardisierte, software- und hardwareunabhängige Auszeichnungssprache, mithilfe derer Daten gegliedert, formatiert und zwischen verschiedenen Anwendungen ausgetauscht werden können. XML ist vor allem für Webanwendungen beliebt und bietet insbesondere hinsichtlich Flexibilität und Erweiterbarkeit in der Datenspeicherung viele Vorteile. Die Sprache kann sowohl von Menschen als auch von Maschinen gelesen werden und gilt aufgrund ihrer einfachen Syntax als leicht zu erlernen.

XML-Datenbanken zählen zu den dokumentenorientierten Datenbanken und daher zu den NoSQL-Datenbanksystemen. Sie ermöglichen die Speicherung großer Datens Mengen im XML-Format. Die Informationen einer XML-Datenbank können mit einer Abfragesprache, beispielsweise *XQuery*, durchsucht werden um anschließend Teile daraus zu extrahieren.

Dokumente, die im XML-Format gespeichert sind, haben die Endung *.xml*. Sie sind textbasiert und können mit einem gewöhnlichen Texteditor geöffnet werden. Es gibt jedoch auch eigene XML-Editoren, die speziell für das Editieren von XML-Dateien vorgesehen sind. Der Text eines XML-Dokuments ist durch sogenannte *Tags* gegliedert. Ein Tag besteht aus spitzen Klammern und kann – als Starttag – öffnend `<>` oder – als Endtag – schließend `</>` sein. Die Tags repräsentieren jeweils die einzelnen *Elemente*, die den Baustein von XML-Dokumenten bilden und diese strukturieren. Sie sind mit einem sie bezeichnenden Namen sowie optional mit weiteren Attributen versehen, die auf den Inhalt zwischen den beiden Tags verweisen. Dort wiederum sind die spezifischen Informationen bzw. Daten der XML-Datenbank enthalten. Zum Beispiel wird mit `<zah1>5</zah1>` das entsprechende *zahl*-Element mit dem Inhalt „5“ geöffnet und sogleich wieder geschlossen.

Es ist ebenfalls möglich, innerhalb eines Start- und eines Endtags weitere Tags zu schachteln oder längere Textinhalte vorzusehen. So kann XML-Code, der den Text *Hallo Welt!* enthält, die folgende Form haben:

```
1 <nachricht>
2   <text>Hallo Welt!</text>
3 </nachricht>
```

## 14.4 XML-Datenbanken

XML-Dokumente haben einen baumähnlichen Aufbau mit einem einzigen Wurzel-element (im obigen Beispiel: <nachricht>) und daraus verzweigenden Ästen, was hierarchische Datenstrukturen ermöglicht.

Um Ihnen den Aufbau von XML-Code weiter zu veranschaulichen, möchten wir nun die Tabelle *Nutzer-ID* aus Abschnitt 14.2. als einfaches XML-Dokument ausdrücken. Dabei ist <nutzer-id> das Wurzel-Tag, aus dem sich die einzelnen Nutzereinträge verzweigen:

```
1  <?xml version = "1.0"?>
2  <nutzer-id>
3      <_1>
4          <vorname>Amro</vorname>
5          <nachname>Mustermann</nachname>
6      </_1>
7      <_2>
8          <vorname>Cassidy</vorname>
9          <nachname>Hatcher</nachname>
10     </_2>
11     <_3>
12         <vorname>Christiane</vorname>
13         <nachname>Müller</nachname>
14     </_3>
15 </nutzer-id>
```

In der ersten Zeile steht hierbei die optionale XML-Deklaration, die unter anderem Informationen zur XML-Version und zur Zeichenkodierung beinhalten kann. Danach verzweigen sich aus dem Wurzel-Tag die einzelnen Datensätze mit allen weiteren Informationen. Da XML-Tag-Bezeichnungen nicht mit einer Zahl anfangen dürfen, wurde den ID-Zahlen jeweils ein Unterstrich vorangestellt.

In Python können Sie zum Lesen und Schreiben von XML-Dateien das Standardbibliothek-Modul `xml` importieren.

## 14 Mit Datenbanken arbeiten

### 14.5 Übung: Eine Datenbank mit SQLite anlegen und bearbeiten

#### Übungsaufgaben

1. Verwenden Sie die drei in 14.2. erstellten Tabellen *Buch*, *Nutzer\_in* und *Ausleihen*. Finden Sie durch eine Datenbankabfrage mit SQLite heraus, welche Titel von Dos-tojewskij entliehen sind und wer diese ausgeliehen hat. Lassen Sie sich die beiden Informationen durch `print`-Befehle ausgeben.
2. Lassen Sie sich die Anzahl der Zeilen der Tabelle *Ausleihen* anzeigen. Löschen Sie einen spezifischen Eintrag aus dieser Tabelle und lassen Sie sich anschließend die Anzahl der gelöschten Zeilen ausgeben.

**Tipp:** Überlegen Sie, für welchen Vorgang Sie `fetchall` und `len`, und für welchen Vorgang Sie `rowcount` verwenden müssen.

3. Schreiben Sie ein Programm, das auf eine Nutzereingabe hin der *Nutzer\_in*-Tabelle neue Nutzereinträge hinzufügt und diese abspeichert. Lesen Sie anschließend den Inhalt der Tabelle zeilenweise aus.

## 14.5 Übung: Eine Datenbank mit SQLite anlegen und bearbeiten

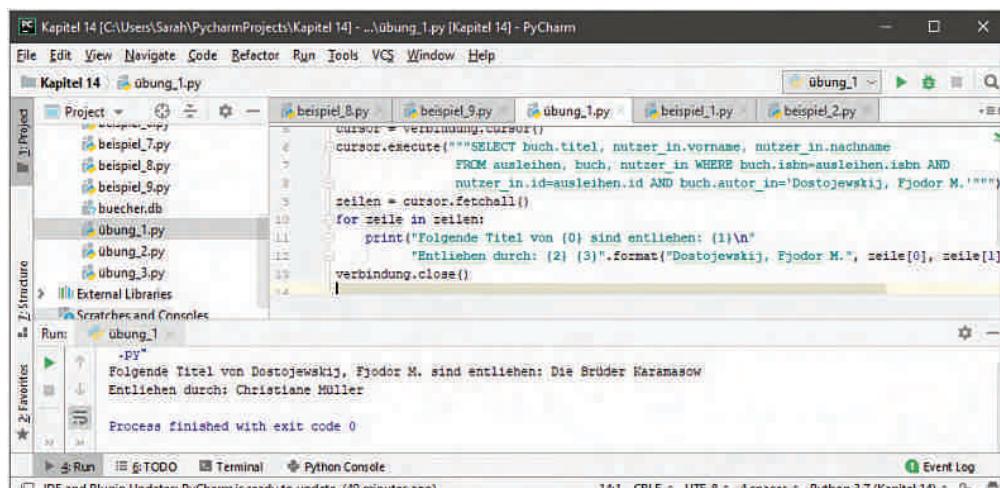
### Lösungen

1. Hier gilt es, die Datenbankabfrage möglichst geschickt in einem einzigen Befehl zu stellen. Anschließend werden die Informationen in einem Tupel abgespeichert und innerhalb des print-Befehls ausgegeben.

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 cursor.execute("""SELECT buch.titel, nutzer_in.vorname, nutzer_in.nachname
6 FROM ausleihen, buch, nutzer_in WHERE buch.isbn=ausleihen.isbn AND nutzer_
7 in.id=ausleihen.id AND buch.autor_in='Dostojewskij, Fjodor M.'""")
8 zeilen = cursor.fetchall()
9 for zeile in zeilen:
10     print("Folgende Titel von {0} sind entliehen: {1}\n"
11           "Entliehen durch: {2} {3}").format("Dostojewskij, Fjodor M.",
12 zeile[0], zeile[1], zeile[2]))
13 verbindung.close()

```



**Abb. 14.10** Eine Datenbankabfrage aus drei Tabellen ausgeben lassen

2. Bei einer Datenbankabfrage mit SELECT kann die Zeilenanzahl nur durch fetchall und len ermittelt werden. Nach dem Löschen der gewünschten Zeile mit WHERE können Sie sich durch rowcount die Anzahl der gelöschten Zeilen anzeigen lassen. Beachten Sie, dass Löschvorgänge erst durch verbindung.commit() dauerhaft gespeichert werden.

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 cursor.execute("""SELECT * FROM ausleihen""")
6 zeilen = cursor.fetchall()
7 print("Anzahl der Zeilen:", len(zeilen))

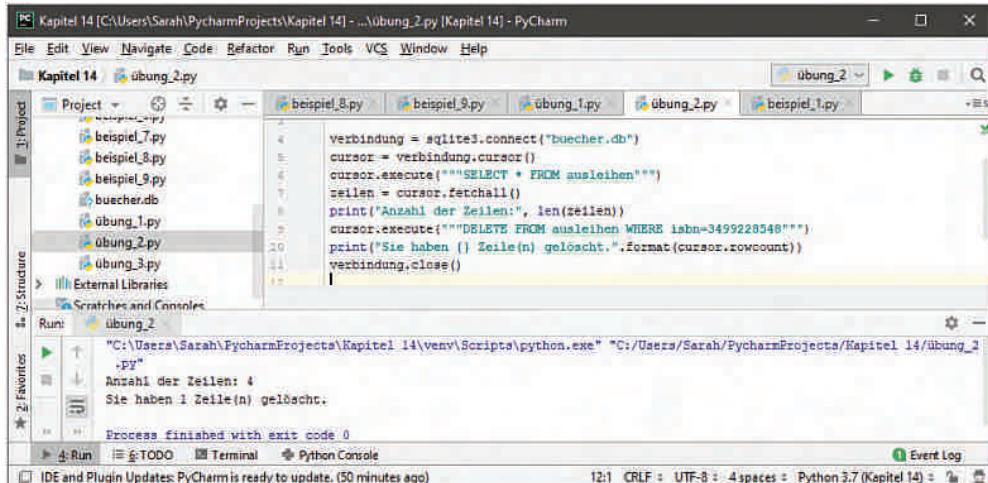
```

## 14 Mit Datenbanken arbeiten

```

8     cursor.execute("""DELETE FROM ausleihen WHERE isbn=3499228548""")
9     print("Sie haben {} Zeile(n) gelöscht.".format(cursor.rowcount))
10    verbindung.close()

```



**Abb. 14.11** Anzeigen und Löschen von Zeileneinträgen

3.

```

1 import sqlite3
2
3 verbindung = sqlite3.connect("buecher.db")
4 cursor = verbindung.cursor()
5 nutzer_id = input("Geben Sie die neue Nutzer-ID ein: ")
6 vorname = input("Geben Sie den Vornamen ein: ")
7 nachname = input("Geben Sie den Nachnamen ein: ")
8 nutzerin_neu = (nutzer_id, vorname, nachname)
9 cursor.execute("INSERT INTO nutzer_in VALUES(?, ?, ?)", nutzerin_neu)
10 verbindung.commit()
11 cursor.execute("""SELECT * FROM nutzer_in""")
12 for zeile in cursor:
13     print(zeile)
14 verbindung.close()

```

## 14.5 Übung: Eine Datenbank mit SQLite anlegen und bearbeiten

The screenshot shows the PyCharm IDE interface. The project navigation bar at the top lists files like 'beispiel\_7.py', 'beispiel\_8.py', 'beispiel\_9.py', 'huecher.db', 'übung\_1.py', 'übung\_2.py', 'übung\_3.py', and 'beispiel\_1.py'. The main code editor window displays a Python script named 'übung\_3.py' which contains the following code:

```
vorname = input("Geben Sie den Vornamen ein: ")
nachname = input("Geben Sie den Nachnamen ein: ")
nutzerin_neu = (nutzer_id, vorname, nachname)
cursor.execute("INSERT INTO nutzer_in VALUES(?, ?, ?)", nutzerin_neu)
verbindung.commit()
cursor.execute("""SELECT * FROM nutzer_in""")
for zeile in cursor:
    print(zeile)
verbindung.close()
```

The terminal output pane below the editor shows the execution of the script:

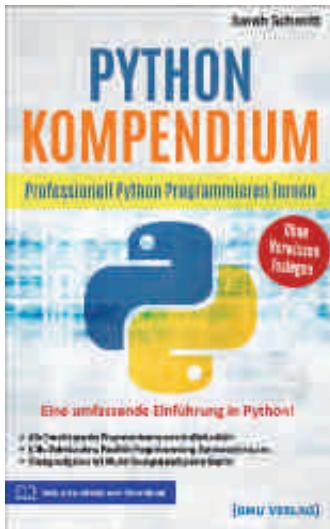
```
Geben Sie die neue Nutzer-ID ein: 2
Geben Sie den Vornamen ein: Nadja
Geben Sie den Nachnamen ein: Bien
(1, 'Anro', 'Mustermann')
(5, 'Cassidy', 'Hatcher')
(7, 'Christiane', 'Müller')
(2, 'Nadja', 'Bien')
```

The status bar at the bottom indicates the script is ready to update.

Abb. 14.12 Hinzufügen von Einträgen durch eine Nutzereingabe

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
Downloadcode: siehe Kapitel 18

## Kapitel 15

# Grafische Benutzeroberflächen mit PyQt erstellen

Sobald Sie Ihre erste längere Anwendung für den praktischen Gebrauch geschrieben haben, möchten Sie diese wahrscheinlich in einer handhabbaren und auch für andere Nutzer ansprechenden Form verwenden können. Für diesen Zweck gibt es *grafische Benutzeroberflächen* (auch *GUIs* genannt, von engl. *Graphical User Interface*). Im Vergleich zur Bedienung eines Programms aus der IDE oder dem Kommandozeileninterpreter heraus, sind GUIs insbesondere für Drittanwender einfacher zu bedienen und dadurch breiter einsetzbar. Außerdem erlauben sie als interaktive Schnittstelle unterschiedliche Eingabe- und Aktionsmöglichkeiten zwischen den Nutzerinnen bzw. Nutzern und dem Programm.

Obwohl Python meist nicht die erste Wahl für die Erstellung fortgeschrittenerer GUIs ist, kann es selbstverständlich durchaus sinnvoll sein, den eigenen Python-Code in einer interaktiveren, anwenderfreundlichen Form präsentieren und verwenden zu können. Je nach Anwendung, Betriebssystem und Anforderungen an die GUI kommen hierfür verschiedene GUI-Toolkits bzw. Frameworks infrage. Dabei handelt es sich um Programmabibliotheken, mithilfe derer sich GUIs für Desktop-Anwendungen erstellen lassen. Toolkits enthalten verschiedene Steuerelemente – *Widgets* genannt –, die die Interaktion zwischen den Nutzerinnen und Nutzern und dem Programm ermöglichen. Solche Widgets sind etwa Buttons, Textfelder, Dropdown-Listen oder Scroll-Balken.

Das Standard-GUI-Toolkit in Python, *Tk*, verwendet das ältere *Tkinter*-Paket. Dieses ist im Umfang der Standardbibliothek enthalten, bietet eine grundlegende Funktionalität und läuft auf Windows, Mac OS X sowie auf Linux. Es ist zwar sehr einfach in der Anwendung, verfügt jedoch über weniger Gestaltungsmöglichkeiten, ist nicht mehr zeitgemäß sowie auch optisch weniger ansprechend als viele andere Alternativen, die mittlerweile entwickelt wurden. Zu diesen zählen beispielsweise *wxPython*, *PyQt*, *Kivy* oder *PyGUI*. Sie sind jedoch nicht Teil der Python-Standardbibliothek und müssen aus externen Quellen von Drittanbietern heruntergeladen und installiert werden.

Für die Erstellung von GUIs in diesem Buch werden wir eines der beliebtesten und aktuell weitverbreitetsten dieser externen GUI-Modulpakete verwenden: *PyQt*. Wenn Sie dennoch das Tkinter-Paket aus der Python-Standardbibliothek benutzen möchten, so können Sie dieses ganz einfach durch den Befehl `import tkinter` importieren und direkt verwenden. Die weitere Dokumentation zu Tkinter finden Sie unter <https://docs.python.org/3/library/tk.html>.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

Die Qt-Bibliothek (Aussprache wie engl. *cute* für „süß“, „goldig“) ist sehr mächtig und basiert auf C++. Das Modul läuft auf Windows-, Mac-, Linux-, Android- und iOS-Betriebssystemen und weist im Vergleich zu Tkinter ein moderneres GUI-Design auf. Die aktuelle Version, Qt 5, wurde im Dezember 2012 freigegeben. Aufgrund seiner umfassenden Funktionalität findet Qt in vielen Bereichen Anwendung und kommt beispielsweise in den Linux-Versionen von Spotify und Skype, im VLC Media Player, der Desktop-Version des Instant-Messaging-Dienstes Telegram sowie in Google Earth zum Einsatz.

Obwohl ursprünglich in C++ entwickelt, ist Qt durch unterschiedliche Sprachanbindungen ebenfalls für andere Programmiersprachen erhältlich, darunter C#, Ruby, Java, PHP und Python. Eine der für Python infrage kommenden Qt-Sprachanbindungen ist beispielsweise das PyQt-Framework, das wir im Folgenden nutzen werden. Es besteht wiederum aus mehreren Modulen für unterschiedliche Funktionalitätsbereiche, zu denen Tools zur graphischen Gestaltung, Ereignishandhabung, Multimedia-inhalte, XML- und SQL-Datenbanken und vieles mehr zählen.

Die gesamte Dokumentation des Qt-Frameworks finden Sie unter <https://doc.qt.io/>.

### 15.1 PyQt installieren

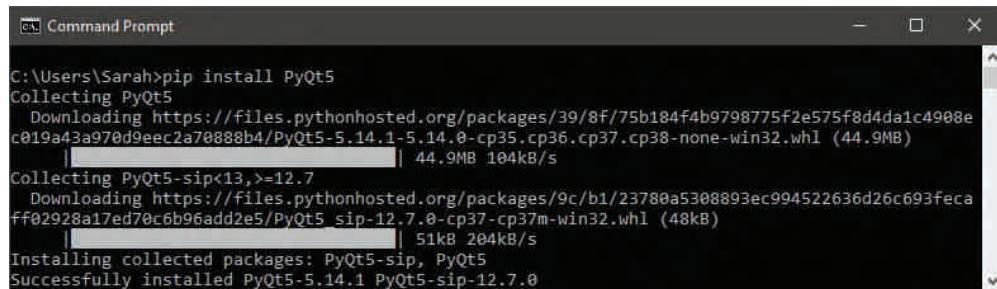
Qt wurde ursprünglich in den 90er Jahren von der norwegischen Firma *Trolltech* entwickelt, gehörte zwischenzeitlich unter anderem *Nokia* und wird seit 2014 von der Firma *Qt Company* vertrieben und weiterentwickelt. Das Qt-Toolkit basiert auf einem sogenannten dualen Lizenzmodell. Für Python bzw. PyQt bedeutet dies insbesondere, dass es sowohl kommerziell – mit einer kostenlosen Testversion – als auch als Open-Source-Variante unter der GPL-Lizenz für freie Software erhältlich ist. Obwohl die Features der letztgenannten Variante eingeschränkt sind, wird diese freie Version für die in diesem Buch vorgestellten Beispiele genügen.

Zur Installation einer GPL-Version von PyQt 5 können Sie ganz einfach in einem Kommandozeilenfenster den Befehl

```
1 pip install PyQt5
```

eingeben. Wie in Kapitel 10 erwähnt, werden nach diesem Schema standardmäßig die meisten Softwarepakete für Python installiert oder – mit `pip uninstall Paketname` – deinstalliert. Mehr über das PyQt5-Paket können Sie unter <https://pypi.org/project/PyQt5/> nachlesen.

## 15.1 PyQt installieren



```
C:\Users\Sarah>pip install PyQt5
Collecting PyQt5
  Downloading https://files.pythonhosted.org/packages/39/8f/75b184f4b9798775f2e575f8d4da1c4908ec019a43a970d9eec2a70888b4/PyQt5-5.14.1-5.14.0-cp35.cp36.cp37.cp38-none-win32.whl (44.9MB)
    |████████| 44.9MB 104kB/s
Collecting PyQt5-sip<13,>=12.7
  Downloading https://files.pythonhosted.org/packages/9c/b1/23780a5308893ec994522636d26c693feca ff02928a17ed70c6b96add2e5/PyQt5_sip-12.7.0-cp37-cp37m-win32.whl (48kB)
    |████████| 51kB 204kB/s
Installing collected packages: PyQt5-sip, PyQt5
Successfully installed PyQt5-5.14.1 PyQt5-sip-12.7.0
```

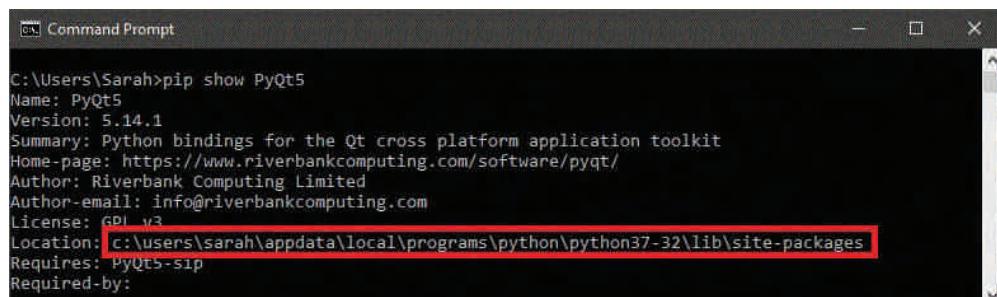
**Abb. 15.1** Die Installation von PyQt 5 aus der Kommandozeile

Sollte die Installation mit Ihrer Python-Version oder für Ihr Betriebssystem nicht funktionieren, finden Sie weitere Informationen und Downloadoptionen für die verschiedenen PyQt-Versionen unter [www.riverbankcomputing.com](http://www.riverbankcomputing.com).

Geben Sie nun im Kommandozeilenfenster den Befehl

```
1 pip show PyQt5
```

ein. Sie sehen ein Fenster mit Informationen zu PyQt 5. Sofern Ihnen der Pfad, unter dem die PyQt-Bibliothek installiert wurde, nicht bekannt ist, können Sie ihn hier unter Location einsehen und kopieren. Anschließend müssen Sie ihn in PyCharm angeben, damit die IDE die externe Bibliothek finden und importieren kann.

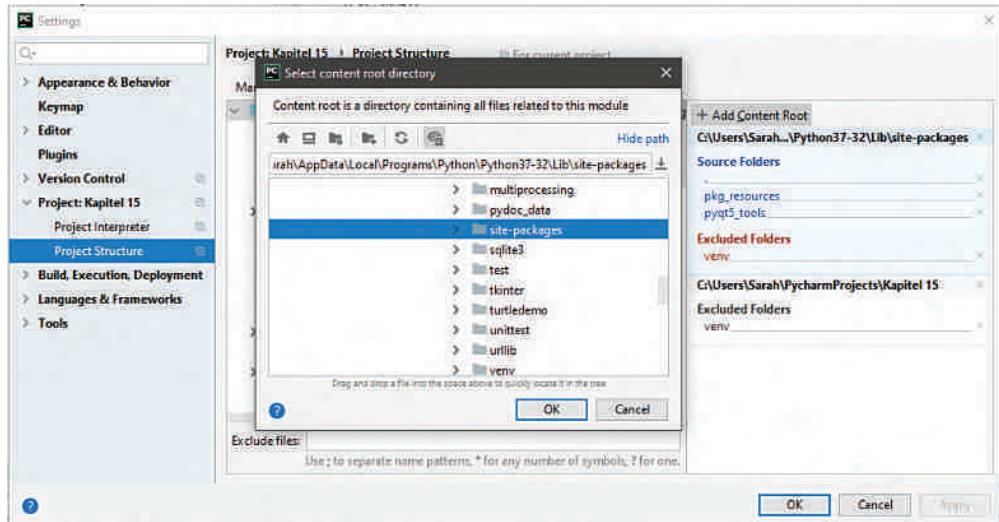


```
C:\Users\Sarah>pip show PyQt5
Name: PyQt5
Version: 5.14.1
Summary: Python bindings for the Qt cross platform application toolkit
Home-page: https://www.riverbankcomputing.com/software/pyqt/
Author: Riverbank Computing Limited
Author-email: info@riverbankcomputing.com
License: GPL v3
Location: c:/users/sarah/appdata/local/programs/python/python37-32/lib/site-packages
Requires: PyQt5-sip
Required-by:
```

**Abb. 15.2** Den Installationspfad von PyQt 5 identifizieren

Hierfür gehen Sie im PyCharm-Menü auf **File → Settings**, woraufhin sich das Einstellungsfenster öffnet. Klicken Sie im linken Navigationsfeld auf Ihr Projekt (**Project: ...**) und darunter auf die Option **Project Structure**. Rechts im selben Fenster sehen Sie die Option **+ Add Content Root**. Klicken Sie darauf, öffnet sich ein neues Fenster, in welches Sie nun den entsprechenden Pfad eingeben können. Bestätigen Sie mit **OK** und klicken Sie abschließend auf **Mark as: Sources**.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen



**Abb. 15.3** Eine externe Bibliothek in PyCharm verwenden

Nun können Sie in PyCharm mit der PyQt-Bibliothek arbeiten.

### 15.2 Erste einfache Fenster erstellen

Jede graphische Benutzeroberfläche besteht aus einem oder mehreren Fenstern, innerhalb derer sich die Widgets zur Steuerung des Programms befinden. Ein einzelnes Fenster einer GUI wird auch *Dialog* genannt. Einige Widgets im Dialogfenster dienen der Gruppierung weiterer Widgets und können wiederum untergeordnete Widgets enthalten, die auch als *Children* (engl. für *Kinder*) bezeichnet werden.

Zu Anfang möchten wir nun untersuchen, wie man mit PyQt ein einfaches Fenster erstellt. Dafür genügt der folgende Code:

#### Codebeispiel #1

```
1 from PyQt5.QtWidgets import QApplication, QLabel
2
3 app = QApplication([])
4 text = QLabel("Hallo Welt!")
5 text.show()
6 app.exec_()
```

In der ersten Codezeile importieren wir aus dem `PyQt5.QtWidgets`-Modul, das verschiedene Komponenten zur Gestaltung von GUIs bereithält, die für unseren Code benötigten Komponenten `QApplication` und `QLabel`. Jedes GUI-Programm muss genau eine Instanz von `QApplication` enthalten, mithilfe derer die Einstellungen

## 15.2 Erste einfache Fenster erstellen

der GUI gehandhabt werden können. Den eckigen Klammern von `QApplication` können Argumente übergeben werden, was in unserem einfachen Fall jedoch noch nicht nötig ist. Mit `QLabel` wird ein Label-Widget angelegt, in dem Text oder Bilder angezeigt werden können. In der vorletzten Zeile wird der an `QLabel` übergebene Text ("Hallo") mit `text.show()` angezeigt. Die letzte Zeile garantiert, dass der Code solange ausgeführt bzw. das Fenster so lange geöffnet bleibt, bis es seitens der Nutzerinnen oder Nutzer geschlossen wird. Wenn Sie den Code ausführen, sehen Sie nun ein kleines Fenster mit der Grußformel „Hallo Welt!“.

Um die Größe und Textposition unseres ersten erstellten Fensters zu ändern, können Sie in der dritten Codezeile eine neue Zeile mit dem folgenden Code einfügen:

### *Codebeispiel #2*

```
1 app.setStyleSheet("QLabel { margin: 50ex; }")
```

Mit `setStyleSheet` lässt sich hier angeben, welchen Abstand der Text zum Fensterrand haben soll. Je nachdem, wie Sie die Zahl hinter `margin` variieren, ändert sich der Abstand.

Es besteht ebenfalls die Möglichkeit, einen Titel für unser Fenster zu wählen. Hierfür fügen Sie ganz einfach vor der drittletzten Zeile den Code

```
1 text.setWindowTitle("Erstes Fenster")
```

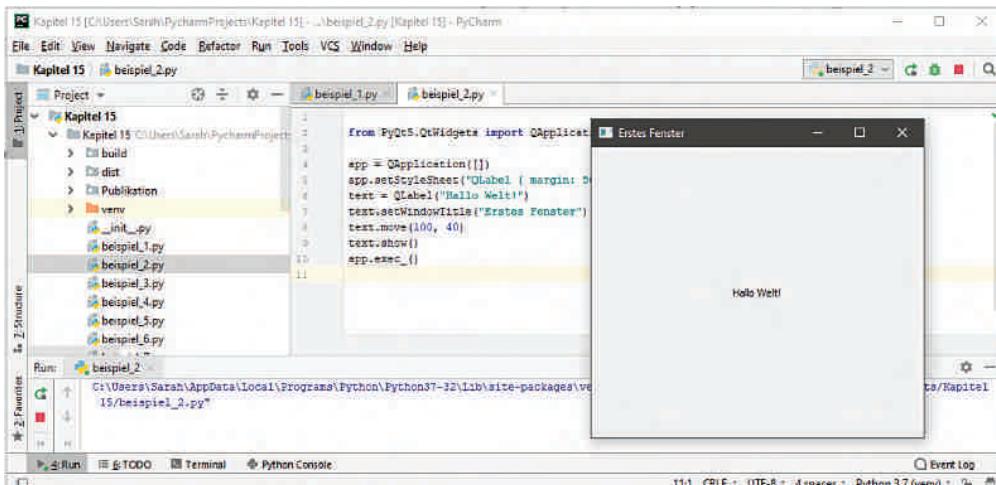
ein. In den Klammern steht dabei der gewünschte Fenstertitel.

Standardmäßig wird sich das Dialogfenster im mittleren Bildschirmbereich öffnen. Dies können Sie jedoch auch ändern, indem Sie zwei Zahlen angeben, die für die horizontale und vertikale Position des Dialogfensters stehen. Fügen Sie hierfür im Anschluss an die vorige Fenstertitel-Zeile folgende Codezeile ein:

```
1 text.move(100, 40)
```

Sie können ausprobieren, wie sich das Fenster verschiebt, wenn Sie die Zahlen variieren.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen



**Abb. 15.4** Die Anzeige unseres ersten Fensters

Obwohl unser erstes Fenster sicherlich einen ersten ehrenhaften Schritt in Richtung GUI-Erstellung darstellt, bietet es noch keine Möglichkeit zur Interaktion. In Wirklichkeit bestehen GUIs nämlich aus vielen verschiedenen Widgets, die innerhalb des Fensters positioniert werden müssen.

Wir werden nun betrachten, wie sich die Anordnung zweier Knöpfe innerhalb eines Fensters angeben lässt:

### Codebeispiel #3

```

1 from PyQt5.QtWidgets import QApplication, QWidget, QPushButton, QHBoxLayout
2
3 app = QApplication([])
4 fenster = QWidget()
5 layout = QHBoxLayout()
6 layout.addWidget(QPushButton("OK"))
7 layout.addWidget(QPushButton("Abbrechen"))
8 fenster.setLayout(layout)
9 fenster.show()
10 app.exec_()

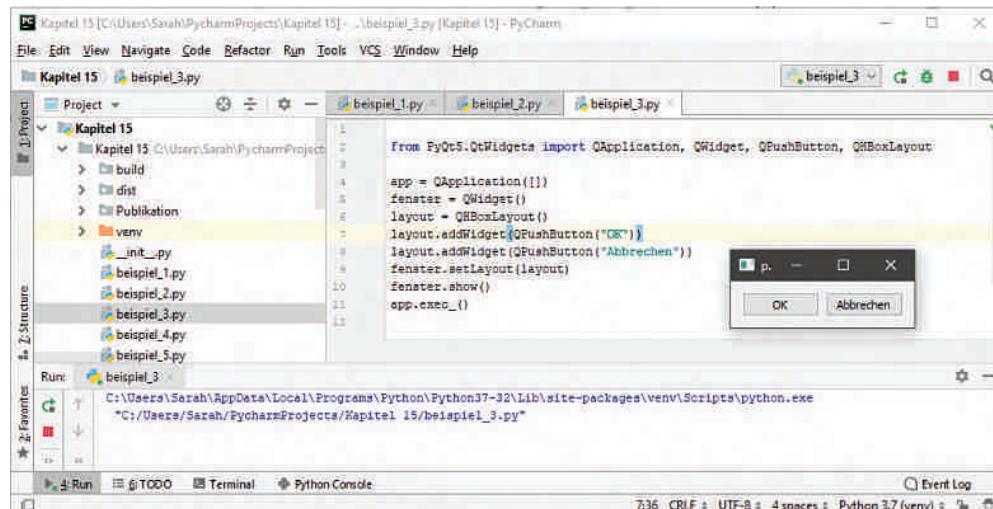
```

Mit dem Modul `QHBoxLayout` werden die beiden Buttons hier horizontal angeordnet. Verwenden Sie beispielsweise `QVBoxLayout`, erscheinen die Buttons in vertikaler Anordnung. Sie können diese Option selbst einmal ausprobieren.

Nach dem obligatorischen Instanziieren von `QApplication` erzeugen wir ein `fenster`, für das wir die einfache `QWidget`-Klasse verwenden. Dem gewünschten `layout` fügen wir sodann zwei `QPushButton`-Knöpfe mit der entsprechenden Be-

## 15.2 Erste einfache Fenster erstellen

schriftung hinzu und geben mit `fenster.setLayout(layout)` an, dass wir dieses Layout für unser Fenster verwenden möchten.



**Abb. 15.5** Ein Fenster mit zwei Buttons erstellen

Die Optik Ihrer GUI lässt sich mithilfe unterschiedlicher *Styles* anpassen. Hierfür können Sie nach `QApplication` als dritte Zeile den Code `app.setStyle()` einfügen. Innerhalb der Klammern muss dabei der gewünschte Style als String stehen. Zu den verfügbaren Qt-Styles gehören – je nach Betriebssystem – Fusion, Windows, WindowsVista und Macintosh. Probieren Sie die Designvariationen selbst einmal aus!

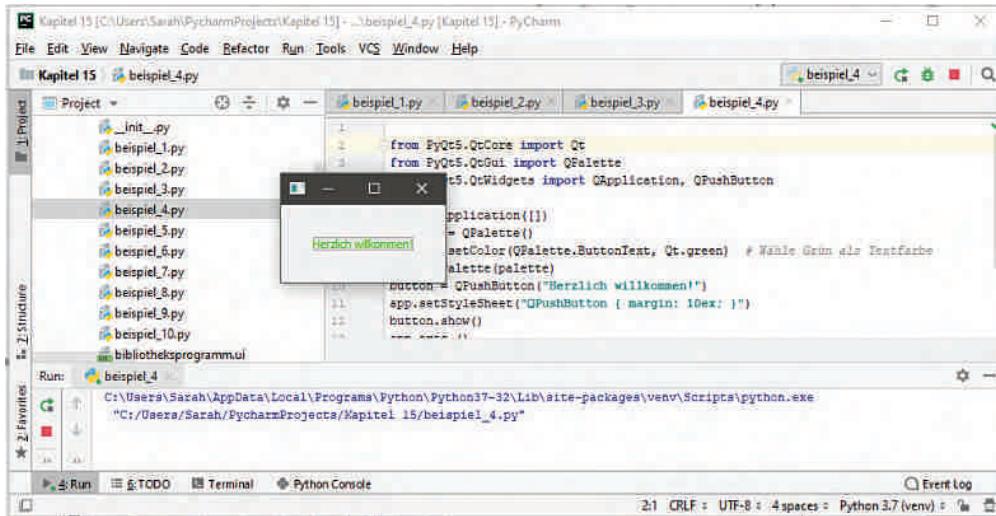
Doch nicht nur das Gesamtdesign, sondern auch einzelne Farben lassen sich spezifizieren. So lässt sich beispielsweise mit `QPalette` als Textfarbe für Buttons Grün anstatt Schwarz auswählen:

### Codebeispiel #4

```

1  from PyQt5.QtCore import Qt
2  from PyQt5.QtGui import QPalette
3  from PyQt5.QtWidgets import QApplication, QPushButton
4
5  app = QApplication([])
6  palette = QPalette()
7  palette.setColor(QPalette.ButtonText, Qt.green)  # Wähle Grün als Textfarbe
8  app.setPalette(palette)
9  button = QPushButton("Herzlich willkommen!")
10 app.setStyleSheet("QPushButton { margin: 10ex; }")
11 button.show()
12 app.exec_()
    
```

## 15 Grafische Benutzeroberflächen mit PyQt erstellen



**Abb. 15.6** Die Textfarbe für einen Button wählen

Beachten Sie hier auch in der drittletzten Zeile wieder die Wahl des Buttonabstands zum Fensterrand.

Nun haben Sie einen ersten Einblick in den Aufbau und die Funktionsweise einer mit PyQt erstellten graphischen Benutzeroberfläche erhalten. Der Code, der sich hinter einem einzigen Fenster mit mehreren Texteingabefeldern, Buttons und weiteren Optionen zur Interaktion verbirgt, kann jedoch schnell sehr komplex und lang werden, was den Rahmen der hier vorstellbaren Gestaltungsmöglichkeiten sprengen würde. Aus diesem Grund möchten wir Ihnen im nächsten Abschnitt ein praktisches Tool vorstellen, mithilfe dessen sich die GUI-Erstellung mit Widgets wesentlich einfacher gestalten lässt: *Qt Designer*.

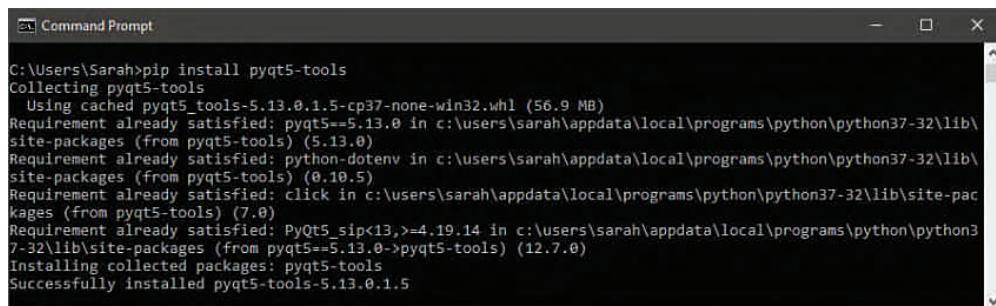
### 15.3 Den Qt Designer verwenden

Das Paket, das den Qt Designer enthält, heißt `pyqt5-tools` und wird separat zu PyQt mit einem zusätzlichen `pip`-Befehl installiert. Angaben zum Paket finden Sie unter <https://pypi.org/project/pyqt5-tools/>.

Geben Sie in einem Kommandozeilenfenster nun Folgendes ein:

```
1 pip install pyqt5-tools
```

### 15.3 Den Qt Designer verwenden

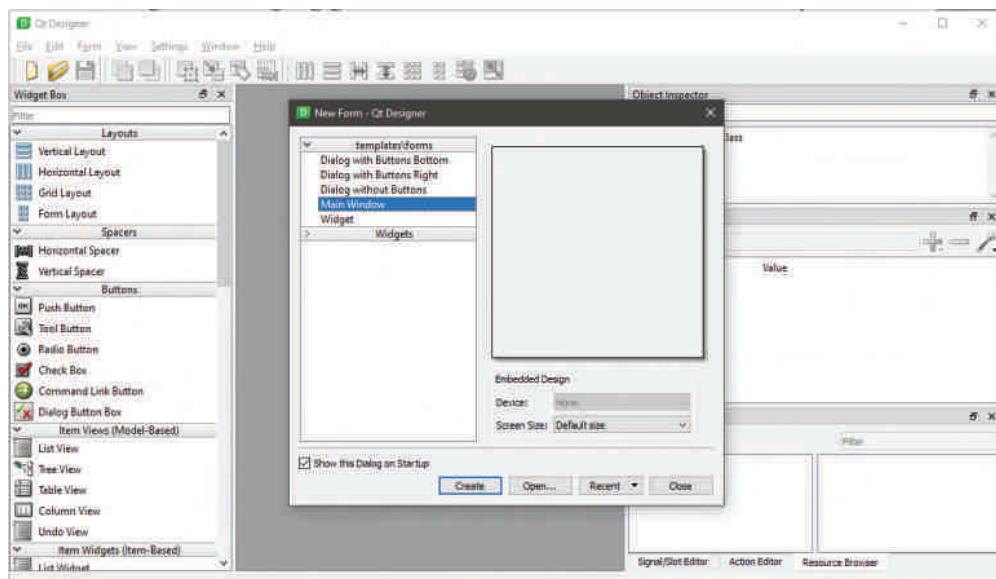


```
C:\Users\Sarah>pip install PyQt5-tools
Collecting PyQt5-tools
  Using cached PyQt5_tools-5.13.0.1.5-cp37-none-win32.whl (56.9 MB)
Requirement already satisfied: PyQt5==5.13.0 in c:\users\sarah\appdata\local\programs\python\python37-32\lib\site-packages (from PyQt5-tools) (5.13.0)
Requirement already satisfied: python-dotenv in c:\users\sarah\appdata\local\programs\python\python37-32\lib\site-packages (from PyQt5-tools) (0.10.5)
Requirement already satisfied: click in c:\users\sarah\appdata\local\programs\python\python37-32\lib\site-packages (from PyQt5-tools) (7.0)
Requirement already satisfied: PyQt5_sip<13,>=4.19.14 in c:\users\sarah\appdata\local\programs\python\python37-32\lib\site-packages (from PyQt5-tools) (12.7.0)
Installing collected packages: PyQt5-tools
Successfully installed PyQt5-tools-5.13.0.1.5
```

**Abb. 15.7** Die Installation von Qt Designer aus der Kommandozeile

Nachdem der Installationsvorgang abgeschlossen ist, können Sie Qt Designer direkt aus demselben Kommandozeilenfenster durch die Eingabe von `designer` starten. Sie finden die `designer.exe`-Datei ebenfalls im Ordner `site-packages`, dessen Pfad in der obigen Abbildung angegeben ist. Am besten suchen Sie innerhalb von `site packages` nach der Datei `designer.exe` und richten eine Verknüpfung zum Desktop oder im Startmenü ein.

Sie sehen nun das Startfenster von Qt Designer mit der Option, ein neues Formular anzulegen.



**Abb. 15.8** Die erste Ansicht des Qt Designers

Wählen Sie hier **Main Window** und klicken Sie auf **Create**. Das Fenster, in dem Sie Ihre Widgets anlegen können, wird in der Mitte des Bildschirms angezeigt. Auf der linken Seite sehen Sie alle Ihnen zur Verfügung stehenden Widget-Optionen. Um diese

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

Widgets zu verwenden genügt es, auf das gewünschte Widget zu klicken und dieses in das neue Fenster zu ziehen. Versuchen Sie dies beispielsweise einmal mit einem **Push Button** im Auswahlmenü **Buttons** oder einem **Label** unter **Display Widgets**, die wir bereits zuvor in unserem PyQt-Code verwendet hatten. Auf diese Weise lassen sich die gewünschten Layout-Vorgänge viel schneller durchführen, als wenn wir sie manuell als Code erstellen. Das fertige Fenster wird jedoch etwas anders aussehen als in der aktuellen Ansicht. Für eine Vorschau des Fensters können Sie entweder **Strg + R** drücken oder unter **Form → Preview...** auswählen. Andere Styles sind unter **Form → Preview in** wählbar.

Verkleinern Sie nun einmal die Größe des Vorschau-Fensters. Wie Sie sehen, werden die gewählten Widgets nicht dem neuen Format angepasst und verschwinden, wenn das Fenster zu klein gewählt ist. Selbstverständlich wäre es besser, wenn sich die Dimensionen zwischen den Widgets beim Anpassen der Fenstergröße ebenfalls relativ zueinander ändern, sodass die Widgets allesamt sichtbar bleiben. Dies lässt sich mithilfe der unterschiedlichen Layoutoptionen bewerkstelligen, die wir bereits im vorigen Abschnitt kennen gelernt haben.

### 15.3.1 Layouts für ein Fenster angeben

Um ein Layout für das gesamte Fenster anzugeben, das für alle darin enthaltenen Widgets gilt, müssen Sie einen rechten Mausklick auf einen widget-freien Bereich im Fenster vornehmen. Im anschließend geöffneten Fenster wählen Sie **Lay out** und die gewünschte Layoutoption, beispielsweise horizontale, vertikale oder Rasteranordnung. Wenn Sie nun in den Vorschaumodus des Fensters wechseln, werden Sie feststellen, dass sich die Widgets bei Größenänderungen allesamt dem Fensterformat anpassen. Experimentieren Sie einmal selbst, wie sich das Layout mehrerer im Fenster positionierter Widgets mit den unterschiedlichen Layoutoptionen ändert. Mit **Break Layout** können Sie Ihre Wahl anschließend wieder rückgängig machen.

Nachdem Sie per Rechtsklick ein allgemeines Layout – beispielsweise ein Rasterlayout (**Grid Layout**) – für Ihr Fenster gewählt haben, können Sie anschließend innerhalb dieses Layouts weitere Layoutarten vorgeben, die jeweils nur für einen bestimmten Bereich innerhalb des Fensters gelten. Hierzu wählen Sie das gewünschte Layout per Drag-and-drop links aus dem Widgetmenü **Widget Box**, in dem sich ebenfalls ein Menüpunkt **Layout** mit den entsprechenden Auswahlmöglichkeiten befindet. Eine weitere Möglichkeit, das Layout anzugeben, ist die Layout-Toolbar im oberen Bildschirmbereich unter dem Hauptmenü:



Die Layout-Toolbar zur Auswahl des Layouts

### 15.3 Den Qt Designer verwenden

So lassen sich anschließend beispielsweise innerhalb eines Rasters drei *Push Buttons* oder mehrere *Check Boxes* in vertikaler Anordnung einfügen. Umgekehrt können Sie auch zuerst mehrere Widgets innerhalb des allgemeinen Layouts platzieren und diese durch **Strg** oder durch die Auswahl des Bereichs mit der Maus gleichzeitig markieren. Wählen Sie nun im Menü links oder aus der Layout-Toolbar ein Layout aus, werden die Widgets sogleich diesem Layout entsprechend angeordnet. Nachdem Sie Ihr Fenster mit allen Widgets und den passenden Layouts erstellt haben, ist es eine gute Idee, abschließend auf **Adjust Size** ganz rechts in der Layout-Toolbar zu klicken. Somit werden automatisch die optimalen Größenverhältnisse für das Fenster gewählt.

Haben Sie an dieser Stelle keine Eile und verbringen Sie ruhig etwas Zeit damit, sich mit den verschiedenen Widget- und Layoutoptionen vertraut zu machen, bis Sie sich in ihrer Verwendung sicher fühlen.

Im folgenden Bild sehen Sie ein Beispiel eines Vorschau-Fensters, in dessen Hauptlayout als Raster zusätzlich sowohl ein vertikales als auch ein horizontales Unterlayout verwendet werden:

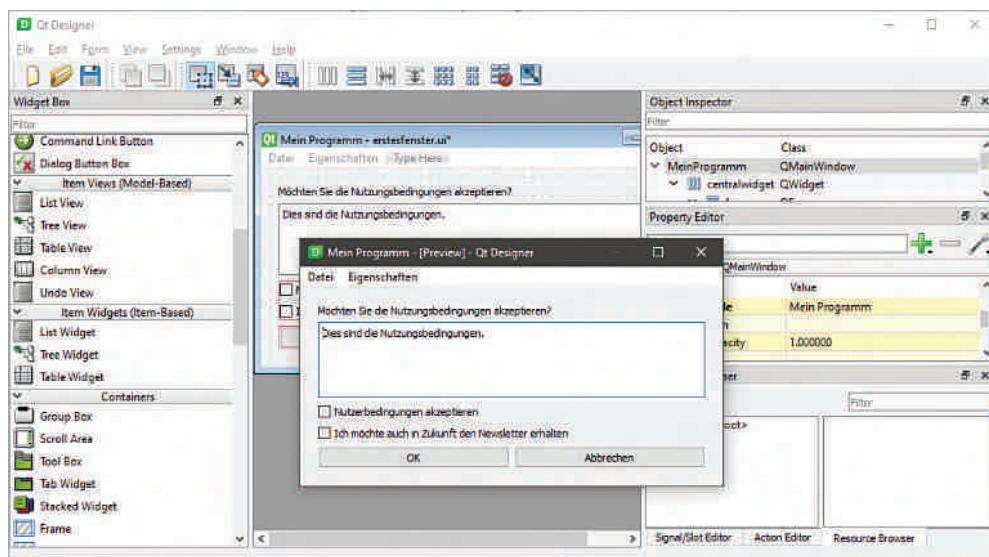


Abb. 15.9 Ein Beispelfenster mit Preview

Horizontale oder vertikale **Spacers** gehören ebenfalls zu den Layout-Management-Optionen. Sie dienen dazu, andere Widgets zentral (mit je einem Spacer auf jeder Seite) oder auf einer gewünschten Seite des Fensters (mit nur einem Spacer) zu positionieren. Sie funktionieren dabei wie Federn, die immer genauso viel Platz einnehmen, wie sie können, ohne dass sich dabei die ursprüngliche Größe der Widgets ändert.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

### 15.3.2 Eigenschaften der Fenster und Widgets anpassen

Auf der rechten Seite des Qt Designers sehen Sie die drei Fenster **Object Inspector**, **Property Editor** und **Resource Browser**, von denen vor allem die ersten beiden sehr nützlich sind.

Der **Object Inspector** zeigt die hierarchische Struktur des gesamten Fensters an: Oben steht zunächst das allgemeine Layout, danach folgen alle darin enthaltenen Widgets samt deren Klassen und eventuelle weitere Layouts.

Die Standard-Beschriftungen der Widgets lassen sich übrigens ganz einfach ändern, indem Sie einen Doppelklick darauf machen und den gewünschten Text hineinschreiben. Klicken Sie ein bestimmtes Widget an, werden dessen Eigenschaften zudem im **Property Editor** sichtbar. Unter **objectName** sehen Sie beispielsweise die ursprüngliche Bezeichnung des Widgets. Diese ändert sich nicht, auch wenn Sie dessen Beschriftung bereits geändert haben. Für unseren späteren Python-Code ist es jedoch ratsam, die **objectName**-Bezeichnungen unseres selbst gewählten Beschriftungen entsprechend anzupassen, damit wir sie leichter aufrufen können. Sie können die **objectName**-Namen entweder durch einen Doppelklick ändern oder indem Sie einen Rechtsklick auf das entsprechende Widget vornehmen und dann **Change objectName...** auswählen.

Übrigens lässt sich auch der Titel des gesamten Fensters angeben. Hierfür klicken Sie im **Object Inspector** auf **MainWindow** und tippen den gewünschten Namen anschließend im **Property Editor** unter **windowTitle** ein.

Unterhalb des Fenstertitels können Sie selbst eine Menüleiste definieren. Mit einem Doppelklick auf **Type Here** können Sie dort eigene Menüoptionen eintragen. Darunter lassen sich weitere Optionen sowie eine Trennlinie (**Add Separator**) angeben. Haben Sie eine Menü-Registerkarte erstellt, können Sie rechts davon weitere Registerkarten anlegen.

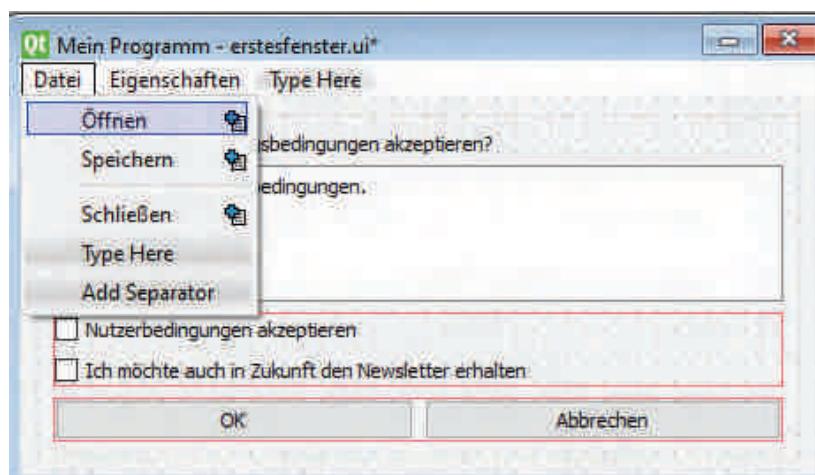


Abb. 15.10 Die Menüleiste anpassen

### 15.3 Den Qt Designer verwenden

Im Texteingabefeld unter **Property Editor** besteht die Möglichkeit, nach bestimmten Begriffen für Widget-Eigenschaften zu suchen. Für Widgets, die Text enthalten – etwa ein *Label* oder einen *Push Button* – können Sie dort nach der Eigenschaft *text* suchen. Klicken Sie auf dessen Wert (*Value*), so erscheint rechts daneben eine Toolbox mit drei Punkten. Klicken Sie wiederum darauf, öffnet sich ein neues Fenster, in dem Sie verschiedene Schriftoptionen wie Größe, Ausrichtung und Farbe anpassen können.

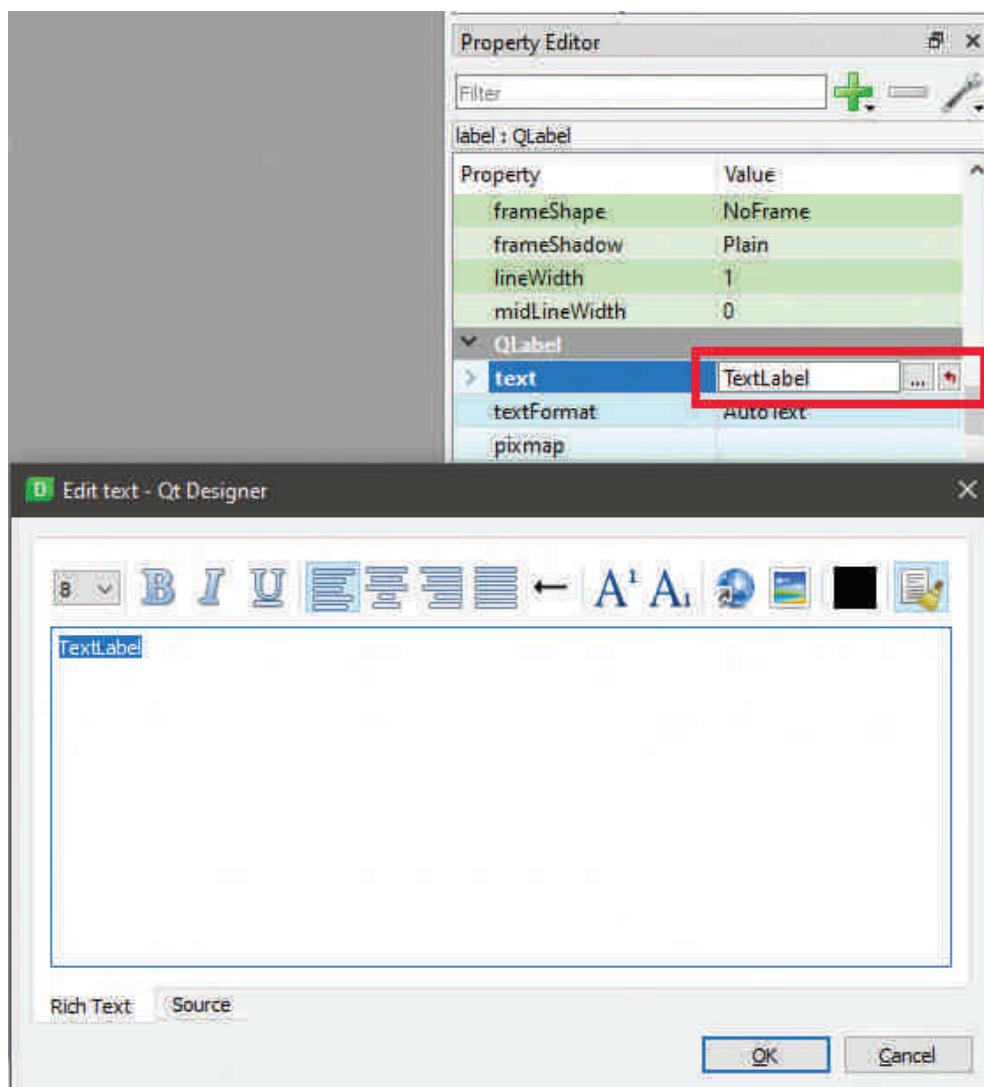


Abb. 15.11 Schriftoptionen aus dem Property Editor ändern

Geben Sie im Texteingabefeld des **Property Editor** als Suchbegriff *palette* ein, so wird unter **QWidget** die Eigenschaft **palette** mit der Option **Change Palette** angezeigt. Durch einen Klick auf diese Option können Sie die gesamte Optik aller Widgets – beispielsweise die Farbgebung der Buttons – anpassen, wie wir dies im letzten Codebei-

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

spiel in Abschnitt 15.2. bereits getan hatten. Um die Farbwahl für einen Button zu ändern, klicken Sie auf **ButtonText** und wählen Sie die gewünschte Farbe aus. Wenn Sie jetzt in einem Fenster einen *Push Button* als Widget hinzufügen, wird dessen Textfarbe automatisch in der gewünschten Farbe angezeigt. Im **Property Editor** lassen sich unter der Eigenschaft **geometry** ebenfalls die Größenangaben des Buttons anpassen. Auch gibt es die Möglichkeit, einen Button als Standardschaltfläche eines Fensters optisch hervorzuheben. Geben Sie hierzu im **Property Editor** als Suchbegriff *default* ein und setzen Sie unter *Value* ein Häkchen.

Auch an dieser Stelle sollten Sie sich am besten selbst mit den verschiedenen Eigenschaften vertraut machen, um einen Überblick über die Fülle an Auswahlmöglichkeiten für Widgets zu bekommen.

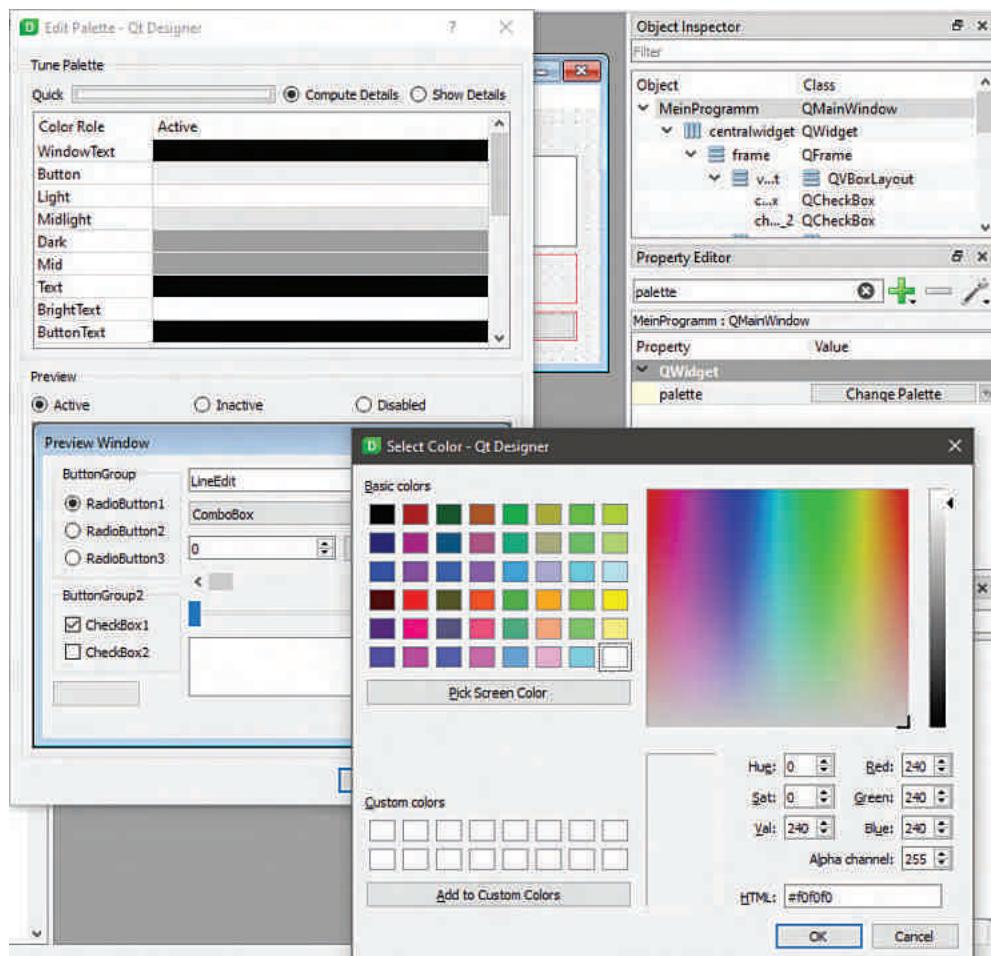


Abb. 15.12 Die Farbpalette im Property Editor auswählen

## 15.4 In Qt Designer erstellte Fenster in einem PyQt-Programm verwenden

### 15.4 In Qt Designer erstellte Fenster in einem PyQt-Programm verwenden

Glückwunsch! Nun haben Sie Ihr erstes brauchbares Fenster mit Qt Designer erstellt. Um dieses Fenster später im Python-Code für ein Programm verwenden zu können, müssen Sie es unter **File → Save As...** in einer *.ui*-Datei abspeichern. Das Kürzel *ui* steht für *user interface*. Als Speicherort sollten Sie denjenigen Ordner wählen, in dem Sie Ihre *.py*-Programmdatei erstellen werden.

Bei einer mittels QT Designer erstellten *.ui*-Datei handelt es sich um eine Datei im XML-Format, die sich zwar mit einem Texteditor oder als Fenster im Qt Designer öffnen, sich jedoch nicht ohne Weiteres als Python-Code implementieren lässt. Um dies zu bewerkstelligen, sind weitere Schritte nötig.

Angenommen, wir haben unser erstes Fenster unter dem Dateinamen *einfachesfenster.ui* in einem PyCharm-Ordner abgespeichert, in dem wir nun auch unseren Code mit PyQt erstellen möchten. Der folgende Code zeigt die einfachste Möglichkeit, unsere *.ui*-Datei in Python zu verwenden:

#### Codebeispiel #5

```

1 import sys
2 from PyQt5 import QtWidgets, uic
3
4 class EinfachesFenster(QtWidgets.QMainWindow):
5     def __init__(self):
6         QtWidgets.QMainWindow.__init__(self) # Alternative: super()
7         self.ui = uic.loadUi("einfachesfenster.ui", self)
8
9
10 app = QtWidgets.QApplication(sys.argv)
11 fenster = EinfachesFenster()
12 fenster.show()
13 sys.exit(app.exec_())

```

Das *sys*-Modul wird hierbei benötigt, um auf Argumente zugreifen zu können. Das in der PyQt5-Bibliothek enthaltene *uic*-Modul dient zum Laden von *.ui*-Dateien in Python.

Unsere selbst definierte Klasse *EinfachesFenster* erbt von der übergeordneten Klasse *QtWidgets.QMainWindow*. Achten Sie darauf, dass Sie die richtige Oberklasse angeben; da wir beim ursprünglichen Anlegen der Datei unter **New Form** die Option **Main Window** wählten, heißt unsere Klasse hier *QMainWindow*. Wenn Sie sich anfangs etwa für die Optionen **Dialog** oder **Widget** entschieden hatten, müssen Sie *QMainWindow* mit den Begriffen *QDialog* bzw. *QWidget* für die gewünschte Klasse ersetzen.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

Wenn Sie nicht den gesamten Namen der Oberklasse ausschreiben möchten, können Sie hier genauso gut anstatt mit der Oberklasse `QtWidgets.QMainWindow`, von der die Unterklasse `EinfachesFenster` erbt, mit dem Begriff `super()` arbeiten (vgl. Kapitel 11.6.). Die entsprechende Zeile besitzt dann die etwas abgeänderte Form `super(EinfachesFenster, self).__init__()`.

Die Methode `loadUI()` innerhalb der Klassendefinition lädt den angegebenen Dateinamen und erzeugt dabei ein PyQt-Objekt für unseren Dialog. Jedes erstellte Widget ist darin als Instanz einer entsprechenden PyQt5-Klasse mit seinem von uns unter **objectName** zuvor gewählten Namen enthalten und lässt sich über diesen abrufen.

Mit `app = QtWidgets.QApplication(sys.argv)` wird anschließend eine Instanz von `QtWidgets.QApplication` erzeugt. `fenster` ist eine Instanz unserer neu angelegten Klasse `EinfachesFenster`. Die Methode `show` sorgt dafür, dass unser Fenster angezeigt wird. Mit `app.exec_()` wird das Programm gestartet.

Sofern es in dem PyCharm-Ordner, der Ihre `.py`-Codedatei enthält, eine Datei mit dem in `loadUi()` angegebenen Namen gibt, sollten Sie beim Starten des Programms das entsprechende GUI-Fenster sehen:

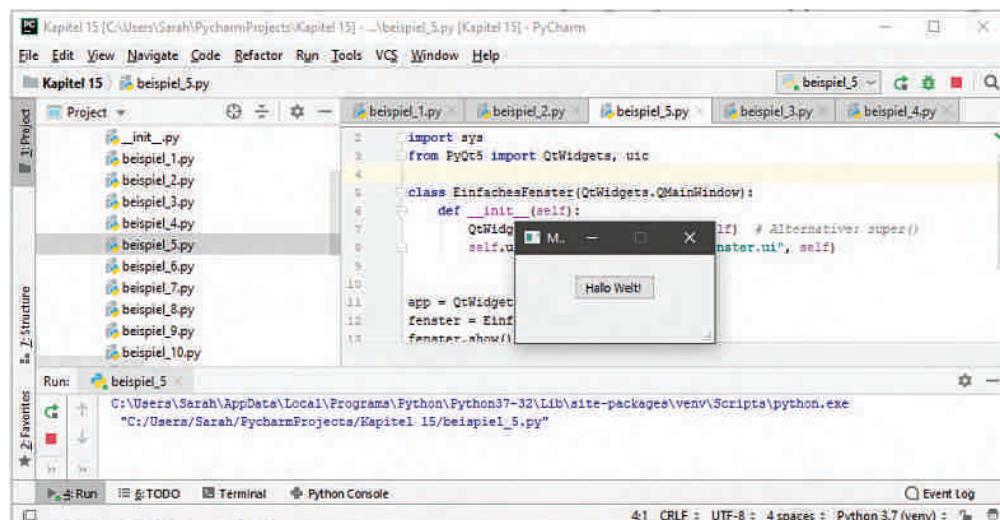


Abb. 15.13 Die GUI aus PyCharm starten

Es gibt eine weitere Möglichkeit, wie Sie eine `.ui`-Datei als Python-Code verwenden können. Dabei wird das gesamte Layout mit allen Widgetdefinitionen in Python-Code ausgedrückt, wie wir dies in Abschnitt 15.2. vorgenommen hatten. Navigieren Sie hierfür in einem Kommandozeilenfenster mit `cd` (engl. für *change directory*) in den entsprechenden Ordner, in dem Sie Ihre `.ui`-Datei abgespeichert haben, beispielsweise

```
1 cd C:\Users\Sarah\PycharmProjects\Programme
```

## 15.4 In Qt Designer erstellte Fenster in einem PyQt-Programm verwenden

Anschließend können Sie die `.ui`-Datei nach dem folgenden Schema in eine `.py`-Datei umwandeln, wobei Sie anstelle von `einfachesfenster` den jeweiligen Dateinamen eingeben:

```
1 pyuic5 -o einfachesfenster.py einfachesfenster.ui
```

Nun wird im ausgewählten Ordner eine `.py`-Datei angelegt, die den kompletten Code für das erstellte Fenster enthält. Diese Verwendungsoption eines mit Qt Designer erzeugten Fensters führt in den meisten Fällen jedoch zu wesentlich längerem Code als die erste hier vorgestellte und aufgrund ihrer Einfachheit bevorzugte Variante.

### 15.4.1 Signale und Slots

Sie kennen nun bereits verschiedene Gestaltungsoptionen mit Widgets, die eine interaktive Nutzung der GUIs ermöglichen. Ebenso haben Sie im vorigen Abschnitt gesehen, wie sich GUIs, die Sie mithilfe von Qt Designer erstellt haben, aus einem Programm heraus aufrufen lassen. Unsere bisherigen GUIs stellten jedoch lediglich die Designstruktur bereit, die wir unserer GUI verleihen wollten und boten keinerlei Möglichkeit zur Interaktion. Klicken Sie beispielsweise im oben angezeigten Dialogfenster auf den Push Button *Hallo Welt!*, führt das Programm keine weiteren Vorgänge aus. Dies ist natürlich nicht im Sinne einer realen GUI, die auf Nutzeraktionen reagieren und bestimmte Informationen für den weiteren Programmverlauf verarbeiten oder speichern können soll.

Um aus einer allgemeinen Designstruktur Programme für den realen Gebrauch zu erstellen, kommen sogenannte *Signale und Slots* zum Einsatz. Damit lässt sich vorgeben, wie das Programm auf eine Aktion seitens der Nutzerinnen und Nutzer reagiert – beispielsweise, dass auf das Drücken eines Buttons hin das aktuelle Fenster geschlossen oder ein weiteres Fenster geöffnet werden soll. In diesem Fall wird durch den Klick auf den Button ein Signal ausgesendet, das anschließend von einem Slot (engl. für *Steckplatz*) empfangen wird, der wiederum eine bestimmte Funktion veranlasst. Dabei ist es ebenfalls möglich, dass mehrere Signale auf ein und denselben Slot verweisen.

Bei Signalen handelt es sich um nichts anderes als Attribute einer Widget-Instanz. Je nach Widget-Typ sind unterschiedliche Signale möglich. Die wichtigsten Widget-Klassen samt den hierfür verfügbaren Signalen und Methoden werden wir Ihnen in Abschnitt 15.5. vorstellen. Für die Widget-Klasse `QPushButton` existiert als Signal beispielsweise das `clicked`-Signal und als Klassenmethode die Option `setText()` zur Beschriftung des Buttons.

15

Wenn Sie mit Signalen und Slots arbeiten möchten, sollten Sie – wie in 15.3.2. erwähnt – im Qt Designer allen Widgets der GUI zuvor individuelle Objektnamen verliehen

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

haben, durch die sie sich anschließend im Code leicht adressieren lassen. Im nächsten Codebeispiel, das eine Erweiterung unseres vorigen Codebeispiels mit Signalen und Slots darstellt, hat unser Push Button etwa die Bezeichnung *hallowelt* erhalten:

### Codebeispiel #6

```

1 import sys
2 from PyQt5 import QtWidgets, uic
3
4 class EinfachesFenster(QtWidgets.QMainWindow):
5     def __init__(self):
6         QtWidgets.QMainWindow.__init__(self) # Alternative: super()
7         self.ui = uic.loadUi("einfachesfenster.ui", self)
8         self.ui.hallowelt.clicked.connect(self.halloweltklick) # Signal &
9             Slot
10
11     def halloweltklick(self):
12         print("Hallo!")
13         self.close()
14
15
16 app = QtWidgets.QApplication(sys.argv)
17 fenster = EinfachesFenster()
18 fenster.show()
19 sys.exit(app.exec_())

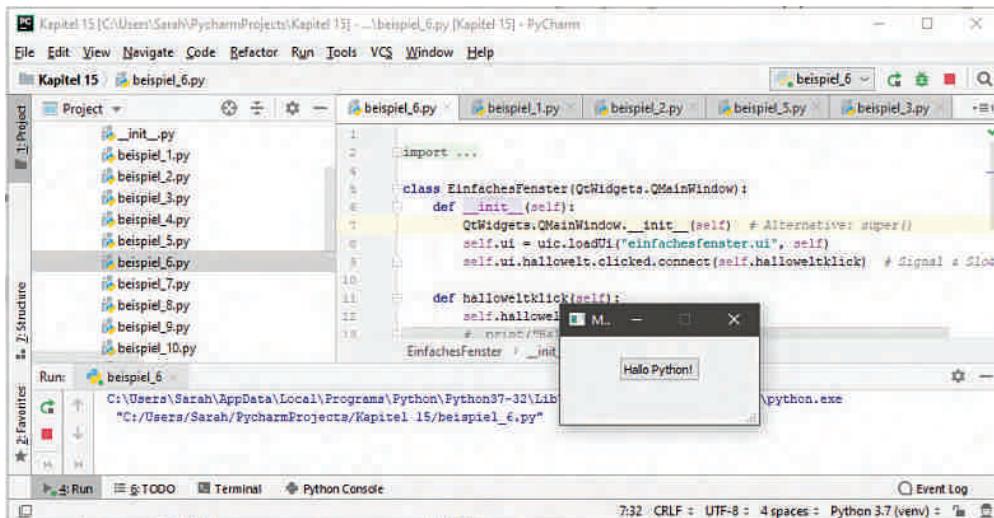
```

In diesem Codebeispiel haben wir innerhalb der Konstruktormethode der Klassendefinition für unseren Push Button *hallowelt* ein *clicked*-Signal eingerichtet, dessen *connect*-Methode es wiederum mit einem Slot verbindet, der als Parameter mit der Funktion *halloweltklick* übergeben wird.

Diese Instanzmethode *halloweltklick* wird anschließend innerhalb der Klassendefinition definiert. Sie sorgt dafür, dass als Output in der IDE der String "Hallo!" ausgegeben wird, woraufhin sich das Dialogfenster schließt. Sie können anstelle der beiden Zeilen in der Methodendefinition auch andere Prozesse ausführen lassen. Probieren Sie einmal aus, was geschieht, wenn Sie in der Methodendefinition für *halloweltklick* folgenden Befehl mit der Klassenmethode *setText()* verwenden:

```
1 self.hallowelt.setText("Hallo Python!")
```

## 15.4 In Qt Designer erstellte Fenster in einem PyQt-Programm verwenden



**Abb. 15.14** Die Beschriftung des Buttons ändert sich beim Klicken darauf

Wenn Sie nun auf den `hallowelt`-Button klicken, ändert sich dessen Beschriftung zu *Hallo Python!*. Nach diesem Schema lassen sich also ganz unterschiedliche Aktionen auf das Anklicken eines Buttons hin ausführen.

Versuchen wir nun, zwei Fenster dahingehend miteinander zu verbinden, dass nach dem Klicken auf den *Hello-Welt*-Knopf des ersten Fensters das zweite Fenster mit einem *Hello-Python*-Label geöffnet wird. Hierfür müssen Sie zunächst das zweite Fenster im Qt Designer erstellen und in einer eigenen .ui-Datei abspeichern, hier *hallopython.ui*. Der Code, der die beiden Fenster durch ein `clicked`-Signal verknüpft, hat die folgende Form:

## *Codebeispiel #7*

```
1 import sys
2 from PyQt5 import QtWidgets, uic
3
4 class ErstesFenster(QtWidgets.QMainWindow):
5     def __init__(self):
6         QtWidgets.QMainWindow.__init__(self) # Alternative: super()
7         self.ui = uic.loadUi("einfachesfenster.ui", self)
8         self.ui.hallowelt.clicked.connect(self.halloweltklick) # Signal &
9             Slot
10
11     def halloweltklick(self):
12         fenster_zwei = ZweitesFenster()
13         fenster_zwei.show()
14
15
16 class ZweitesFenster(QtWidgets.QMainWindow):
17     def __init__(self):
```

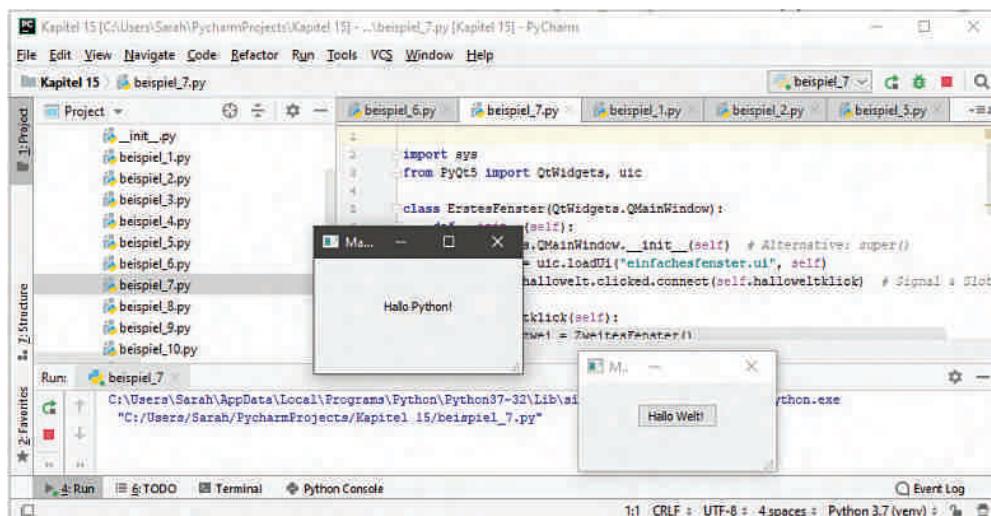
## 15 Grafische Benutzeroberflächen mit PyQt erstellen

```

18         QtWidgets.QMainWindow.__init__(self) # Alternative: super()
19         self.gui = uic.loadUi("hallopython.ui", self)
20
21
22     app = QtWidgets.QApplication(sys.argv)
23     fenster_eins = ErstesFenster()
24     fenster_eins.show()
25     sys.exit(app.exec_())

```

Die Logik sollte ersichtlich sein: Für jede Fenster-Datei legen wir wie gewohnt eine entsprechende Klasse an. Standardmäßig wird im Code zunächst das erste Fenster, `fenster_eins`, geöffnet. Dessen Klassendefinition enthält zudem das `clicked`-Signal, das ausgeführt wird, wenn die Nutzerinnen und Nutzer auf den `hallowelt`-Button drücken. In diesem Fall wird die `halloweltklick`-Methode der Klasse `ErstesFenster()` aufgerufen, die das zweite Fenster, `fenster_zwei`, öffnet.



**Abb. 15.15** Das zweite Fenster öffnet sich, nachdem auf den Button des ersten Fensters geklickt wurde

Nun haben Sie gelernt, wie Sie eine `.ui`-Datei innerhalb von Python-Code verwenden und mithilfe von Signalen und Slots sinnvoll zu einem interaktiven Programm erweitern können.

Ehe wir Ihnen einen kurzen Überblick über die wichtigsten Widget-Klassen samt deren verfügbaren Methoden und Signalen geben, wollen wir Ihnen im nächsten Unterkapitel vorstellen, wie Sie aus der `.ui`-Datei eine `.exe`-Datei zur Verwendung der GUI auf mehreren Rechnern erstellen können.

## 15.4 In Qt Designer erstellte Fenster in einem PyQt-Programm verwenden

### 15.4.2 Eine GUI aus einer .exe-Datei starten

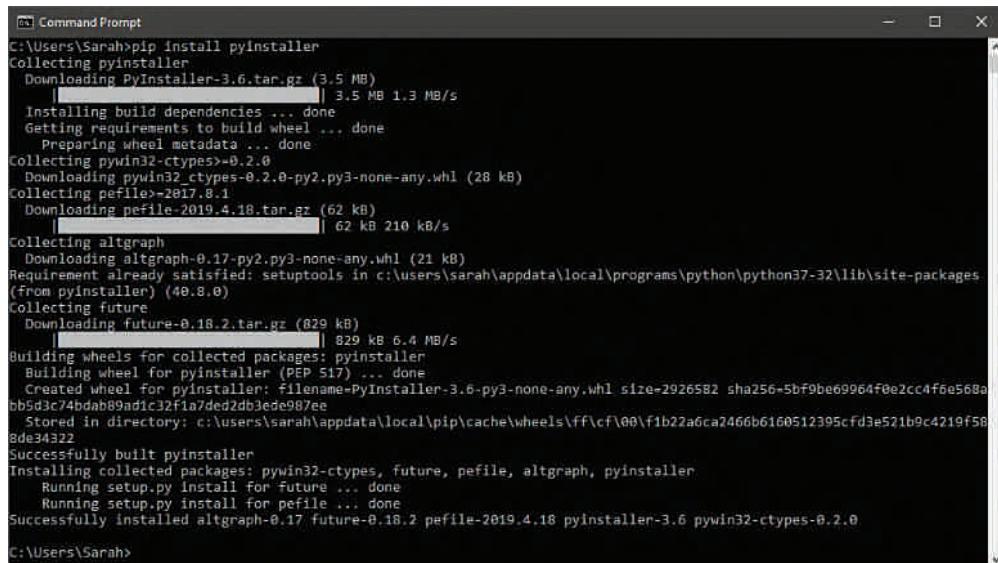
Wenn Sie eine fertige graphische Benutzeroberfläche erstellt und in der IDE die nötigen Signale und Slots hinzugefügt haben, ist die GUI im Grunde fertig für den allgemeinen Gebrauch und kann von weiteren Personen auf mehreren Computern gestartet werden. Da womöglich nicht alle Anwendenden mit der Funktionsweise einer IDE vertraut sind oder keine entsprechende Software auf ihrem Rechner installiert haben, ist es von Vorteil, Ihr neues Programm in der unter Windows üblichen Form einer .exe-Datei unter die Leute zu bringen.

Hierfür können Sie das zusätzliche PyPI-Paket PyInstaller verwenden, das unter anderem Bibliotheken wie PyQt, NumPy, Django oder Matplotlib unterstützt. Die Webseite der Software finden Sie unter <http://www.pyinstaller.org/> und eine nähere Beschreibung des PyPI-Pakets PyInstaller mit weiteren nützlichen Links unter <https://pypi.org/project/PyInstaller/>.

Analog zur Installation von Qt Designer wird das externe Paket über die Kommandozeile installiert. Geben Sie hierfür den Befehl

```
1 pip install pyinstaller
```

in das Kommandozeilenfenster ein.



```
C:\Users\Sarah>pip install pyinstaller
Collecting pyinstaller
  Downloading PyInstaller-3.6.tar.gz (3.5 MB)
    |██████████| 3.5 MB 1.3 MB/s
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing wheel metadata ... done
Collecting pywin32-ctypes>=0.2.0
  Downloading pywin32_ctypes-0.2.0-py2.py3-none-any.whl (28 kB)
Collecting pefile>=2017.8.1
  Downloading pefile-2019.4.18.tar.gz (62 kB)
    |██████████| 62 kB 210 kB/s
Collecting altgraph
  Downloading altgraph-0.17-py2.py3-none-any.whl (21 kB)
Requirement already satisfied: setuptools in c:\users\sarah\appdata\local\programs\python\python37-32\lib\site-packages (from pyinstaller) (40.8.0)
Collecting future
  Downloading future-0.18.2.tar.gz (829 kB)
    |██████████| 829 kB 6.4 MB/s
Building wheels for collected packages: pyinstaller
  Building wheel for pyinstaller (PEP 517) ... done
  Created wheel for pyinstaller: filename=PyInstaller-3.6-py3-none-any.whl size=2926582 sha256=5bf9be69964f0e2cc4f6e568ab5d3c74bdab9adfc32f1a7ded2db3ede987ee
  Stored in directory: c:\users\sarah\appdata\local\pip\cache\wheels\ff\cf\f0\0\f1b22a6ca2466b6160512395cf3e521b9c4219f588de34322
Successfully built pyinstaller
Installing collected packages: pywin32-ctypes, future, pefile, altgraph, pyinstaller
  Running setup.py install for future ... done
  Running setup.py install for pefile ... done
Successfully installed altgraph-0.17 future-0.18.2 pefile-2019.4.18 pyinstaller-3.6 pywin32-ctypes-0.2.0
C:\Users\Sarah>
```

**Abb. 15.16** Installation von PyInstaller aus der Kommandozeile

Nachdem die Installation abgeschlossen ist, müssen Sie in der Kommandozeile den Ordner angeben, der die .py-Datei enthält, aus der Sie die .exe-Datei erstellen wollen.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

Für unser Beispiel haben wir im Pfad `C:\Users\Sarah\PycharmProjects\Programme` einen neuen *Programme*-Ordner angelegt, der die Datei `hallowelt.py` enthält.

Navigieren Sie nun in der Kommandozeile zu diesem Ordner:

```
1 cd C:\Users\Sarah\PycharmProjects\Programme
```

Um die Konvertierung der `.py`- in eine einzige `.exe`-Datei vorzunehmen, genügt der Befehl

```
1 pyinstaller --onefile -w hallowelt.py
```

Im *Programme*-Ordner wurden nun mehrere neue Dateien und Ordner angelegt. Die gewünschte Datei `hallowelt.exe` befindet sich im Ordner `dist`. Durch einen Doppelklick darauf wird die GUI direkt angezeigt. Diese Datei können Sie nun ebenfalls an andere Nutzer weitergeben.

**Wichtig:** Damit die Konvertierung funktioniert und PyInstaller die im Qt Designer erstellte `.ui`-Datei finden kann, muss in der `loadUi`-Methode im `hallowelt.py`-Code (siehe Abschnitt 15.4.) der komplette Dateipfad in der Klammer mit einem vorangestellten `r` angegeben werden. Eine korrekte Schreibweise ist hier also beispielsweise

```
1 self.ui = uic.loadUi(r"C:\Users\Sarah\PycharmProjects\Kapitel 15\hallowelt.
2 ui", self)
```

In diesem Abschnitt haben wir Ihnen die einfachste Variante vorgestellt, wie sich eine `.exe`-Datei erzeugen lässt. Enthält Ihr Python-Script Verweise auf externe Module oder Bilddateien, oder möchten Sie eine brauchbare Datei für ein anderes Betriebssystem erstellen, müssen Sie den Vorgang eventuell durch zusätzliche Maßnahmen und Befehle anpassen. Informationen und Anweisungen hierzu finden Sie im Internet, beispielsweise auf der Webseite von PyInstaller.

### 15.5 Widgetoptionen: Klassen, Methoden und Signale

In der **Widget Box** auf der linken Fensterseite des Qt Designers sehen Sie die verschiedenen Widgets, die Ihnen zur Gestaltung Ihrer GUI zur Verfügung stehen. Mit den unterschiedlichen Optionen unter **Layouts** und **Spacers** sollten Sie bereits vertraut sein. Im Folgenden werden wir Ihnen weitere wichtige Widgets samt deren verfügbaren Signalen und Klassenmethoden vorstellen.

Beachten Sie dabei, dass Sie Methoden, die eine Ausgabe in PyCharm veranlassen sollen, in Verbindung mit einem `print`-Befehl verwenden müssen.

## 15.5 Widgetoptionen: Klassen, Methoden und Signale

### 15.5.1 Buttons

Unter **Buttons** finden Sie unter anderem die Auswahlmöglichkeiten **Push Button**, **Radio Button** und **Check Box**.

Die Einsatzmöglichkeiten von **Push Buttons**, der wichtigsten Schaltfläche einer GUI, haben Sie bereits im ersten Beispiel des vorigen Teilkapitels 15.4.1. kennen gelernt. Push Buttons gehören zur Klasse `QPushButton` und sind häufig mit Begriffen wie *Speichern, OK, Öffnen, Abbrechen, Ja oder Nein* beschriftet. Sie lassen sich anhand der Methode `setText()` mit dem gewünschten Text beschriften und können für den Fall, dass der Button angeklickt wurde, durch ein `clicked`-Signal mit einer entsprechenden Aktion verbunden werden.

Methode	Beschreibung
<code>setDefault()</code>	Mit <code>True</code> als Argument in der Klammer wird der Button farblich hervorgehoben und als Standardauswahl gekennzeichnet.
<code>setEnabled()</code>	Mit <code>False</code> wird der Button deaktiviert.
<code>setText()</code>	Ändert die Beschriftung des Buttons.
<code>text()</code>	Gibt die Beschriftung des Buttons aus.
Signal	Beschreibung
<code>clicked()</code>	Löst beim Klick auf den Button die gewünschte Aktion aus.

**Tab. 15.1** Methoden und Signale der Klasse `QPushButton`



Ansicht eines vorausgewählten und eines deaktivierten Pushbuttons

**Radio Buttons** (Klasse: `QRadioButton`) bestehen aus einem klickbaren runden Knopf und einem Textlabel. Mit diesen Buttons lassen sich Optionen angeben, unter denen die Nutzerin oder der Nutzer jeweils nur eine auswählen kann. Hierfür müssen sich jedoch alle Radio Buttons innerhalb einer Gruppe mit demselben Eltern-Widget, beispielsweise einem unsichtbaren **Frame**-Widget, befinden. Existieren mehrere Eltern-Widgets, die jeweils wiederum mehrere Radio Buttons enthalten, kann pro Widget-Gruppe ein Button angeklickt werden. Beim Anklicken eines neuen Buttons wird ein zuvor ausgewählter Button automatisch deaktiviert.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

Methode	Beschreibung
isChecked()	Gibt an, ob die entsprechende Option ausgewählt wurde. Diese Methode kann als mögliche Werte die booleschen Datentypen True oder False annehmen.
setText()	Ändert die Beschriftung des Labels.
text()	Gibt den Inhalt des Textlabels aus.
Signal	Beschreibung
toggled()	Dieses Signal wird ausgelöst, wenn sich der Status des Buttons ändert, d. h. wenn er durch einen Klick aktiviert oder deaktiviert wurde.

**Tab. 15.2** Methoden und Signale der Klasse QRadioButton



Ansicht dreier Radiobuttons

Im Gegensatz zu Radio Buttons lassen sich mit **Check Box** mehrere Optionen gleichzeitig auswählen, die jeweils mit einer danebengestellten Beschriftung versehen sind. Durch das Setzen eines oder mehrerer Häkchen hat die Nutzerin oder der Nutzer die Möglichkeit, die gewünschten Optionen auszuwählen.

Ein Checkbox-Widget gehört zur Klasse QCheckBox und stellt unter anderem die folgenden Methoden und Signale bereit:

Methode	Beschreibung
checkState()	Zeigt den aktuellen Status der Checkbox an. Die möglichen Status sind <code>QtCore.Qt.Unchecked</code> , <code>QtCore.Qt.Partially Checked</code> und <code>QtCore.Qt.Checked</code> .
setCheckState(state)	Hier können Sie den gewünschten Status der Checkbox vorgeben, z. B. <code>setCheckState(QtCore.Qt.Checked)</code> .
setText()	Ändert die Beschriftung der Checkbox.
setTristate()	Mit True werden alle drei Status ermöglicht; mit False ist nur Unchecked und Checked vorgesehen.
Signal	Beschreibung
stateChanged()	Dieses Signal wird gesendet, wenn sich der Status der Checkbox durch Anklicken geändert hat.

**Tab. 15.3** Methoden und Signale der Klasse QCheckBox

## 15.5 Widgetoptionen: Klassen, Methoden und Signale

Um Angaben zum Status der Checkbox machen zu können, müssen Sie zusätzlich das Modul `QtCore` importieren.



Ansicht dreier Checkboxes

Das folgende Codebeispiel veranschaulicht die Verwendung der unterschiedlichen Methoden und Signale einer Checkbox mit dem Objektnamen `check`. Der Status wird dabei zu Anfang durch `setCheckState` standardmäßig auf `Checked` gesetzt. Durch `setTristate()` sind drei Zustände möglich, die jeweils eine entsprechende Beschriftung der Checkbox zur Folge haben.

### *Codebeispiel #8*

```

1  import sys
2  from PyQt5 import QtCore, QtWidgets, uic
3
4  class Checkbox(QtWidgets.QMainWindow):
5      def __init__(self):
6          QtWidgets.QMainWindow.__init__(self)  # Alternative: super()
7          self.ui = uic.loadUi("meintestprogramm.ui", self)
8          self.ui.check.stateChanged.connect(self.box_clicked)  # Signal & Slot
9          self.ui.check.setCheckState(QtCore.Qt.Checked)
10
11     def box_clicked(self):
12         self.check.setTristate(True)
13         if self.check.checkState() == QtCore.Qt.Checked:
14             self.check.setText('(checked)')
15         elif self.check.checkState() == QtCore.Qt.PartiallyChecked:
16             self.check.setText('(partially checked)')
17         else:
18             self.check.setText('(unchecked)')
19
20
21 app = QtWidgets.QApplication(sys.argv)
22 fenster = Checkbox()
23 fenster.show()
24 sys.exit(app.exec_())

```

15

In der Kategorie **Buttons** finden Sie drei weitere Widgets, die allesamt mit dem `clicked`-Signal ausgelöst werden:

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

Ein **Tool Button** (Klasse: `QToolButton`) ermöglicht den Schnellzugriff auf Befehle und Optionen, die anstelle eines Textlabels durch ein Icon dargestellt sind. Hierfür müssen Sie zusätzlich das Teilmodul `QtGui` importieren. Das Icon lässt sich durch die Methode `setIcon()` auswählen. Anschließend können Sie Ihrem Code folgende Zeilen hinzufügen, wobei `tool` der Objektname des Tool Buttons ist und der Dateipfad den entsprechenden Speicherort der Bilddatei auf Ihrem Computer angibt:

```

1 from PyQt5 import QtGui, QtWidgets, uic
2
3 icon = QtGui.QIcon("C:/Users/Sarah/Desktop/datei.png")
4 self.tool.setIcon(QtGui.QIcon(icon)) # in der Klassendefinition

```

**Command Link**-Widgets kommen häufig in Wizard-GUIs zum Einsatz, in denen sich beim Anklicken des Command Links ein weiteres Fenster öffnet. Sie befinden sich in der Klasse `QCommandLinkButton`.

Das Widget **Dialog Button Box** (Klasse: `QDialogButtonBox`) stellt mehrere Buttons in einem Layout bereit, das zum aktuellen Widgetstil passt.

### 15.5.2 Item Widgets

Ein **List**-Widget aus der Klasse `QListWidget` ermöglicht das Entfernen oder Hinzufügen von mehreren Einträgen, die in einer Listenansicht angezeigt werden.

Methode	Beschreibung
<code>addItem(name)</code>	Fügt dem Ende der Liste den Eintrag mit dem Inhalt von <code>name</code> hinzu.
<code>clear()</code>	Entfernt alle Listeneinträge.
<code>currentItem()</code>	Zeigt den aktuell ausgewählten Eintrag als <code>QListWidgetItem</code> -Instanz an, dessen Namen über die <code>text</code> -Methode angezeigt werden kann (Beispiel siehe unten).
Signal	Beschreibung
<code>currentItemChanged()</code>	Dieses Signal wird ausgelöst, wenn ein neuer Eintrag aus der Liste ausgewählt wurde.
<code>itemClicked()</code>	Wird ausgelöst, sobald ein Eintrag in der Liste angeklickt wurde.

**Tab. 15.4** Methoden und Signale der Klasse `QListWidget`

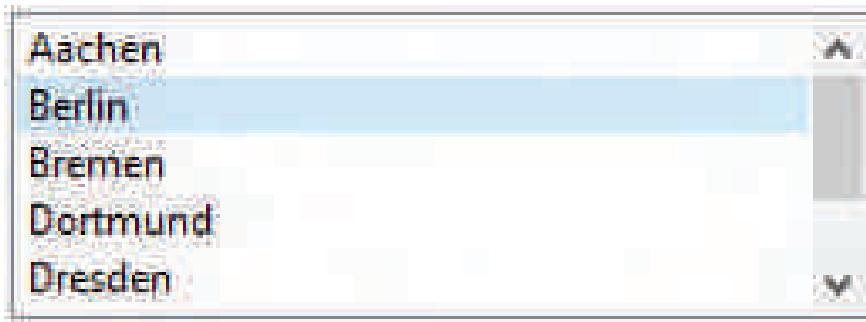
Auf die Aktivierung des `itemClicked`-Signals hin können Sie durch die Verbindung mit dem Slot `self.liste_geklickt` zum Beispiel folgende Klassenmethode definieren, die die angeklickte `QListWidgetItem`-Instanz samt deren Inhalt ausgibt:

## 15.5 Widgetoptionen: Klassen, Methoden und Signale

```

1 def liste_geklickt(self):
2     item = self.list.currentItem()
3     print(item, item.text())

```



Ansicht einer Liste mit Inhalt

Ähnlich wie die Darstellung von Informationen in einer Liste ermöglicht ein **Table Widget**, Einträge in Tabellenform anzeigen zu lassen. Wir werden dessen Funktionsweise anhand eines GUI-Anwendungsbeispiels für unser Bibliotheksprogramm im nächsten Teilkapitel genauer kennen lernen.

### 15.5.3 Container

Eine **Group Box** (Klasse: QGroupBox) gehört zur Kategorie der **Container**. Sie dient der thematischen Strukturierung von Widgets innerhalb eines Rahmens mit Titel. Ein Fenster kann mehrere Groupboxes enthalten, die sich anhand des gewünschten Layouts anordnen lassen. Sie können Ihre Groupbox durch einen Doppelklick am linken oberen Rand mit einem Titel beschriften und ihre Größe und Position im Dialogfenster oder im **Property Editor** vorgeben. Innerhalb einer Groupbox können Sie beliebig viele Widgets platzieren, auf die sich wie gewohnt über den Objektnamen zugreifen lässt. Die Anordnung der Widgets wird nicht automatisch vorgenommen, sondern lässt sich anhand der zuvor beschriebenen Layout-Methoden – Rechtsklick oder Layout-Toolbar – angeben.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

Methode	Beschreibung
setCheckable()	Optional kann die Groupbox mit True als Argument in der Klammer mithilfe einer Checkbox links neben dem Titel deaktiviert werden. Ist kein Häkchen gesetzt, lassen sich innerhalb der Groupbox keine Widgets anklicken.
setFlat()	Ändert die Optik der Groupbox zu einem platzsparendem Design.
Signal	Beschreibung
clicked()	Ist die Groupbox mit setCheckable(True) durch ein Häkchen versehen, kann beim Anklicken des Titels oder Häkchens das clicked-Signal ausgelöst und eine entsprechende Funktion ausgeführt werden.

Tab. 15.5 Methoden und Signale der Klasse QGroupBox

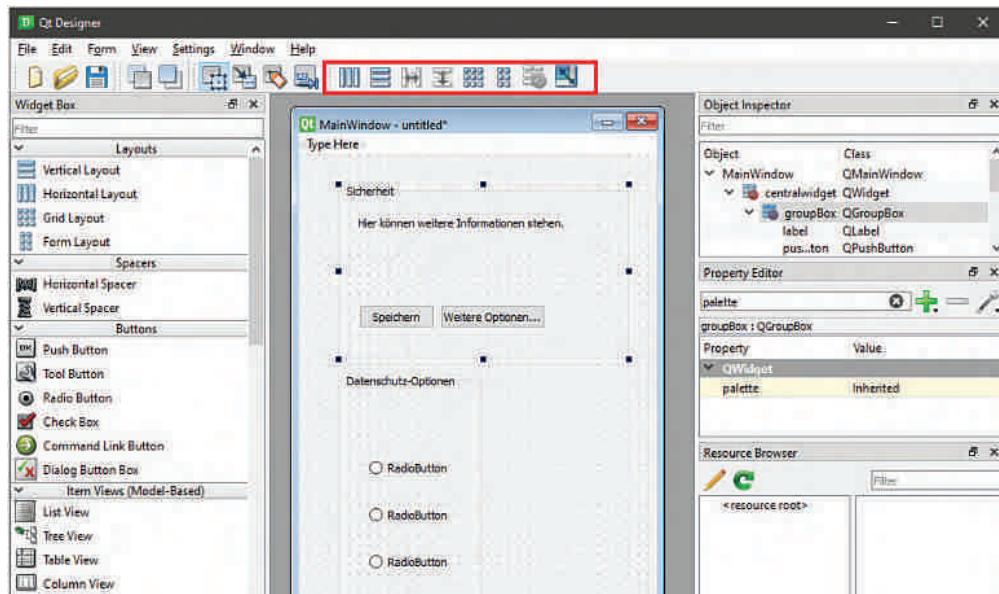


Abb. 15.17 Die Verwendung zweier Groupboxes in einem Fenster und Ansicht der Layout-Toolbar

Weitere Container-WIDGETS sind beispielsweise das WIDGET **Scroll Area** (Klasse: `QScrollArea`), das für größere Anzeigebereiche – etwa bei Bilddateien – Scrollbalken zum Navigieren vorsieht.

Mit dem **Tab**-WIDGET aus der Klasse `QTabWidget` lassen sich mehrere hintereinander angeordnete anklickbare Tab-Seiten definieren, die wiederum unterschiedliche WIDGETS enthalten.

## 15.5 Widgetoptionen: Klassen, Methoden und Signale

Ein **Frame**-Widget (Klasse: `QFrame`) ähnelt einer Groupbox, hat jedoch keinen sichtbaren Rahmen oder Titel. Für dieses Widget lassen sich insbesondere weitere Designoptionen wie ein Format und Schattierungen festlegen.

Die Klasse `QMdiArea` stellt das **MDI-Area**-Widget bereit. Wie der Name erraten lässt – *MDI* steht für „Multi Document Interface“ – ermöglicht dieses Widget die gleichzeitige Anzeige mehrerer Fenster innerhalb eines Hauptfensters.

Allgemein kann auch ein **Widget**-Container aus der Basisklasse `QWidget` als Container verwendet werden, der weitere untergeordnete Widgets enthält.

### 15.5.4 Input Widgets

Mit einer **Combo Box** (Klasse: `QComboBox`) können Sie eine Listenauswahl in Form eines Drop-down-Menüs realisieren. Sie wird eingesetzt, wenn unter mehreren vorgegebenen Einträgen ein Eintrag ausgewählt werden soll, beispielsweise bei der Angabe von Ländern oder einer Anrede.

<b>Methode</b>	<b>Beschreibung</b>
<code>addItem(text)</code>	Fügt dem Ende der Combobox den Eintrag mit dem Inhalt von <code>text</code> hinzu.
<code>clear()</code>	Entfernt alle Einträge aus der Combobox.
<code>currentIndex()</code> , <code>currentText()</code>	Zeigt den Index bzw. Namen des aktuell ausgewählten Eintrags an.
<b>Signal</b>	<b>Beschreibung</b>
<code>activated()</code>	Wird beim Anklicken eines Eintrags gesendet.
<code>currentIndexChanged()</code>	Dieses Signal wird ausgelöst, wenn sich der Index eines ausgewählten Eintrags ändert.

**Tab. 15.6** Methoden und Signale der Klasse `QComboBox`

Es ist ebenfalls möglich, die zur Auswahl stehenden Einträge direkt im Qt Designer anzugeben. Hierfür machen Sie einen Doppelklick auf die Combobox und geben im neuen Fenster unter „+“ die gewünschten Optionen ein. Im folgenden Code wird etwa beim Anklicken einer Länder-Option deren Index und Inhalt ausgegeben:

#### Codebeispiel #9

15

```

1 import sys
2 from PyQt5 import QtWidgets, uic
3
4 class Combobox(QtWidgets.QMainWindow):
5     def __init__(self):

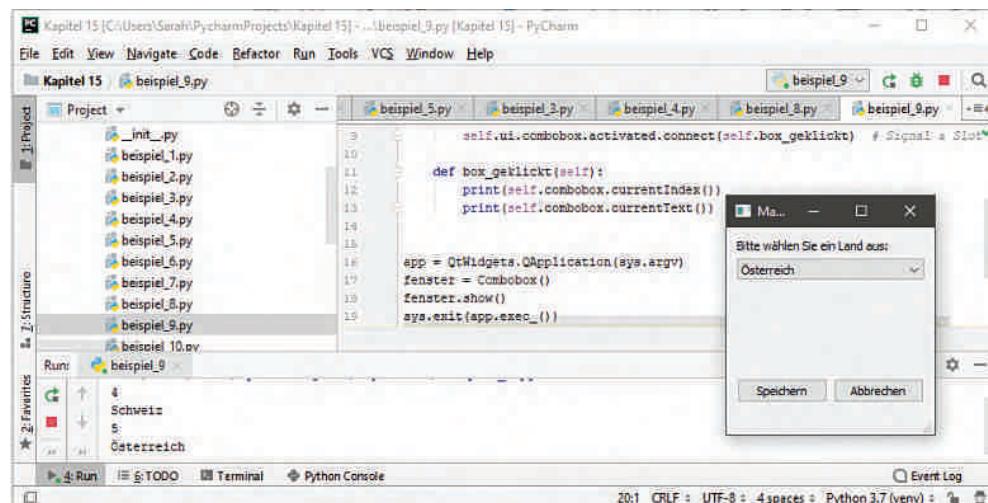
```

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

```

6         QtWidgets.QMainWindow.__init__(self) # Alternative: super()
7         self.ui = uic.loadUi("testprogramm2.ui", self)
8         self.ui.comboBox.activated.connect(self.box_geklickt) # Signal & Slot
9
10    def box_geklickt(self):
11        print(self.comboBox.currentIndex())
12        print(self.comboBox.currentText())
13
14
15    app = QtWidgets.QApplication(sys.argv)
16    fenster = Combobox()
17    fenster.show()
18    sys.exit(app.exec_())
19
20

```



**Abb. 15.18** Ausgabe beim Anklicken von Einträgen einer Combobox

Mithilfe eines **Line-Edit**-Widgets aus der Klasse `QLineEdit` können Benutzerinnen und Benutzer Informationen in ein einzeiliges Texteingabefeld eingeben. Oftmals wird ein Line Edit zusätzlich mit einem danebengestellten Label versehen, in dem auf den gewünschten Inhalt für das Eingabefeld hingewiesen wird.

Methode	Beschreibung
<code>clear()</code>	Löscht den Inhalt des Felds.
<code>setReadOnly()</code>	Mit True als Argument in der Klammer kann der Text im Eingabefeld nicht durch eine Nutzereingabe geändert werden.
<code>setText(text)</code>	Füllt das Eingabefeld mit dem angegebenen Textinhalt.
<code>text()</code>	Zeigt den Text des Felds an.

## 15.5 Widgetoptionen: Klassen, Methoden und Signale

Signal	Beschreibung
textChanged ()	Wird bei einer Änderung des Textinhalts ausgelöst.

**Tab. 15.7** Methoden und Signale der Klasse QLineEdit



Zwei Line Edits mit danebengestellten Labels

Dementsprechend stellt das Widget **Text Edit** (Klasse: QTextEdit) ein mehrzeiliges Eingabefeld bereit, in das umfassendere Textinhalte eingegeben werden können. Zu den verfügbaren Klassenmethoden gehören `setPlainText(text)` zum Eingeben des gewünschten Inhalts sowie `toPlainText()` zum Ausgeben des Inhalts. Als Signal wird hier ebenfalls `textChanged()` verwendet.

Eine ähnliche Funktionalität wie Line Edit und Text Edit wird durch **Spin Box** und **Double Spin Box** realisiert, wobei die Textfelder hier aus ganzen Zahlen bzw. Fließkommazahlen bestehen, zwischen denen durch zwei Knöpfe (oben/unten) am rechten Rand navigiert werden kann. Mit den dazugehörigen Methoden lassen sich unter anderem Maximum- und Minimumwerte sowie die Schrittweite zwischen den Zahlen angeben.

Weitere praktische Widgets finden Sie in den Klassen `QTimeEdit`, `QDateEdit` und `QDateTimeEdit`, mit denen sich Zeitangaben machen lassen. Mit dem Widget **Time Edit** können Nutzer und Nutzerinnen in einem Texteingabefeld eine bestimmte Uhrzeit, mit **Date Edit** ein Datum sowie mit **Date/Time Edit** beide Optionen festlegen. Mögliche Methoden sind unter anderem `setMinimumTime` und `setMaximumTime`, `minimumTime` und `maximumTime`, `setTime` und `time`. Analog zu `time` können Sie in den genannten Methoden den Begriff `date` verwenden. Achten Sie dabei auf Groß- und Kleinschreibung. Zu den verfügbaren Signalen gehören `dateChanged`, `dateTimeChanged` und `timeChanged`.

### 15.5.5 Display Widgets

Unter den Display Widgets besteht mit dem **Calendar Widget** ebenfalls die Option, anhand eines größeren Kalenders ein gewünschtes Datum anzugeben.

Ein weiteres nützliches Widget in dieser Kategorie ist das **Label**-Widget aus der Klasse `QLabel`. Hierbei handelt es sich um ein sehr einfaches Widget, das standardmäßig keinerlei Interaktionsmöglichkeit bietet und aus einer einzigen Zeile besteht. Die ver-

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

fügbaren Methoden lassen Layoutmöglichkeiten und die Spezifizierung des Label-Textinhalts zu. Signale lassen sich nur angeben, wenn der Labelinhalt aus einem Link zu einer Webseite besteht. In diesem Fall existiert das Signal `linkActivated`, wenn der Link angeklickt wird sowie `linkHovered`, wenn der Nutzer mit der Maus über den Link fährt.

Mit dem Widget **Progress Bar** (Klasse: `QProgressBar`) können Sie Ihre GUI mit einem Fortschrittsbalken versehen, der den Fortschritt eines längeren Prozesses, beispielsweise eines Installationsvorgangs, anzeigt. Der Balken, dessen Wertebereichsgrenzen zuvor festgelegt werden können, kann eine horizontale oder vertikale Ausrichtung haben. Er bietet keinerlei Möglichkeit zur Interaktion und stellt die folgenden Methoden bereit:

Methode	Beschreibung
<code>setMinimum(minimum)</code> , <code>setMaximum(maximum)</code>	Zur Festlegung des unteren bzw. oberen Grenzwerts, jeweils eine ganze Zahl.
<code>setOrientation(orientation)</code>	Mögliche Werte für eine horizontale oder vertikale Ausrichtung sind <code>QtCore.Qt.Horizontal</code> bzw. <code>QtCore.Qt.Vertical</code> . Um diese Angabe machen zu können, müssen Sie zusätzlich das Modul <code>QtCore</code> importieren.
<code>setValue(value)</code>	Gibt den Wert an, der aktuell angezeigt werden soll, ebenfalls eine ganze Zahl.

**Tab. 15.8** Methoden und Signale der Klasse `QProgressBar`



Anzeige des Fortschritts mit Progress Bar

Wenn Sie mehr über Widgets und deren Einsatzmöglichkeiten wissen möchten, können Sie sich beispielsweise mit der Model-View-Architektur beschäftigen.

## 15.6 Anwendungsbeispiel: Bibliotheksprogramm

Wir werden nun eine vereinfachte GUI für unser Bibliotheksprogramm aus den vorigen Kapiteln erstellen. Diese wird aus zwei Fenstern bestehen: Im ersten Hauptfenster sollen die Nutzerinnen und Nutzer die Möglichkeit haben, die entsprechenden Infor-

## 15.6 Anwendungsbeispiel: Bibliotheksprogramm

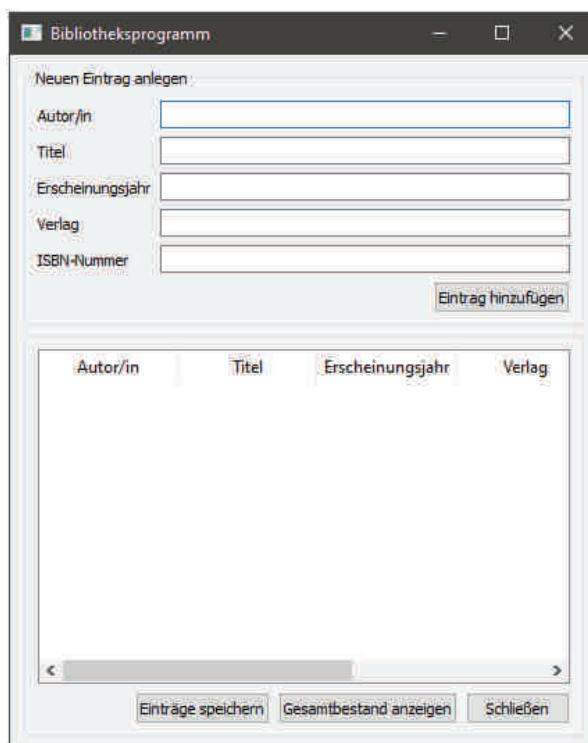
mationen für Werke im Bibliotheksbestand einzugeben und in einer Liste mit neuen Einträgen anzuzeigen.

Anschließend können neue Einträge gespeichert und dem Gesamtbestand hinzugefügt werden, was durch eine SQLite-Datenbank realisiert wird. Der in dieser Datenbank angelegte Gesamtbestand kann sodann in einem zweiten Fenster eingesehen werden, in dem ebenfalls die Gesamtanzahl aller Werke angezeigt wird. In beiden Fenstern besteht die Möglichkeit, das Fenster ohne weitere Aktionen zu schließen.

Wir werden die beiden Fenster nun schrittweise erstellen und die nötige Funktionalität durch Signale und Slots hinzufügen. Bei umfassenderen graphischen Benutzeroberflächen ist es immer eine gute Idee, langsam vorzugehen und sich im Voraus über Ziel, Struktur und den allgemeinen Aufbau im Klaren zu sein, um hinterher unnötigen Arbeitsaufwand zu vermeiden.

### 15.6.1 Das Aussehen des ersten Fensters festlegen

Zunächst legen wir im Qt Designer ein neues **Main Window** namens *bibliotheksprogramm.ui* an. Das Hauptfenster unseres Programms soll das folgende Aussehen haben:



15

**Abb. 15.19** Hauptfenster zum Anlegen eines neuen Eintrags im Bibliotheksprogramm

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

Innerhalb eines Frame-Widgets haben wir hierbei zwei Groupboxes in vertikaler Anordnung platziert. Die obere Groupbox enthält in Rasteranordnung jeweils fünf Line-Edit-Widgets mit entsprechenden Label-Beschriftungen sowie darunter einen horizontalen Spacer und einen Push Button, die horizontal miteinander verbunden sind. Die Line Edits haben wir ihren Beschriftungen entsprechend in `autor_in`, `titel`, `jahr`, `verlag` und `isbn` umbenannt, der Button **Eintrag hinzufügen** hat den Objektnamen `hinzufuegen`.

Die zweite Groupbox enthält in vertikaler Anordnung ein Table-Widget (Objektname: `tabelle`), sowie darunter einen horizontalen Spacer und die drei Buttons **Einträge speichern** (Objektname: `speichern`), **Gesamtbestand anzeigen** (anzeigen) und **Schließen** (`schliessen`), die wiederum durch ein horizontales Layout verbunden sind. Mit einem Doppelklick auf die Tabelle lassen sich in einem neuen Fenster die gewünschten Titel der Zeilen und Spalten angeben. Als Spalten geben wir hier *Autor/in*, *Titel*, *Erscheinungsjahr*, *Verlag* und *ISBN-Nummer* an.

Um festzulegen, in welcher Tabulatorreihenfolge die Zeilen später mit Text ausgefüllt werden, gehen Sie in der Menüleiste auf **Edit → Edit Tab Order**. Nun können Sie die Zeilen und andere Elemente wie Buttons in derjenigen Reihenfolge anklicken, in der der Tabulator springen soll. Mit **F3** bzw. **Edit → Edit Widgets** kehren Sie anschließend zur Bearbeitungsansicht zurück.

Die Statusbar am unteren Fensterrand können Sie entfernen, indem Sie einen rechten Mausklick auf den äußeren Fensterbereich machen und **Remove Status Bar** auswählen.

Klicken Sie abschließend auf den Tool Button **Adjust Size (Strg + J)**, um automatisch die passenden Größenverhältnisse der Widgets untereinander zu schaffen.

Nachdem Sie das aktuelle Fenster in der Datei `bibliotheksprogramm.ui` abgespeichert haben, können Sie dieses wie gewohnt in Python-Code verwenden und die für die Verwendung nötigen Signale und Slots einfügen.

### 15.6.2 Die Fensterwidgets mit Signalen und Slots ausstatten

Unser Bibliotheksprogramm-Fenster wird für jeden der vier Buttons jeweils ein Signal mit einem entsprechenden Slot bereithalten. Das erste Signal wird bei einem Klick auf den Button **Eintrag hinzufügen** ausgelöst:

```
1 self.ui.hinzufuegen.clicked.connect(self.in_tabelle)
```

Die dazugehörige Methode `in_tabelle` definieren wir folgendermaßen:

## 15.6 Anwendungsbeispiel: Bibliotheksprogramm

```

1 def in_tabelle(self):
2     self.tabelle.insertRow(self.tabelle.rowCount())    # gibt die aktuelle
3     Zeilenanzahl zurück
4     self.tabelle.setItem(self.tabelle.rowCount() - 1, 0, QtWidgets.
5     QTableWidgetItem(self.autor_in.text()))
6     self.tabelle.setItem(self.tabelle.rowCount() - 1, 1, QtWidgets.
7     QTableWidgetItem(self.titel.text()))
8     self.tabelle.setItem(self.tabelle.rowCount() - 1, 2, QtWidgets.
9     QTableWidgetItem(self.jahr.text()))
10    self.tabelle.setItem(self.tabelle.rowCount() - 1, 3, QtWidgets.
11    QTableWidgetItem(self.verlag.text()))
12    self.tabelle.setItem(self.tabelle.rowCount() - 1, 4, QtWidgets.
13    QTableWidgetItem(self.isbn.text()))
14    self.autor_in.clear()
15    self.titel.clear()
16    self.jahr.clear()
17    self.verlag.clear()
18    self.isbn.clear()

```

Hier gibt die Methode `insertRow` zunächst an, in welcher Zeile der neue Eintrag eingefügt werden soll. Als Parameter wird dabei mit `rowCount` stets die aktuelle Zeilenanzahl gewählt, die beim Anlegen des ersten Eintrags durch `insertRow` auf 1 gesetzt wird und sich beim Einfügen jedes weiteren Eintrags um 1 erhöht. Somit wird sicher gestellt, dass keine bereits angelegten Zeilen überschrieben werden. Dies wird aus den nächsten Codezeilen ersichtlich:

Die Methode `setItem` der Klasse `QTableWidgetItem` fügt der Tabelle an der gewünschten Stelle denjenigen Text hinzu, der im entsprechenden Line-Edit-Widget eingegeben wurde. In den Klammern der `setItem`-Methode steht dabei zunächst die Zeilenposition, dann die Spaltenposition und danach der eingegebene Textinhalt. Die Zählung der Zeilen- und Spaltenpositionen in der Tabelle beginnt – wie etwa auch bei Listen und Tupeln üblich – bei 0.

Die Zeilenposition ist in unserem Fall durch die aktuelle Zeilenanzahl – 1 definiert, da die Zeilenanzahl durch die obige `insertRow`-Methode beim Hinzufügen des ersten Eintrags bereits auf 1 gesetzt wurde und sich mit jedem weiteren neuen Eintrag um 1 erhöht.

Analog geht die Spaltenzählung für unsere fünf vorgesehenen Spalten von 0 bis 4. Das dritte Element in der Klammer legt den Inhalt des Tabellenfelds an und hat stets die Form `QtWidgets.QTableWidgetItem()`. In Klammern können Sie ebenfalls direkt einen String eingeben. Da wir den Inhalt der Tabelle je nach Nutzereingabe dynamisch gestalten wollen, verwenden wir hier die `text`-Methode des Tabellen-Widgets, das den jeweiligen Line-Edit-Inhalt in das Tabellenfeld einfügt.

Anschließend werden die Line-Edit-Eingaben mit `clear()` gelöscht, sodass die Nutzerinnen und Nutzer dort neue Angaben machen können.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

Das zweite Signal wird bei einem Klick auf den Button **Einträge speichern** ausgelöst:

```
1 self.ui.speichern.clicked.connect(self.tabellen_speichern)
```

Die Methode `tabellen_speichern` legt daraufhin eine – zunächst leere – Liste an. Mithilfe einer `for`-Schleife wird für jede Tabellenzeile sodann ein Tupel mit allen fünf Werkseinträgen erstellt. Diese Tupel werden in jedem Iterationsschritt der `for`-Schleife nach und nach der Liste hinzugefügt, bis die Liste den gesamten Tabelleninhalt enthält. Danach wird die Liste einer neuen Funktion, `sql_datenbank`, übergeben:

```
1 def tabellen_speichern(self):
2     liste = []
3     for i in range(self.tabelle.rowCount()):
4         inhalt = (self.tabelle.item(i, 0).text(), self.tabelle.item(i,
5             1).text(),
6                     self.tabelle.item(i, 2).text(), self.tabelle.item(i,
7                         3).text(),
8                         self.tabelle.item(i, 4).text())
9         liste.append(inhalt)
10    self.sql_datenbank(liste)
```

Der Übersichtlichkeit halber ist es vorteilhaft, längere Programmierungsvorgänge ihren Aufgaben gemäß in unterschiedliche Methoden aufzuteilen. So übernimmt die Methode `sql_datenbank` die Liste aus der vorigen Methode und legt eine SQLite-Datenbank namens `daten.db` mit der Tabelle `bibliotheksbestand` und dem uns bekannten Listeninhalt an. Die Funktionsweise einer SQLite-Tabelle sollte Ihnen noch aus dem vorigen Kapitel 14.2. bekannt sein:

```
1 def sql_datenbank(self, liste):
2     verbindung = sqlite3.connect("daten.db")
3     cursor = verbindung.cursor()
4     cursor.execute(
5         """CREATE TABLE IF NOT EXISTS bibliotheksbestand(autor_in TEXT, titel
6             TEXT, erscheinungsjahr TEXT, verlag TEXT, isbn TEXT)"""
7     )
8     cursor.executemany("INSERT INTO bibliotheksbestand VALUES(?, ?, ?, ?, ?)", liste)
9     verbindung.commit()
10    verbindung.close()
```

Die letzten beiden Signale definieren wir folgendermaßen:

```
1 self.ui.anzeigen.clicked.connect(self.bestand_anzeigen)
2 self.ui.schliessen.clicked.connect(self.programm_schliessen)
```

Sie veranlassen folgende Methoden:

```
1 def bestand_anzeigen(self):
2     fenster_bestand = Bestandsfenster()
3     fenster_bestand.show()
4
```

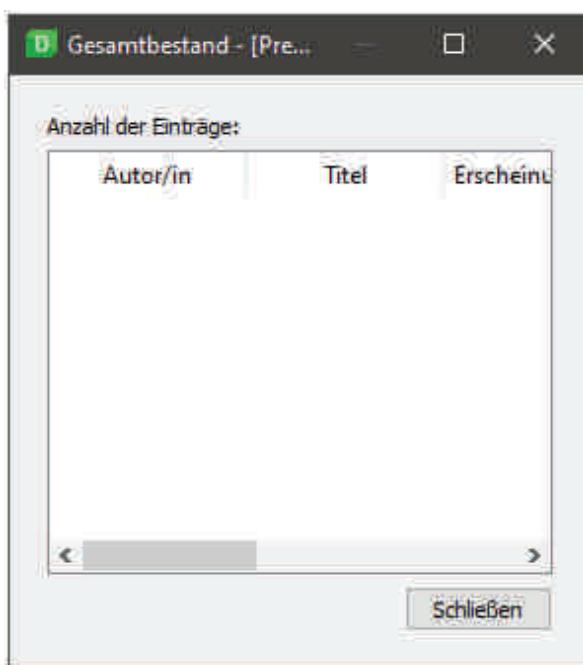
## 15.6 Anwendungsbeispiel: Bibliotheksprogramm

```
5 def programm_schliessen(self):
6     self.close()
```

Während `programm_schliessen` das Bibliotheksprogramm-Fenster schließt, öffnet `bestand_anzeigen` ein neues Fenster, in dem der aktuelle Gesamtbestand angezeigt wird. Dieses Fenster werden wir nun ebenfalls im Qt Designer erstellen.

### 15.6.3 Das Aussehen des zweiten Fensters vorgeben

Im nächsten Schritt legen wir im Qt Designer ein neues **Main Window** namens *gesamtbestand.ui* an. Es besteht ebenfalls aus einem Frame-Widget, innerhalb dessen sich horizontal angeordnet ein Label (Objektname: `anzahl1`), eine Tabelle (Objektname: `tabelle2`) sowie – wiederum zu einem horizontalen Layout verbunden – ein Spacer und ein Push Button (Objektname: `schliessen`) befinden. Die Tabelle versehen wir wie zuvor mit denselben gewünschten Spaltennamen. Klicken Sie am Ende auf den Tool Button **Adjust Size**, um die Größenverhältnisse anzupassen:



**Abb. 15.20** Anzeige des Gesamtbestands in einem neuen Fenster

In diesem Fenster wollen wir uns den Inhalt der SQLite-Datenbank anzeigen lassen. Wir benötigen für unser vereinfachtes Bibliotheksprogramm also die beiden CRUD-Operationen *Create* zum Anlegen der Tabelle (Abschnitt 15.6.2.) sowie *Read* zum Auslesen des Tabelleninhalts. Der SQLite-Befehl zum Anzeigen des Inhalts der Gesamtbestand-Tabelle wird beim Öffnen des Fensters automatisch ausgeführt.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

### 15.6.4 Die Funktionalität des Gesamtbestand-Fensters festlegen

Wir definieren die Klasse Bestandsfenster nun wie folgt:

```

1 class Bestandsfenster(QtWidgets.QMainWindow):
2     def __init__(self):
3         QtWidgets.QMainWindow.__init__(self) # Alternative: super()
4         self.ui = uic.loadUi(r"C:\Users\Sarah\PycharmProjects\Kapitel 15\
5             gesamtbestand.ui", self)
6         self.ui.schliessen.clicked.connect(self.fenster_schliessen)
7         self.gesamtbestand()
8
9     def gesamtbestand(self):
10        verbindung = sqlite3.connect("daten.db")
11        cursor = verbindung.cursor()
12        cursor.execute("""SELECT * FROM bibliotheksbestand""")
13        zeilen = cursor.fetchall()
14        self.tabelle2.setRowCount(len(zeilen))
15        for i in range(len(zeilen)):
16            for j in range(5):
17                self.tabelle2.setItem(i, j, QtWidgets.
18                    QTableWidgetItem(zeilen[i][j]))
19        self.anzahl.setText("Anzahl der Einträge: {}".format(self.tabelle2.
20            rowCount()))
21        verbindung.close()
22
23    def fenster_schliessen(self):
24        self.close()

```

Sie verfügt lediglich über ein einziges `clicked`-Signal, bei dem das Gesamtbestand-Fenster geschlossen wird, nachdem auf den Button **Schließen** geklickt wurde.

Die übrige Funktionalität wird beim Öffnen des Fensters automatisch ausgeführt. Sie besteht hauptsächlich aus der Klassenmethode `gesamtbestand`, die aus unserer SQLite-Datenbank die Informationen mittels `cursor.fetchall` ausliest und in der Variablen `zeilen` abspeichert. Die Variable `zeilen` ist dabei eine Liste, innerhalb derer jeder Datensatz bzw. jede Zeile wiederum in einem Tupel steht. Für jeden solchen Datensatz legen wir nun mithilfe einer verschachtelten `for`-Schleife unsere Tabelle an.

Nachdem die Schleife alle Einträge durchlaufen hat, lassen wir uns im `anzahl`-Label die mit `rowCount` ermittelte Zeilenanzahl durch die `setText`-Methode anzeigen.

### 15.6.5 Das Bibliotheksprogramm verwenden

Der gesamte Code für unser vereinfachtes Bibliotheksprogramm hat die folgende Form:

## 15.6 Anwendungsbeispiel: Bibliotheksprogramm

## Codebeispiel #10

```

1 import sys
2 import sqlite3
3 from PyQt5 import QtWidgets, uic
4
5 class Eingabefenster(QtWidgets.QMainWindow):
6     def __init__(self):
7         QtWidgets.QMainWindow.__init__(self) # Alternative: super()
8         self.ui = uic.loadUi(r"C:\Users\Sarah\PycharmProjects\Kapitel 15\bibliotheksprogramm.ui", self)
9         self.ui.hinzufuegen.clicked.connect(self.in_tabelle)
10        self.ui.speichern.clicked.connect(self.tabelle_speichern)
11        self.ui.anzeigen.clicked.connect(self.bestand_anzeigen)
12        self.ui.schliessen.clicked.connect(self.programm_schliessen)
13
14
15    def in_tabelle(self):
16        self.tabelle.insertRow(self.tabelle.rowCount()) # gibt die aktuelle
17        Zeilenanzahl zurück
18        self.tabelle.setItem(self.tabelle.rowCount() - 1, 0, QtWidgets.
19        QTableWidgetItem(self.autor_in.text()))
20        self.tabelle.setItem(self.tabelle.rowCount() - 1, 1, QtWidgets.
21        QTableWidgetItem(self.titel.text()))
22        self.tabelle.setItem(self.tabelle.rowCount() - 1, 2, QtWidgets.
23        QTableWidgetItem(self.jahr.text()))
24        self.tabelle.setItem(self.tabelle.rowCount() - 1, 3, QtWidgets.
25        QTableWidgetItem(self.verlag.text()))
26        self.tabelle.setItem(self.tabelle.rowCount() - 1, 4, QtWidgets.
27        QTableWidgetItem(self.isbn.text()))
28        self.autor_in.clear()
29        self.titel.clear()
30        self.jahr.clear()
31        self.verlag.clear()
32        self.isbn.clear()
33
34    def tabelle_speichern(self):
35        liste = []
36        for i in range(self.tabelle.rowCount()):
37            inhalt = (self.tabelle.item(i, 0).text(), self.tabelle.item(i,
38            1).text(),
39                        self.tabelle.item(i, 2).text(), self.tabelle.item(i,
40            3).text(),
41                        self.tabelle.item(i, 4).text())
42            liste.append(inhalt)
43        self.sql_datenbank(liste)
44
45    def sql_datenbank(self, liste):
46        verbindung = sqlite3.connect("daten.db")
47        cursor = verbindung.cursor()
48        cursor.execute(
49            """CREATE TABLE IF NOT EXISTS bibliotheksbestand(autor_in TEXT,
50            titel TEXT, erscheinungsjahr TEXT, verlag TEXT, isbn TEXT)"""
51            )
52        cursor.executemany("INSERT INTO bibliotheksbestand VALUES(?, ?, ?, ?, ?)", liste)
53        verbindung.commit()
54        verbindung.close()

```

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

```

55
56     def bestand_anzeigen(self):
57         fenster_bestand = Bestandsfenster()
58         fenster_bestand.show()
59
60     def programm_schliessen(self):
61         self.close()
62
63
64 class Bestandsfenster(QtWidgets.QMainWindow):
65     def __init__(self):
66         QtWidgets.QMainWindow.__init__(self) # Alternative: super()
67         self.ui = uic.loadUi(r"C:\Users\Sarah\PycharmProjects\Kapitel 15\
68 gesamtbestand.ui", self)
69         self.ui.schliessen.clicked.connect(self.fenster_schliessen)
70         self.gesamtbestand()
71
72     def gesamtbestand(self):
73         verbindung = sqlite3.connect("daten.db")
74         cursor = verbindung.cursor()
75         cursor.execute("""SELECT * FROM bibliotheksbestand""")
76         zeilen = cursor.fetchall()
77         self.tabelle2.setRowCount(len(zeilen))
78         for i in range(len(zeilen)):
79             for j in range(5):
80                 self.tabelle2.setItem(i, j, QtWidgets.
81 QTableWidgetItem(zeilen[i][j]))
82             self.anzahl.setText("Anzahl der Einträge: {}".format(self.tabelle2.
83 rowCount())))
84         verbindung.close()
85
86     def fenster_schliessen(self):
87         self.close()
88
89
90 app = QtWidgets.QApplication(sys.argv)
91 fenster = Eingabefenster()
92 fenster.show()
93 sys.exit(app.exec_())

```

Achten Sie hierbei insbesondere darauf, die Dateipfade der *.ui*-Dateien dem Speicherort auf Ihrem Computer entsprechend anzupassen.

Wenn Sie das Programm in PyCharm laufen lassen, können Sie dessen Funktionalität testen. Sie können nun etwa neue Werkseinträge eingeben, diese durch **Eintrag hinzufügen** in einer Liste anzeigen lassen und anschließend mit **Einträge speichern** in einer Datenbankdatei abspeichern. Die Tabelle bietet ebenfalls die Funktionalität, die eingegebenen Informationen vor dem Abspeichern zu bearbeiten bzw. eventuelle fehlerhafte Informationen zu korrigieren. Hierzu machen Sie einfach einen Doppelklick in das entsprechende Feld und ändern die Angaben dementsprechend.

## 15.6 Anwendungsbeispiel: Bibliotheksprogramm

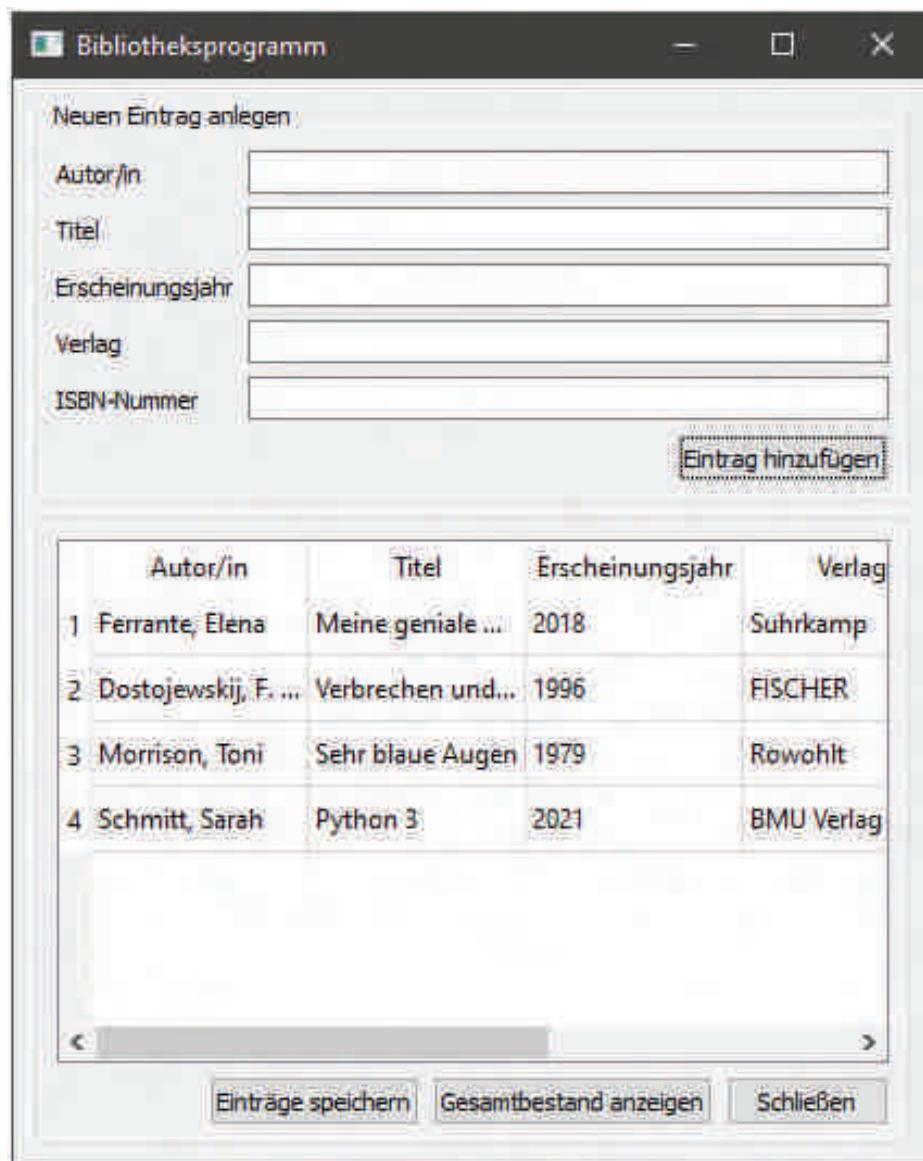
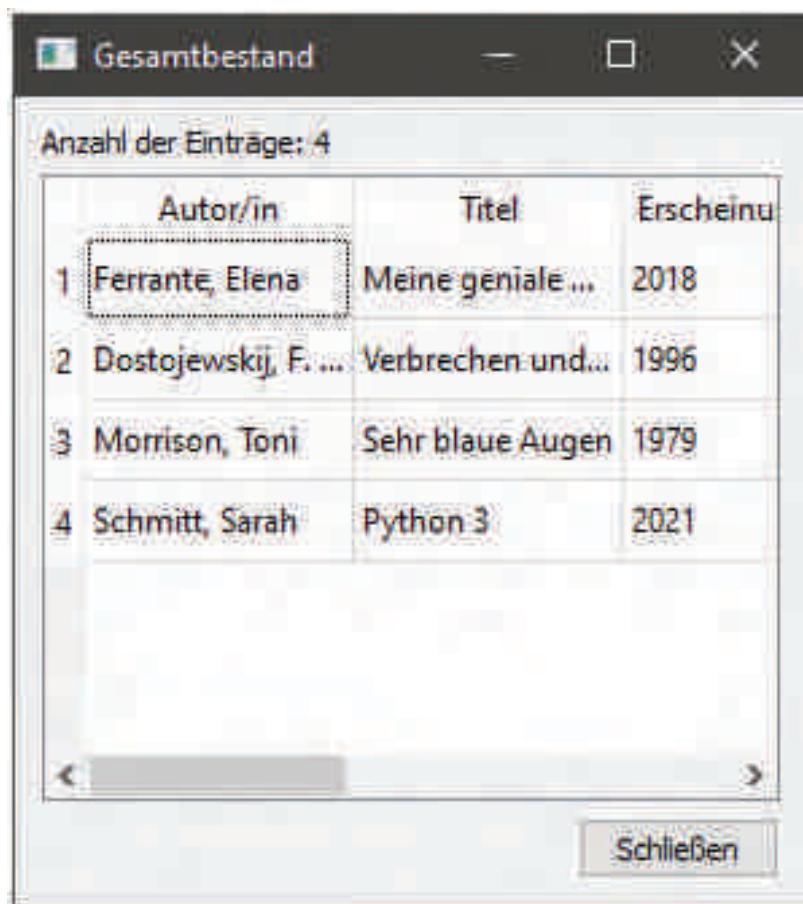


Abb. 15.21 Hinzufügen von Werkseinträgen

Wenn Sie die Einträge gespeichert haben, können Sie sich diese mit **Gesamtbestand anzeigen** im nächsten Fenster ausgeben lassen:

## 15 Grafische Benutzeroberflächen mit PyQt erstellen



**Abb. 15.22** Den Gesamtbestand anzeigen

Wie Sie sehen, wird die Anzahl der Einträge automatisch im dafür vorgesehenen Label angezeigt.

Für eine fortgeschrittenere, realistischere Anwendung können Sie den Code selbstverständlich variieren und weitere sinnvolle Fälle und Optionen vorsehen. Denkbar ist hier beispielsweise eine Erweiterung durch die Datenoperationen *Update* und *Delete* sowie die automatische Erkennung doppelt eingetippter oder unvollständiger Einträge. Ebenso nützlich wäre es, Ausnahmen durch unvorhergesehene Nutzeraktionen abzudecken – zum Beispiel, wenn der Gesamtbestand abgerufen wird, ohne dass zuvor eine Datenbankdatei mit Einträgen angelegt wurde.

## 15.7 Übung: Programme mit Fenstern selbst gestalten

### 15.7 Übung: Programme mit Fenstern selbst gestalten

#### Übungsaufgaben

1. Fügen Sie im Gesamtbestand-Fenster unseres Bibliotheksprogramms einen weiteren Button **Eintrag löschen** ein, der einen angeklickten Eintrag sowohl aus der Gesamtbestand-Tabelle als auch aus der SQLite-Datenbank löscht. Die Anzeige *Anzahl der Einträge* im Gesamtbestand-Fenster soll dabei automatisch aktualisiert werden. Sehen Sie für die beiden Vorgänge zwei verschiedene Klassenmethoden vor und lassen Sie sich den aktuellen Bibliotheksbestand nach dem Löschvorgang in der IDE ausgeben.

**Tipp:** Die Methode `currentRow()` wählt die angeklickte Zeile der Gesamtbestand-Tabelle aus, während `removeRow()` die aktuelle Zeile löscht. Aus der SQLite-Datenbank lässt sich der Eintrag beispielsweise über die ISBN-Nummer löschen, die mit einem ?-Platzhalter als Variable für ein zweielementiges Tupel nach dem Schema (`isbn_nr,` ) angegebenen werden kann.

2. a) Erstellen Sie mit Qt Designer einen Taschenrechner, der aus einem Line Edit zum Anzeigen der Berechnungen sowie Buttons mit den Zahlen 0-9, den vier Grundrechenarten (+, -, \*, /), einem Gleichheitszeichen und einer C-Taste zum Löschen der Eingabe besteht. Achten Sie darauf, das richtige Layout zu verwenden, damit das Fenster korrekt angezeigt wird.  
b) Versehen Sie die Buttons anschließend mit den entsprechenden Signalen und Methoden, um Berechnungen zu ermöglichen. Suchen Sie ebenfalls nach einer passenden Methode für das Line-Edit-Widget, die direkte Nutzereingaben in das Texteingabefeld unterbindet, sodass lediglich Änderungen über die Buttons vorgenommen und angezeigt werden. Alle benötigten Methoden und Signale finden Sie unter den Angaben zum Line-Edit-Widget in Abschnitt 15.5.4. Zum Handhaben fehlerhafter Eingaben sollten Sie zudem Ausnahmen mit `SyntaxError` und `ZeroDivisionError` definieren.

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

### Lösungen

- Zunächst versehen wir unser Fenster mit einem zusätzlichen Button mit dem Objektnamen `loeschen`, der durch ein `clicked`-Signal Einträge aus der Tabelle löscht. Die Klassendefinition erweitern wir dementsprechend durch die folgende Zeile:

```
1 self.ui.loeschen.clicked.connect(self.eintrag_loeschen)
```

Anschließend fügen wir unserer `Bestandsfenster`-Klasse die beiden Methoden `eintrag_loeschen` sowie `sql_loeschen` hinzu, die die beiden Löschvorgänge in der Tabelle bzw. in der SQLite-Datenbank realisieren. Die Methode `eintrag_loeschen` ruft dabei die Methode `sql_loeschen` auf und übergibt ihr den Wert von `isbn_nr`.

```
1 def eintrag_loeschen(self):
2     zeile = self.tabellen2.currentRow()
3     isbn_nr = self.tabellen2.item(zeile, 4).text()
4     self.tabellen2.removeRow(self.tabellen2.currentRow())
5     self.anzahl.setText("Anzahl der Einträge: {}".format(self.tabellen2.
6         rowCount()))
7     self.sql_loeschen(isbn_nr)
8
9 def sql_loeschen(self, isbn_nr):
10    verbindung = sqlite3.connect("daten2.db")
11    cursor = verbindung.cursor()
12    cursor.execute("""DELETE FROM bibliotheksbestand WHERE isbn=?""", (isbn_
13        _nr, ))
14    verbindung.commit()
15    cursor.execute("""SELECT * FROM bibliotheksbestand""")
16    for zeile in cursor:
17        print(zeile)
18    verbindung.close()
```

Das Gesamtbestand-Fenster hat nun das folgende Aussehen:

## 15.7 Übung: Programme mit Fenstern selbst gestalten

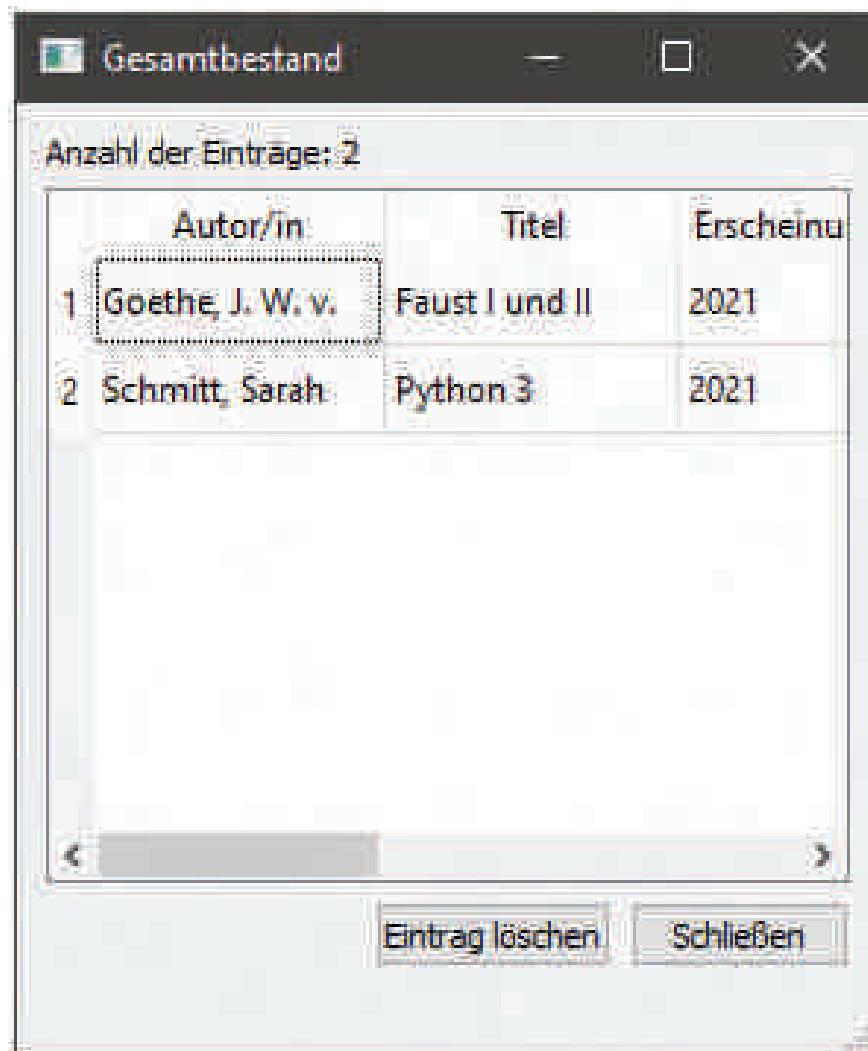


Abb. 15.23 Das neue Gesamtbestand-Fenster mit einer zusätzlichen Löschen-Option

2. a) Wir verwenden für alle Layouts des Taschenrechner-Fensters die Rasteranordnung. Der Interaktionsbereich mit dem Line Edit und den Buttons befindet sich innerhalb einer Group Box, die wiederum in einem Frame-Widget enthalten ist. Das Line-Edit-Widget erhält den Objektnamen `eingabe`, während wir die Buttons der Zahlen 0-9 nach dem Schema `button_null`, `button_eins`, etc. benennen. Als Objektnamen für die übrigen Buttons wählen wir die Bezeichnungen `plus`, `minus`, `multipliziert`, `geteilt`, `gleich` und `neu`. Das erstellte Fenster sollte nun das folgende Aussehen haben:

## 15 Grafische Benutzeroberflächen mit PyQt erstellen



**Abb. 15.24** Die Ansicht des Taschenrechners

b) Wir versehen jeden der 16 Buttons mit einem `clicked`-Signal, das jeweils auf eine eigene Methode verweist. Im Falle der Eingabebuttons (**0-9**, **+**, **-**, **\*** und **/**) wird beim Klicken darauf die entsprechende Zahl oder der Operator als String zum bereits im Eingabefeld vorhandenen Text hinzugefügt. Wird auf das Gleichheitszeichen geklickt, löst die Methode `gleich_geklickt` eine Ausnahmebehandlung aus, die den String im Texteingabefeld mittels eines `eval`-Befehls (Kapitel 5.3.) auswertet und im Falle einer gültigen Eingabe das Rechenergebnis anzeigt. Andernfalls wird eine der beiden möglichen Ausnahmen ausgelöst. Ein Klick auf den **C**-Button setzt den Inhalt des Eingabefelds wieder zurück. Direkte Eingaben in das Texteingabefeld werden durch die `setReadOnly`-Methode unterbunden.

```

1 import sys
2 from PyQt5 import QtWidgets, uic
3
4 class Taschenrechner(QtWidgets.QMainWindow):
5     def __init__(self):
6         QtWidgets.QMainWindow.__init__(self) # Alternative: super()
7         self.ui = uic.loadUi(r"C:\Users\Sarah\PycharmProjects\Kapitel 15\
```

## 15.7 Übung: Programme mit Fenstern selbst gestalten

```

8     taschenrechner.ui", self)
9         self.ui.button_eins.clicked.connect(self.eins_geklickt)
10        self.ui.button_zwei.clicked.connect(self.zwei_geklickt)
11        self.ui.button_drei.clicked.connect(self.drei_geklickt)
12        self.ui.button_vier.clicked.connect(self.vier_geklickt)
13        self.ui.button_fuenf.clicked.connect(self.fuenf_geklickt)
14        self.ui.button_sechs.clicked.connect(self.sechs_geklickt)
15        self.ui.button_sieben.clicked.connect(self.sieben_geklickt)
16        self.ui.button_acht.clicked.connect(self.acht_geklickt)
17        self.ui.button_neun.clicked.connect(self.neun_geklickt)
18        self.ui.button_null.clicked.connect(self.null_geklickt)
19        self.ui.plus.clicked.connect(self.plus_geklickt)
20        self.ui.minus.clicked.connect(self.minus_geklickt)
21        self.ui.multipliziert.clicked.connect(self.multipliziert_geklickt)
22        self.ui.geteilt.clicked.connect(self.geteilt_geklickt)
23        self.ui.gleich.clicked.connect(self.gleich_geklickt)
24        self.ui.neu.clicked.connect(self.neu_geklickt)
25        self.ui.eingabe.setReadOnly(True)
26
27    def eins_geklickt(self):
28        self.eingabe.setText(self.eingabe.text()+"1")
29
30    def zwei_geklickt(self):
31        self.eingabe.setText(self.eingabe.text()+"2")
32
33    def drei_geklickt(self):
34        self.eingabe.setText(self.eingabe.text()+"3")
35
36    def vier_geklickt(self):
37        self.eingabe.setText(self.eingabe.text()+"4")
38
39    def fuenf_geklickt(self):
40        self.eingabe.setText(self.eingabe.text()+"5")
41
42    def sechs_geklickt(self):
43        self.eingabe.setText(self.eingabe.text()+"6")
44
45    def sieben_geklickt(self):
46        self.eingabe.setText(self.eingabe.text()+"7")
47
48    def acht_geklickt(self):
49        self.eingabe.setText(self.eingabe.text()+"8")
50
51    def neun_geklickt(self):
52        self.eingabe.setText(self.eingabe.text()+"9")
53
54    def null_geklickt(self):
55        self.eingabe.setText(self.eingabe.text()+"0")
56
57    def plus_geklickt(self):
58        self.eingabe.setText(self.eingabe.text()+"+")
59
60    def minus_geklickt(self):
61        self.eingabe.setText(self.eingabe.text()+"-")
62
63    def multipliziert_geklickt(self):

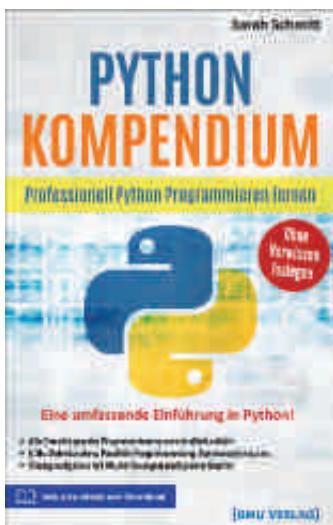
```

## 15 Grafische Benutzeroberflächen mit PyQt erstellen

```
64         self.eingabe.setText(self.eingabe.text()+"*")
65
66     def geteilt_geklickt(self):
67         self.eingabe.setText(self.eingabe.text() + "/")
68
69     def gleich_geklickt(self):
70         try:
71             x = eval(self.eingabe.text())
72             self.eingabe.setText(str(x))
73         except SyntaxError:
74             self.eingabe.setText("Syntax error")
75         except ZeroDivisionError:
76             self.eingabe.setText("Zero division error")
77
78     def neu_geklickt(self):
79         self.eingabe.clear()
80
81
82 app = QtWidgets.QApplication(sys.argv)
83 fenster = Taschenrechner()
84 fenster.show()
85 sys.exit(app.exec_())
```

Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 16

# E-Mails versenden und verwalten

Wussten Sie, dass der Begriff *Spam* auf eine aus Büchsenfleisch bestehende Frühstücksmahlzeit zurückzuführen ist, die es seit den 1930er Jahren gibt? Auch Monty Python thematisierten in den 70er Jahren die wenig gehaltvolle Nahrung in einem eigenen Sketch, in dem die Kellnerin eines Restaurants mit penetranter Stimme „Spam“ als einen sich wiederholenden Bestandteil in nahezu allen Menüvariationen anpreist:  
<https://www.dailymotion.com/video/x2hwqlw>

Der Monty-Python-Sketch führte später wiederum zur Bezeichnung der uns allen bekannten unerwünschten, massenhaft versendeten elektronischen Nachrichten, die Schätzungen zufolge über die Hälfte aller versandten E-Mails ausmachen.

Doch trotz Spamming sind E-Mails nach wie vor ein sehr wichtiger Bestandteil unserer digitalen Kommunikation. So wurden im Jahr 2020 von über 4 Milliarden Nutzerrinnen und Nutzern weltweit jeden Tag etwa 306 Milliarden E-Mails versandt – der Trend ist steigend.

Wenig überraschend, dass Python eine eigene Funktionalität bietet, mithilfe derer sich aus dem Code heraus E-Mails versenden lassen. Dies kann in vielen Bereichen sehr nützlich sein – beispielsweise, wenn Sie einen Newsletter automatisiert an viele Empfänger gleichzeitig versenden möchten, ohne dies händisch für einzelne Empfängeradressen und -namen vorzunehmen, oder wenn auf eine bestimmte Nutzeraktion hin eine E-Mailbenachrichtigung erfolgen soll.

In diesem Kapitel werden wir Ihnen die Funktionsweise von E-Mails erklären und zeigen, wie man mit Python E-Mails versendet und empfängt.

### 16.1 Die Funktionsweise von E-Mails

Während sich das Versenden einer E-Mail meist sehr einfach gestaltet, ist die Technologie, die sich gleichzeitig im Hintergrund abspielt, weitaus komplizierter. Im Folgenden wollen wir uns auf die praktischen Aspekte der E-Mail-Funktionalität konzentrieren und daher in diesem Abschnitt auf das hierfür nötige Hintergrundwissen eingehen.

Wie ein physischer Brief, der per Post versendet wird, besteht auch eine E-Mailnachricht aus verschiedenen Komponenten, die abgesehen vom tatsächlichen Inhalt der

## 16.2 Eine E-Mailnachricht versenden

Nachricht für das Versenden derselben notwendig sind. Dazu gehören beispielsweise der Betreff der E-Mail, Informationen zu Sender und Empfänger, Zeitangaben, etc.

Das *Simple Mail Transfer Protocol* (SMTP) ist ein Kommunikationsprotokoll, das die Übertragung – insbesondere den Versand – von elektronischen Nachrichten in Computernetzwerken über einen Server regelt. Jeder größere E-Mailprovider hat einen eigenen SMTP-Server. Für Outlook- oder Hotmail-Emailadressen ist dies beispielsweise *smtp-mail.outlook.com* bzw. *smtp.live.com*, für Gmail *smtp.gmail.com* und für GMX *mail.gmx.net*.

Wenn Sie in Ihrem E-Mail-Programm eine Nachricht verfasst haben und auf „Senden“ klicken, wird das Programm zunächst den SMTP-Server kontaktieren und die Informationen der Nachricht an diesen übermitteln. Die Verbindung zum SMTP-Server wird dabei über einen sogenannten Port aufgebaut. Während ursprünglich Port 25 diesem Verbindungsaufbau diente, verwenden neuere Server hingegen in der Regel Port 587, was eine sicherere Verschlüsselung ermöglicht und die Anzahl unerwünschter Spam-Nachrichten reduziert.

Ihr SMTP-Server liest die zum Versenden nötigen Informationen der Nachricht ein und leitet sie sodann – in der Regel blitzschnell – über das Internet an den SMTP-Server des Empfängers weiter. Dieser empfängt die Informationen wiederum über dessen Port, überprüft die Nachricht auf Spam und stellt sie an den Empfänger zu.

Auf der Empfängerseite kommen beim Abrufen der E-Mail wiederum zwei andere Server – POP3 oder IMAP4 – zum Einsatz, für die es in Python's Standardbibliothek dementsprechend die Module `poplib` und `imaplib` gibt. Hiervon werden wir das Modul `imaplib` in 16.3. genauer untersuchen.

### 16.2 Eine E-Mailnachricht versenden

Python's Standardbibliothek verfügt zum Versenden von E-Mailnachrichten über ein passendes eingebautes Modul namens `smtplib`, das alle hierfür benötigten Funktionen bereithält. Mithilfe dieses Moduls instanziiieren wir ein Objekt der Klasse `SMTP`, über die die Kommunikation mit dem Server läuft:

#### Codebeispiel #1

```
1 import smtplib
2
3 s = smtplib.SMTP("smtp.strato.de", 587)
4 s.starttls() # nicht benötigt für Port 465
```

## 16 E-Mails versenden und verwalten

Der String in der Klammer der dritten Zeile enthält dabei die optionale Angabe des SMTP-Servers. Die Zahl in der Klammer ist die im vorigen Abschnitt erwähnte Portnummer, in der Regel 587. Diese Angabe ist ebenfalls optional – wird nichts angegeben, wählt Python standardmäßig Port 25. Sie sollten die beiden Angaben zu Host und Port an dieser Stelle jedoch machen, da mit dem Befehl sonst keine Verbindung zum Server hergestellt werden kann und Sie den Verbindungsauftbau anschließend durch zusätzlichen Code ausdrücken müssten.

In der vierten Zeile sorgt die `starttls`-Methode für eine zusätzliche, sicherere TLS-Verschlüsselung, die im Falle von Port 25 und Port 587 zum Einsatz kommt. Sollte Ihr Server den selteneren Port 465 verwenden, können Sie diese Codezeile weglassen. In diesem Fall wird die SSL-Verschlüsselung benötigt, für die zum Verbindungsauftbau mit dem Server stattdessen der Befehl `smtplib.SMTP_SSL()` gilt.

Als Nächstes fragen wir durch den Code das Passwort ab und benutzen es in der da-rauffolgenden Zeile zum Anmelden beim SMTP-Server. Nutzen Sie ein Gmail-Konto, müssen Sie in Ihren Google-Kontoeinstellungen zusätzlich unsicheren Apps den Zu-griff auf die E-Mails ermöglichen.

```
1 passwort = input("Passwort: ")
2 s.login("schmitt@bmu-verlag.de", passwort)
```

Sie können das Passwort zwar auch direkt als String in die Klammer der `login`-Me-thode schreiben, anstatt es durch einen `input`-Befehl abzufragen. Es ist jedoch nicht ratsam, Ihr Passwort in Python-Code abzuspeichern, da sonst auch andere Personen, die Zugriff auf Ihren Computer bzw. den Code haben, dieses einsehen können. Ver-zichten Sie also auf den `input`-Befehl nur dann, wenn Sie keinerlei Sicherheitsbeden-ken für den Zugang zu Ihrem Code haben.

Nachdem die Anmeldung beim Server über die E-Mailadresse und das Passwort er-folgt ist, geben wir die für das Versenden der Nachricht benötigten Daten ein. Dazu gehören Angaben zum Absender, Empfänger, Betreff und Textinhalt, die wir sodann (bis auf die Empfängeradresse) in der Variable `nachricht` zusammenfassen:

```
1 absender_email = "Sarah Schmitt <schmitt@bmu-verlag.de>"
2 empfaenger_email = "schmitt@bmu-verlag.de"
3 betreff = "Wichtige Nachricht"
4 text = "Dies ist kein Spam."
5 nachricht = "From: {}\nSubject: {} \n\n{}".format(absender_email, betreff,
6 text)
```

E-Mailadressen müssen immer im String-Format stehen und lassen sich wie in den ersten beiden Zeilen gezeigt entweder in eckigen Klammern und mit vorangestelltem Namensfeld (1. Zeile) oder als reine Zeichenkette (2. Zeile) schreiben. Geben Sie als Ab-senderadresse nur eine E-Mailadresse ohne Namen an, so wird beim Abrufen der Mail

## 16.2 Eine E-Mailnachricht versenden

durch den Empfänger nur diese Adresse ohne den Namen des Absenders angezeigt. Im obigen Beispiel sind Absender- und Empfängeradresse identisch – wir schicken also eine Mail an uns selbst. Beim Testen der Funktionalität ist diese Vorgehensweise ratsam, um unnötiges Spammen fremder Mailboxen zu vermeiden.

Die anschließende Angabe des Betreffs (engl. *subject*) ist optional. Die Variable `text` in der nächsten Zeile enthält den eigentlichen Inhalt der Mail als String. In der `nachricht`-Variablen werden die drei Angaben zu Absender, Betreff und Text der Nachricht in einem String abgespeichert. Dabei ist es wichtig, das Format genau einzuhalten und die Leerzeilen richtig zu setzen. Nur so erkennt Python, bei welchem Teil des Strings es sich um Absender und Betreff handelt, und wo die Betreffzeile der Nachricht endet und der Textinhalt beginnt.

Nun können Sie die Nachricht mit der `sendmail`-Methode verschicken. Sie enthält die Angaben zu Absender und Empfänger sowie den Textinhalt der Nachricht.

```
1 s.sendmail(absender_email, empfaenger_email, nachricht)
2 s.quit()
```

Der `quit`-Befehl beendet die Sitzung und trennt die Verbindung zum Server, nachdem die Mail verschickt wurde. Sie sollte bereits in Ihrer Inbox angekommen sein. Haben Sie keine E-Mailnachricht erhalten, so hat möglicherweise eine Firewall oder ein Antivirusprogramm den Python-Code am Versenden der E-Mail gehindert.

Ein vollständiger Code zum Versenden von E-Mails sollte sicherheitshalber eine Fehlerbehandlung mit `try ... except` für die Verbindungsanfrage zum Server, die Passworteingabe und das Login beinhalten, um eventuelle falsche Angaben aufzufangen.

Mit der obigen Methode können Sie lediglich E-Mails in *Plain Text* versenden. Das bedeutet, dass der Inhalt der Nachricht beispielsweise keine Sonderzeichen wie Umlaute enthalten darf und sich nicht formatieren lässt. Auch Anhänge lassen sich mit dem einfachen `smtplib`-Modul nicht versenden. Für diese erweiterte Funktionalität kommt das `email`-Modul zum Einsatz, das wir Ihnen im nächsten Abschnitt vorstellen werden.

### 16.2.1 Erweiterte Funktionen mit dem Modul `email`

Um E-Mailnachrichten im *HTML*-Format oder mit Dateianhängen verschicken zu können, müssen Sie das `email`-Modul importieren. Dieses Modul verwendet den so genannten *MIME-Standard*, der die genannten Funktionalitäten durch eine erweiterte Codierung ermöglicht. MIME steht für *Multipurpose Internet Mail Extensions*.

## 16 E-Mails versenden und verwalten

Der Funktionsumfang des `email`-Moduls ist sehr groß, sodass wir hier nur eine Auswahl aller möglichen Anwendungsoptionen vorstellen können. Die gesamte Dokumentation finden Sie unter <https://docs.python.org/3/library/email.html>.

Als Erstes werden Sie lernen, wie man eine mit HTML formatierte E-Mail mithilfe der Klasse `EmailMessage` aus `email.message` erstellt. Zum tatsächlichen Versenden der Nachricht kommt dabei weiterhin das `smtplib`-Modul zum Einsatz. Nachdem wir die beiden benötigten Komponenten durch

### Codebeispiel #2

```
1 from email.message import EmailMessage
2 import smtplib
```

importiert haben, erzeugen wir mit `nachricht` eine Instanz von `EmailMessage` und geben ähnlich wie bei einem Dictionary die jeweiligen Werte für Betreff, Absender und Empfänger nach dem folgenden Schema an:

```
1 nachricht = EmailMessage()
2 nachricht["Subject"] = "Neue Nachricht"
3 nachricht["From"] = "Sarah Schmitt <schmitt@bmu-verlag.de>"
4 nachricht["To"] = "schmitt@bmu-verlag.de"
```

Anschließend definieren wir die Variable `htmltext`, die den Textinhalt unserer Nachricht im HTML-Format enthält:

```
1 htmltext = """\
2 <html>
3   <body>
4     <h2>Liebe Lesende,</h2>
5     <p>Glückwunsch, Sie haben bereits große Fortschritte gemacht!<br>
6       <strong>Erinnerung:</strong> Dieses Programmierbeispiel können
7 Sie auf der Webseite des
8   <a href="http://bmu-verlag.de/">BMU Verlags</a>
9     als PDF herunterladen.
10  </p>
11  <p>Viele Grüße<br>
12    <font face="verdana" color="green">Sarah Schmitt</font>
13  </p>
14  </body>
15 </html>
16 """
```

Die Start- und Endtags mit eckigen Klammern zeichnen dabei besondere HTML-Formatierungen aus. Beispielsweise kennzeichnen die Tags `<h2>...</h2>` einen Überschrifttyp, `<p>...</p>` einen Abschnitt und `<br>` eine neue Zeile. Auch Links zu Webseiten, unterschiedliche Schrifttypen oder -farben lassen sich angeben.

## 16.2 Eine E-Mailnachricht versenden

Der Befehl

```
1  nachricht.set_content(htmltext, "html")
```

fügt unserer Nachricht den Textinhalt `htmltext` hinzu. Das zweite Argument "`html`" in der Klammer gibt an, dass es sich dabei um HTML-Text handelt. Nun können wir die E-Mail nach dem uns bekannten Schema versenden. Anstelle von `sendmail` lässt sich hier auch die einfachere `send_message`-Methode verwenden, die lediglich ein Argument, `nachricht`, benötigt:

```
1  s = smtplib.SMTP("smtp.strato.de", 587)
2  s.starttls()
3  passwort = input("Passwort: ")
4  s.login("schmitt@bmu-verlag.de", passwort)
5  s.send_message(nachricht)
6  s.quit()
```

Wenn Sie die Angaben im Code Ihrem E-Mailkonto entsprechend anpassen, sollte sich nach dem Ausführen des Codes in Ihrem Postfach die E-Mail mit der HTML-Formatierung befinden.



**Abb. 16.1** Die Anzeige der HTML-Nachricht

Nun wollen wir betrachten, wie sich mithilfe des `email`-Moduls E-Mailnachrichten mit Anhängen versenden lassen. Dies realisiert beispielsweise der folgende Code, in dem wir unserer Nachricht als Anhang ein Foto hinzufügen:

*Codebeispiel #3*

```
1  import smtplib
2  from email.message import EmailMessage
3
4  nachricht = EmailMessage()
5  nachricht["Subject"] = "Neue Nachricht"
6  nachricht["From"] = "Sarah Schmitt <schmitt@bmu-verlag.de>"
```

## 16 E-Mails versenden und verwalten

```

7  nachricht["To"] = "schmitt@bmu-verlag.de"
8  nachricht.set_content("siehe Anhang")
9
10 with open("foto.jpg", "rb") as datei:
11     anhang = datei.read()
12     nachricht.add_attachment(anhang, maintype="image", subtype="jpg",
13     filename="foto.jpg")
14
15 s = smtplib.SMTP("smtp.strato.de", 587)
16 s.starttls()
17 passwort = input("Passwort: ")
18 s.login("schmitt@bmu-verlag.de", passwort)
19 s.send_message(nachricht)
20 s.quit()

```

Falls die Datei, die Sie Ihrer Nachricht anhängen möchten, nicht im selben Ordner abgespeichert ist wie Ihr Python-Code, müssen Sie den vollständigen Dateipfad angeben.

Die Befehle `open` und `read` zum Öffnen und Auslesen von Dateien sollten Ihnen aus Kapitel 13 bereits bekannt sein. Die Methode `add_attachment` fügt den Anhang hinzu. Die beiden Parameter `maintype` und `subtype` müssen je nach Dateityp angepasst werden. Möchten Sie beispielsweise eine `.pdf`-Datei anhängen, so müssen Sie deren Werte in `"application"` bzw. `"pdf"` umändern.

### 16.3 Empfangene E-Mailnachrichten mit `imaplib` bearbeiten

Genauso wie Python das Versenden von E-Mails aus dem Code heraus ermöglicht, existiert umgekehrt eine Funktionalität zum Abrufen empfangener E-Mails. Hierfür gibt es die eingebauten Bibliotheken `imaplib` und `poplib`, die eine Verbindung zu einem IMAP4- bzw. POP3-Server herstellen. Obwohl viele E-Mail-Provider beide Protokollvarianten anbieten, kommt das ursprünglich zuerst entwickelte POP3-Netzwerkprotokoll (Abk. für *Post Office Protocol 3*) mittlerweile allmählich außer Gebrauch. Das IMAP4-Netzwerkprotokoll (*Internet Message Access Protocol 4*) bietet hier eine erweiterte Funktionalität und oftmals bessere Gebrauchsqualität. Mit IMAP4 lassen sich E-Mails herunterladen, lesen und verwalten, während POP3 ausschließlich den Nachrichten-Download ohne Synchronisierung zwischen unterschiedlichen Geräten mit E-Mail-Kontozugang ermöglicht.

In diesem Abschnitt werden wir uns mit Abrufoptionen für E-Mailnachrichten durch das Modul `imaplib` befassen.

Wie beim Versenden von E-Mails mit `smtplib` müssen wir auch in diesem Fall nach dem Importieren des Moduls durch `import imaplib` zunächst ein Objekt der Klasse `IMAP` instanziiieren:

## 16.3 Empfangene E-Mailnachrichten mit imaplib bearbeiten

### Codebeispiel #4

```

1 import imaplib
2
3 M = imaplib.IMAP4("imap.strato.de")
4 M.starttls()

```

Der IMAP-Standardport 143 wird automatisch ausgewählt und muss nicht explizit angegeben werden. Verwendet Ihr IMAP-Server das SSL-Verschlüsselungsprotokoll, muss als zweites Argument in der Klammer die Portnummer 993 stehen.

Es folgen die für die Anmeldung benötigten Codezeilen:

```

1 passwort = input("Passwort: ")
2 M.login("schmitt@bmu-verlag.de", passwort)

```

Nun können Sie bereits erste Verwaltungsoperationen vornehmen. Jeder durch eine Methode ausgeführte Kommunikationsvorgang mit dem Server gibt dabei ein zweielementiges Tupel zurück. Das erste Element bezeichnet jeweils den Status der Operation ("OK" oder "NO"), während im zweiten Element die vom Server gesendeten Daten aufgelistet sind.

Mit der Methode `list()` lassen sich beispielsweise alle vorhandenen Ordner des E-Mailkontos anzeigen:

```
1 print(M.list())
```

War der Vorgang erfolgreich, sollten Sie ein Tupel mit "OK" und den Daten zu den vorhandenen Mailboxen sehen. Die Namen der unterschiedlichen Mailboxen sind in der Daten-Liste mit doppelten Anführungsstrichen aufgeführt. Einzelne Mailboxen können Sie daraus mithilfe der `select`-Methode auswählen:

```
1 print(M.select("Inbox"))
```

Der `print`-Befehl gibt dabei aus, wie viele E-Mails sich im ausgewählten *Inbox*-Ordner befinden. Wenn wir die Ausgabe der `select`-Methode als Tupel-Variable abspeichern, können wir folgendermaßen direkt auf die Anzahl der Nachrichten im *Inbox*-Ordner zugreifen:

```

1 (typ, daten) = M.select("Inbox")
2 anzahl_nachr = int(daten[0])
3 print("Es befinden sich %d Nachrichten in der Inbox." % anzahl_nachr)

```

Anschließend können Sie die ausgewählte Mailbox anhand vorgegebener Schlüsselbegriffe, unter anderem zu Datumsangaben (BEFORE, ON und SINCE), zum Sender (FROM), Empfänger (TO) oder zum Nachrichteninhalt (SUBJECT, BODY und TEXT) durchsuchen. Das zweite Argument in der Klammer der `search`-Methode spezifi-

## 16 E-Mails versenden und verwalten

ziert die Suche. Es besteht aus einem String mit dem Schlüsselbegriff, der wiederum einen String mit dem Suchbegriff enthält. Das Ergebnis ist die ID der gefundenen Nachrichten. Mit dem Schlüsselbegriff ALL lassen sich zudem alle Nachrichten-IDs anzeigen. Zuletzt wird die Verbindung mit logout () getrennt.

```
1 print(M.search(None, 'ALL'))
2 print(M.search(None, 'FROM "Sarah Schmitt"'))
3 print(M.search(None, 'TEXT "Spam"'))
4 M.logout()
```

```
beispiel_4.py
1 passwort = input("Passwort: ")
2 M.login("schmitt@bau-verlag.de", passwort)
3
4 print(M.list())
5 print(M.select("Inbox"))
6 (typ, daten) = M.select("Inbox")
7 anzahl_nachr = int(daten[0])
8 print("Es befinden sich %d Nachrichten in der Inbox." % anzahl_nachr)
9 print(M.search(None, 'ALL'))
10 print(M.search(None, 'FROM "Sarah Schmitt"'))
11 print(M.search(None, 'TEXT "Spam"', 'FROM "Sarah Schmitt"'))
12
13 M.logout()

Process finished with exit code 0
```

**Abb. 16.2** Die Anzeige der vorhandenen Mailboxen mit Suchergebnissen

Anhand des Outputs können wir feststellen, dass es im ausgewählten *Inbox*-Ordner dreizehn Nachrichten gibt, von denen diejenigen mit ID 2, 3, 8, 9 und 10 von *Sarah Schmitt* gesendet wurden und die Nachricht mit ID 10 den gesuchten Textinhalt „Spam“ enthält.

Durch die ID-Nummer lassen sich E-Mailnachrichten mithilfe der `fetch`-Methode abrufen. Diese Methode nimmt zwei Argumente auf: Im ersten Argument lässt sich durch einen String angeben, welche IDs ausgewählt werden sollen. Hier sind unter anderem Angaben wie "1", "2:4" oder "2: \*" möglich, wobei mit letztergenannter Schreibweise alle Nachrichten-IDs ab ID 2 ausgewählt werden. Das zweite Argument steht für die Auswahl desjenigen Teils der Nachricht, der heruntergeladen werden soll. Hier sind die Werte "(BODY [HEADER])" für den Header der Nachricht, "(BODY [TEXT])" für den Textinhalt der Nachricht sowie "(RFC822)" für die gesamte Nachricht möglich.

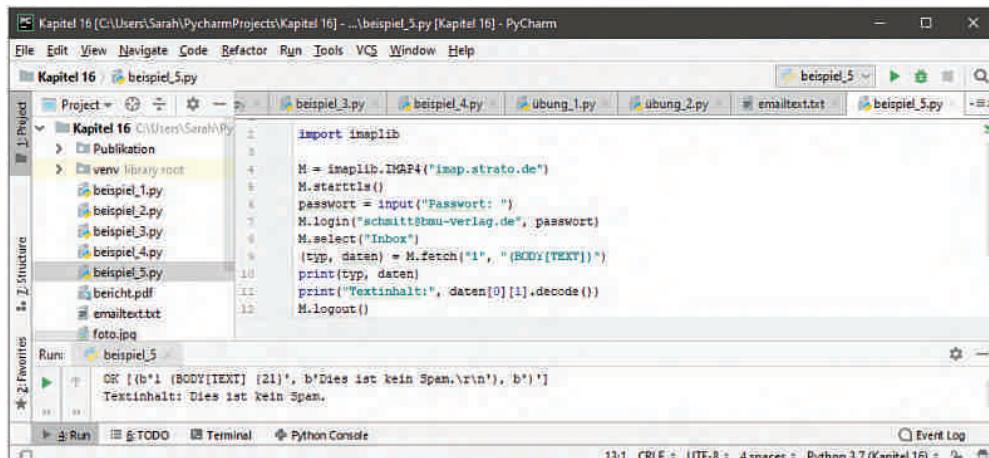
### 16.3 Empfangene E-Mailnachrichten mit imaplib bearbeiten

#### Codebeispiel #5

```

1 import imaplib
2
3 M = imaplib.IMAP4("imap.strato.de")
4 M.starttls()
5 passwort = input("Passwort: ")
6 M.login("schmitt@bmu-verlag.de", passwort)
7 M.select("Inbox")
8 (typ, daten) = M.fetch("1", "(BODY[TEXT])")
9 print(typ, daten)
10 print("Textinhalt:", daten[0][1].decode())
11 M.logout()
12

```



**Abb. 16.3** Die Ausgabe des Textinhalts mit `fetch()`

Der obige Code lädt durch `fetch()` die Daten für den Textinhalt der Nachricht mit der ID 1 herunter. Im zweiten `print`-Befehl sorgt die `decode`-Funktion dafür, dass vom Output nur der tatsächliche Textinhalt angezeigt wird.

## 16 E-Mails versenden und verwalten

### 16.4 Übung: E-Mails mit Python bearbeiten

#### Übungsaufgaben

1. Speichern Sie den HTML-Text für eine E-Mail in einer Textdatei ab. Öffnen Sie die Datei aus dem Code und versenden Sie den Textinhalt in einer E-Mail an drei verschiedene Empfänger. Begrüßen Sie diese automatisch mit deren Vornamen, ohne dies manuell für jeden einzelnen Empfänger durch drei separate E-Mail-nachrichten vorzunehmen.
2. Suchen Sie in Ihrer Inbox mit dem Schlüsselbegriff UNSEEN nach ungelesenen Nachrichten, die ein von Ihnen frei gewähltes Suchwort enthalten. Lassen Sie sich anschließend den Textinhalt dieser Nachrichten ausgeben.

**16.4 Übung: E-Mails mit Python bearbeiten****Lösungen**

1. In demselben Ordner, der auch unsere Codedatei enthält, legen wir eine Textdatei *emailtext.txt* mit folgendem HTML-Inhalt an:

```

1 <html>
2   <body>
3     <h3>Hallo {},</h3>
4     <p>du hast dich erfolgreich für den Newsletter eingetragen.
5     </p>
6     <p>Viele Grüße<br>
7       <font face="verdana" color="green">Sarah Schmitt</font>
8     </p>
9   </body>
10 </html>
```

Im Python-Code erstellen wir nach dem Import der nötigen Module zwei Listen, die jeweils die Namen und E-Mailadressen der drei Empfänger enthalten. Die E-mailadressen müssen gültig sein, damit der Code funktioniert. Anschließend lesen wir die Datei *emailtext.txt* aus und versenden in der `for`-Schleife für jeden Namen mit dazugehöriger Adresse eine E-Mail.

```

1 from email.message import EmailMessage
2 import smtplib
3
4 name = ["Valeria", "Hannelore", "Arne"]
5 email = ["idontlikesp@m.com", "sp@m.de", "sp@m.net"]
6
7 s = smtplib.SMTP("smtp.strato.de", 587)
8 s.starttls()
9 passwort = input("Passwort: ")
10 s.login("schmitt@bmu-verlag.de", passwort)
11
12 with open("emailtext.txt", "r") as textdatei:
13     emailtext = textdatei.read()
14
15 for i in range(len(name)):
16     nachricht = EmailMessage()
17     nachricht["Subject"] = "Neue Nachricht"
18     nachricht["From"] = "Sarah Schmitt <schmitt@bmu-verlag.de>"
19     nachricht["To"] = email[i]
20     nachricht.set_content(emailtext.format(name[i]), "html")
21     s.send_message(nachricht)
22
23 s.quit()
```

## 16 E-Mails versenden und verwalten



**Abb. 16.4** Die Anzeige der personalisierten E-Mailnachricht

2. Wir suchen nach ungelesenen Nachrichten, die das Wort „Fortschritte“ enthalten. Nachdem wir die Nachrichten-ID – hier ID 4 – ermittelt haben, können wir uns den Textinhalt der Nachricht ausgeben lassen, in diesem Fall im HTML-Format.

```

1 import imaplib
2
3 M = imaplib.IMAP4("imap.strato.de")
4 M.starttls()
5 passwort = input("Passwort: ")
6 M.login("schmitt@bmu-verlag.de", passwort)
7 M.select("Inbox")
8 print(M.search(None, 'UNSEEN', 'TEXT "Fortschritte"'))
9 (typ, daten) = M.fetch("4", "(BODY[TEXT])")
10 print("Textinhalt:", daten[0][1].decode())
11 M.logout()
12

```

The screenshot shows the PyCharm IDE with the project 'Kapitel 16' open. The file 'Übung\_2.py' is selected in the project tree. The code in the editor is identical to the one above. In the bottom pane, the 'Run' tab is active, showing the output of the program. The output window displays the personalized email message:

```

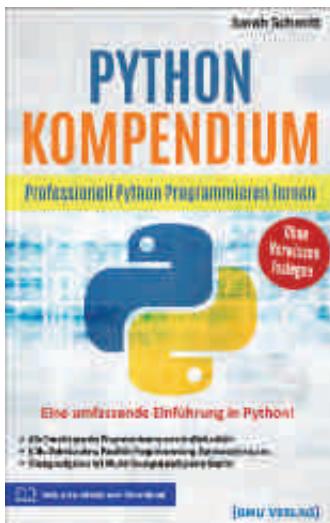
<h2>Liebe Lesende,</h2>
<p>Glückwunsch, Sie haben bereits große Fortschritte gemacht!<br>
<strong>Erinnerung:</strong> Dieses Programmierbeispiel können
Sie auf der Webseite des
<a href="http://bmu-verlag.de/">BMU Verlags</a>
als PDF herunterladen.
</p>
<p>Viele Grüße<br>
<font face="verdana" color="green">Sarah Schmitt</font>

```

**Abb. 16.5** Neue Nachrichten suchen und deren Textinhalt ausgeben

## Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 17

# Webseiten mit Django erstellen: Ein Einstieg

Erinnern Sie sich an die Anfänge des World Wide Web in den 90er Jahren? Damals war die Interneteinwahl über Modem ein längerer, von bizarren Lauten begleiteter Prozess, der über die Telefonleitung lief. War die Internetverbindung endlich hergestellt, so gab es im Netz nicht viel mehr zu tun als elektronische Postkarten zu versenden, mit Yahoo unbeholfen durch das spärliche Internet zu surfen, einzelne mp3-Musikstücke mithilfe von Napster herunterzuladen – solange dies legal war – oder Spiele in niedriger Auflösung zu spielen.

Inzwischen hat sich das Internet grundlegend verändert. Binnen weniger Jahrzehnte hat es sich zu einem globalen, dynamischen Austauschnetzwerk für kommerzielle, politische, soziale, Bildungs- und Unterhaltungszwecke entwickelt, das aus unserem Alltag nicht mehr wegzudenken ist. Der fortwährende Einzug neuer Webanwendungen, Webseiten und Internettechnologien in unser Leben ist inzwischen zum gewohnten Standard geworden. Die Möglichkeiten des Web – wie sinnvoll sie mitunter auch sein mögen – sind dabei noch immer nicht ausgeschöpft.

Umso vorteilhafter ist es, sich mit Webtechnologien und insbesondere dem Erstellen von Webseiten auszukennen – sei es für private oder professionelle Zwecke. In diesem Zusammenhang ist Python ein für die Webentwicklung sehr geeignetes und mächtiges Werkzeug, das sich in Verbindung mit einer Vielzahl unterschiedlicher Frameworks zum Erstellen eigener Webseiten einsetzen lässt. Sie können sich ein Web-Framework dabei wie eine Art Grundgerüst vorstellen, das die nötigen Strukturen für die Entwicklung individueller Anwendungen bereithält.

Unter den Python-Frameworks ist *Django* ein sehr beliebtes, quelloffenes Webframework, mithilfe dessen sich professionelle, datenbankgestützte Webseiten kreieren lassen. Es wurde erstmals im Jahr 2005 unter der BSD-Lizenz veröffentlicht und wird heute unter anderem für die Webseiten von Instagram, Mozilla und der NASA eingesetzt. Es ist nach dem Jazzmusiker Django Reinhardt benannt.

Ein weiteres, derzeit gerne eingesetztes Webframework ist *Flask* (<https://flask.palletsprojects.com>), das aufgrund seiner minimalistischen, flexibleren Funktionalität insbesondere für die schnelle Entwicklung von weniger komplexen Webseiten geeignet ist.

Sobald jedoch Datenbanken und größere, komplexere Webseiten mit hohem Nutzeraufkommen ins Spiel kommen, ist Django die bessere und effizientere Wahl. Das weit verbreitete Django-Framework hält zahlreiche eingebaute Pakete für unterschiedli-

## 17.1 Django installieren und ein neues Projekt starten

che Bedürfnisse bereit und wird von einer großen und lebendigen Community unterstützt, die sich aktiv an dessen Weiterentwicklung beteiligt. Derzeit lässt sich Django in Verbindung mit den Datenbankmanagementsystemen PostgreSQL, MariaDB, MySQL, Oracle und SQLite verwenden.

In diesem Kapitel werden wir Ihnen eine erste Einführung in die Webentwicklung mit Django geben, ohne dabei hinsichtlich der technischen und theoretischen Hintergründe zu sehr ins Detail zu gehen. Vielmehr will Ihnen dieses Kapitel einen nützlichen, praktischen Einblick in die umfassenden Webgestaltungsmöglichkeiten mit Django verschaffen, sodass Sie schnell darauf aufbauen und mit Leichtigkeit eine eigene Webseite erstellen können.

### 17.1 Django installieren und ein neues Projekt starten

Je nach Komplexität und Einsatzdauer Ihres Projekts ist es meist eine gute Idee, im Vorfeld der Arbeit an einem neuen Django-Projekt ein sogenanntes *Virtual Environment* (zu Deutsch: *virtuelle Umgebung*) anzulegen. Dadurch wird sichergestellt, dass die für Ihre Webseiten-Projekte benötigten externen Pakete oder Bibliotheken – man spricht in diesem Zusammenhang auch von *Abhängigkeiten* – auch bei unterschiedlichen Python- oder Paket-Versionen sowie Änderungen daran davon unabhängig und funktional bleiben. Die externen Pakete bzw. Bibliotheken für ein spezifisches Projekt bzw. eine Anwendung werden dabei nicht für das gesamte Betriebssystem installiert, sondern liegen separat in einem eigenen, isolierten Verzeichnis in einer virtuellen Umgebung vor. Dies hat den weiteren Vorteil, dass sich Anwendungen, die verschiedene Versionen eines Pakets bzw. einer Bibliothek benötigen, jeweils separat in unterschiedlichen dafür vorgesehenen virtuellen Umgebungen verwenden lassen. Da diese virtuellen Umgebungen untereinander keine Pakete oder Bibliotheken teilen, sind bei versehentlichen Aktualisierungen der Pakete Konflikte ausgeschlossen. Auch Anwendungen, die mit verschiedenen Python-Versionen laufen müssen, lassen sich somit problemlos parallel auf einem PC verwenden. Nicht zuletzt lassen sich Anwendungen mithilfe einer virtuellen Umgebung sehr einfach von einem Rechner auf einen anderen übertragen.

Virtuelle Umgebungen machen daher vor allem Sinn bei größeren Projekten, die über einen längeren Zeitraum hinweg zum Einsatz kommen und Änderungen hinsichtlich ihrer Versionen oder Bibliotheken unterliegen können.

Während Python zuvor unterschiedliche Module für virtuelle Umgebungen anbot, wird seit Python-Version 3.5 das in der Standardbibliothek von Python 3 enthaltene Modul `venv` empfohlen, das wir auch in diesem Buch verwenden. Mehr zu virtuellen Umgebungen, etwa auch zu virtuellen Umgebungen aus externen Paketen, finden Sie beispielsweise auf den Webseiten der Python- oder Django-Dokumentationen.

## 17 Webseiten mit Django erstellen: Ein Einstieg

Zum Erstellen einer neuen virtuellen Umgebung navigieren Sie in der Kommandozeile (bzw. im Terminal-Fenster für macOS und Linux) zunächst durch die Angabe von `cd` (engl. für *change directory*) und dem gewünschten Dateipfad in dasselbe Verzeichnis, in dem Sie ebenfalls den Ordner für Ihr neues Webseiten-Projekt in Ihrer IDE anlegen möchten, hier beispielsweise:

```
1 cd C:\Users\Sarah\PycharmProjects\
```

Sie können die virtuelle Umgebung zwar ebenfalls an anderer Stelle in Ihrem Rechner speichern, vorzugsweise sollte sie sich jedoch in der Ordnerstruktur *neben* Ihrem geplanten Webseiten-Projekt befinden.

Die virtuelle Umgebung wird anschließend in einem neuen Ordner durch

```
1 python -m venv myvenv
```

angelegt. Der `python`-Befehl muss dabei durch `python3` ersetzt werden, wenn Sie als Betriebssystem macOS oder Linux verwenden. Bei `myvenv` handelt es sich um einen frei wählbaren Ordnernamen für die virtuelle Umgebung. Weitere gängige Bezeichnungen sind etwa `myenv`, `env` oder `venv`.

Die virtuelle Umgebung muss stets vor der Arbeit an dem jeweiligen Projekt aktiviert werden. Zum Aktivieren der virtuellen Umgebung geben Sie in Windows

```
1 myvenv\Scripts\activate.bat
```

bzw. auf Linux- oder macOS-Betriebssystemen

```
1 source myvenv/bin/activate
```

ein. Wie Sie sehen, ist der Kommandozeile nun das Präfix (`myvenv`) vorangestellt. Zum Deaktivieren der virtuellen Umgebung genügt der Befehl `deactivate`. Zur erneuten Aktivierung müssen Sie nur die obige Aktivierungs-Eingabe wiederholen. Sollten Sie das Kommandozeilenfenster zwischenzeitlich geschlossen haben, müssen Sie beim erneuten Aktivieren jedoch darauf achten, dass Sie sich im richtigen Verzeichnis befinden.

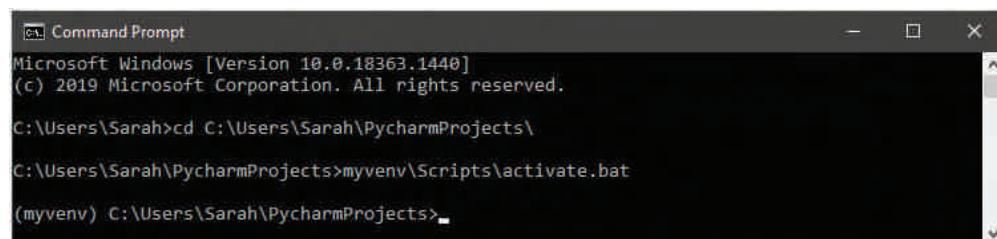


Abb. 17.1 Erstellen einer virtuellen Umgebung

## 17.1 Django installieren und ein neues Projekt starten

Solange die virtuelle Umgebung aktiviert ist, werden Pakete automatisch darin installiert. Als Nächstes installieren wir in unserer virtuellen Umgebung also das externe Django-Framework, um es später durch `import django` in Python importieren zu können. Die Installation ist entweder über die offizielle Webseite des Django-Projekts, [www.djangoproject.com](http://www.djangoproject.com), oder über den Python-Paketmanager `pip` möglich.

Obwohl die Django-Webseite sehr nützlich für zusätzliche Informationen, Tutorials und den Community-Austausch über Django ist, empfehlen wir Ihnen auch hier die Installation über `pip`.

Hierzu geben wir in unserem Kommandozeilenfenster

```
1 pip3 install django
```

ein. Wenn alles richtig funktioniert hat, sollten Sie in der letzten Zeile des Installationsvorgangs eine Meldung über die erfolgreiche Installation samt die Django-Version sehen:

```
(myvenv) C:\Users\Sarah\PycharmProjects>pip3 install django
Collecting django
  Downloading Django-3.1.7-py3-none-any.whl (7.8 MB)
    |████████████████████| 7.8 MB 598 kB/s
Requirement already satisfied: pytz in c:\users\sarah\pycharmprojects\myvenv\lib\site-packages
(from django) (2020.1)
Requirement already satisfied: asgiref<4,>=3.2.10 in c:\users\sarah\pycharmprojects\myvenv\lib\site-packages
(from django) (3.2.10)
Requirement already satisfied: sqlparse>=0.2.2 in c:\users\sarah\pycharmprojects\myvenv\lib\site-packages
(from django) (0.3.1)
Installing collected packages: django
Successfully installed django-3.1.7
```

**Abb. 17.2** Die Installation des Django-Frameworks über `pip`

Wir verwenden in diesem Kapitel die Django-Version 3.1.7. Das Django-Paket wird im `site-packages`-Ordner innerhalb Ihrer virtuellen Umgebung installiert. Damit Sie dieses Paket in Ihrer IDE verwenden können, müssen Sie den Pfad dort angeben. Den Installationspfad finden Sie wie in Kapitel 15.1. beschrieben durch die Eingabe von

```
1 pip3 show django
```

im Kommandozeilenfenster, woraufhin die Informationen zur installierten Django-Bibliothek angezeigt werden. Unter `Location` können Sie den Installationspfad einsehen und kopieren. In unserem Fall ist dies beispielsweise

`c:\users\sarah\pycharmprojects\myvenv\lib\site-packages`

Notieren Sie sich diesen Pfad, da Sie ihn in diesem Teilkapitel nach dem Anlegen des Webseiten-Projekts noch benötigen werden.

## 17 Webseiten mit Django erstellen: Ein Einstieg

Durch die Angabe von `cd` (engl. für *change directory*) und dem gewünschten Dateipfad navigieren Sie anschließend in der Kommandozeile zu dem Ort, an dem Sie die Dateien für Ihre Webseite abspeichern möchten. Dies sollte derselbe Ordner sein, in dem Sie üblicherweise Ihre Python-Programmdateien abspeichern. In unserem Fall ist das der Ordner für unsere PyCharm-Projekte:

```
1 cd C:\Users\Sarah\PycharmProjects\
```

Zum Anlegen eines neuen Projekts kommt der Befehl `django-admin` zum Einsatz. Hiermit lassen sich viele verschiedene Aktionen für Webanwendungen ohne externen Input ausführen. Mit dem Kommandozeilenbefehl

```
1 django-admin startproject projektnname
```

legen Sie nun beispielsweise einen Ordner für ein neues Projekt an. Als Projektnamen und gleichnamigen -ordner wählen wir hier `erstewebsite`. Nachdem Sie **Enter** gedrückt haben, sollte sich das neue Webseiten-Projekt im gewünschten Verzeichnis befinden.

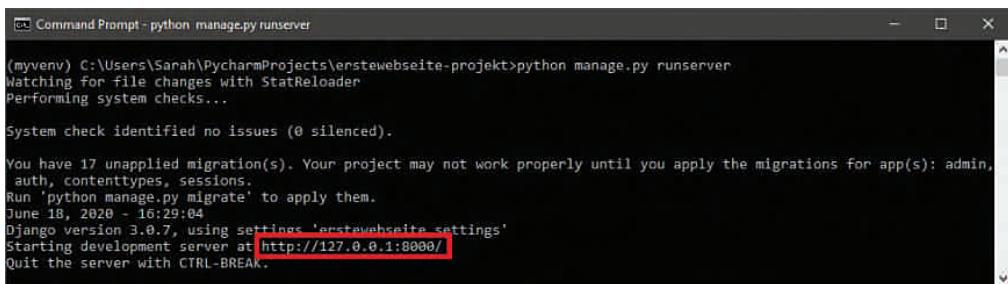
Mit `django-admin` erzeugte Projekte enthalten bereits alle für die Erstellung einer Webseite benötigten Komponenten. Im Ordner `erstewebsite` befinden sich neben einem weiteren `erstewebsite`-Unterordner insbesondere eine Datei namens `manage.py`, mithilfe derer durch Unterbefehle die wichtigsten administrativen Vorgänge für unseren Code gehandhabt und neue Webseiten gestartet werden können. Um Verwechslungen zu vermeiden, können Sie unterschiedliche Bezeichnungen für die beiden `erstewebsite`-Ordner wählen. Wir wählen hier den Namen `erstewebsite-projekt` für den übergeordneten Ordner. Für die Umbenennung müssen Sie eventuell Ihr Kommandozeilenfenster schließen und anschließend erneut öffnen. Dabei dürfen Sie nicht vergessen, die virtuelle Umgebung erneut zu aktivieren.

Machen Sie im Kommandozeilenfenster nun folgende Eingabe und drücken Sie danach **Enter**:

```
1 cd C:\Users\Sarah\PycharmProjects\erstewebsite-projekt
2 python manage.py runserver
```

Der `runserver`-Befehl des Verwaltungsprogramms `manage.py` startet auf unserem Computer einen kleinen Webserver, der unser Django-Projekt hostet. Die Serveradresse ist in der vorletzten Zeile des Fensters angezeigt:

## 17.1 Django installieren und ein neues Projekt starten



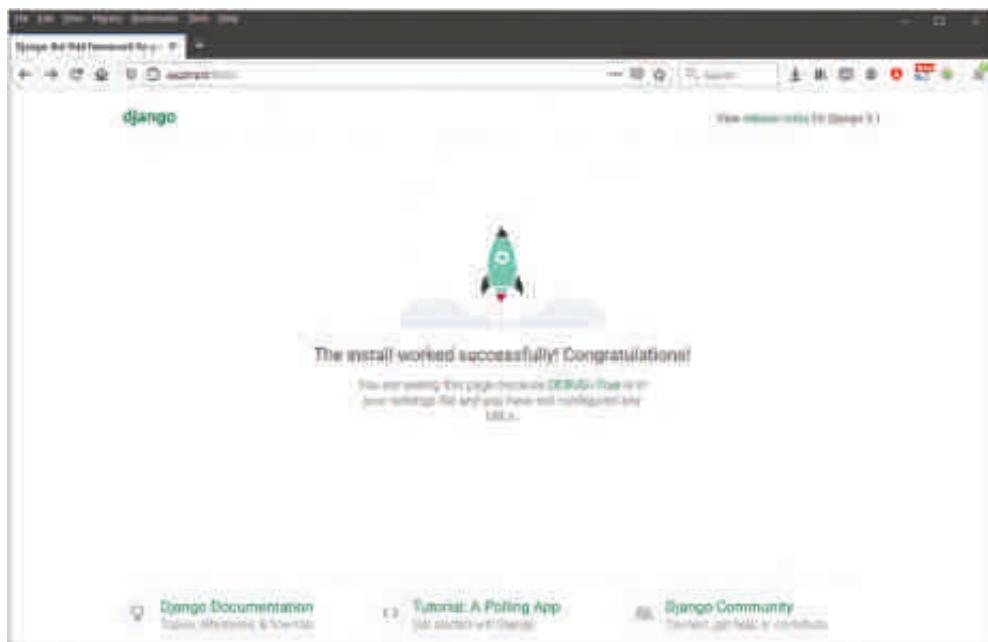
```
(myenv) C:\Users\Sarah\PycharmProjects\erstewebsite-projekt>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin,
auth, contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
June 18, 2020 - 16:29:04
Django version 3.0.7, using settings 'erstewebsite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

**Abb. 17.3** Den Webserver für unsere erste Webseite starten

Kopieren Sie nun diese Adresse, öffnen Sie ein Browserfenster und geben Sie die Adresse in die Adresszeile des Browsers ein. Alternativ hierzu können Sie in die Adresszeile [localhost:8000](http://localhost:8000) eingeben. Wenn alles richtig funktioniert hat, sollten Sie folgende Anzeige sehen, die bestätigt, dass die Webseite als Testumgebung auf Ihrem lokalen Computer läuft:



**Abb. 17.4** Den Webserver für unsere Webseite starten

Glückwunsch – das ist bereits ein wichtiger Schritt auf dem Weg zu Ihrer ersten eigenen Website mit Django!

Wenn Sie das Kommandozeilenfenster schließen, wird die Verbindung zum Server wieder getrennt. Anstelle der obigen Anzeige erhalten Sie dann beim Öffnen von <http://127.0.0.1:8000/> eine Fehlermeldung. Schließen Sie also das Kommandozei-

## 17 Webseiten mit Django erstellen: Ein Einstieg

lenfenster nicht, solange Sie an Ihrer Website arbeiten. Um die Verbindung zum Entwicklungsserver aus der Kommandozeile anzuhalten, können Sie jederzeit **Strg + C** drücken und sie bei Bedarf wieder mit `python manage.py runserver` starten.

Öffnen Sie nun das angelegte Projekt in Ihrer IDE bzw. in PyCharm. Damit PyCharm die externe Django-Bibliothek finden und importieren kann, gehen Sie im PyCharm-Menü auf **File → Settings**. Im daraufhin geöffneten Einstellungsfenster klicken Sie im linken Navigationsfeld auf Ihr Projekt (**Project: ...**) und darunter auf die Option **Project Structure**. Auf der rechten Fensterseite sehen Sie nun die Option **+ Add Content Root**. Klicken Sie darauf und geben Sie anschließend den Django-Installationspfad ein, den Sie zuvor bereits notiert hatten. Bestätigen Sie mit **OK** und klicken Sie abschließend auf **Mark as: Sources**. Jetzt können Sie in PyCharm mit der Django-Bibliothek arbeiten.

Im untergeordneten *erstewebsite*-Ordner befindet sich unter anderem die Datei `settings.py`, in der wichtige Einstellungen der Webseite gespeichert sind. Hier lässt sich auch die Wahl der Datenbank konfigurieren, die standardmäßig auf SQLite gesetzt ist, was für unsere Programmiervorhaben genügen wird. Unter `LANGUAGE_CODE` können Sie anstelle von `'en-us'` die Angabe `'de-DE'` machen sowie bei `TIME_ZONE` mit `Europe/Berlin` auf lokale Gegebenheiten umstellen. Durch die Änderung zu `'de-DE'` wird beispielweise die Entwicklungswebserver-Seite des noch leeren Projekts nach dem Aktualisieren Ihres Browserfensters in deutscher Sprache angezeigt.

Die Programmdatei `urls.py` im *erstewebsite*-Ordner ist der Anfangspunkt für unsere Django-Website, der die möglichen Pfade der einzelnen Webseiten innerhalb der Website enthält. Im nächsten Abschnitt werden wir uns hauptsächlich mit dieser Programmdatei befassen.

Eine weitere wichtige Datei im selben Ordner ist die Datenbank-Datei `db.sqlite3`, die beim Starten des Entwicklungsservers durch `runserver` automatisch angelegt wurde. Mit dieser Datei lassen sich bei Bedarf Datenbankoperationen für Ihre Webseite vornehmen, die wir in Kapitel 17.5. untersuchen werden.

Aufgrund der klar vorgegebenen Struktur eines Django-Projekts haben wir die einzelnen Programmbeispiele im Folgenden nicht wie in den bisherigen Kapiteln durchnummierter. Die behandelten Codeteile finden Sie stattdessen in den jeweiligen Programmdateien des Django-Projekts *erstewebsite-project*.

### 17.2 Eine erste Webseite erstellen

Das Ziel dieses Abschnitts ist es, die im Browser angezeigte Webserver-Seite mit eigens kreierten Inhalten zu füllen. Hierfür müssen wir uns zunächst damit vertraut machen, wie der Informationsfluss, der beim Navigieren durch eine Website stattfin-

## 17.2 Eine erste Webseite erstellen

det, über Django geregelt wird. Jedes Mal, wenn eine Adresse auf der Website besucht wird, kommt die Datei `urls.py` ins Spiel: Bei jeder Anfrage einer spezifischen URL, zum Beispiel [www.djangoproject.com/start](http://www.djangoproject.com/start), wird in `urls.py` überprüft, ob dort ein solcher Pfad – hier also `/start` – existiert. Pfade lassen sich in dieser Datei unter `urlpatterns` definieren. Anfangs steht hierunter Folgendes:

```
1  urlpatterns = [
2      path('admin/', admin.site.urls),
3  ]
```

Das bedeutet, dass bislang lediglich ein '`admin/`'-Pfad definiert ist, der zur `admin`-Seite führt. Testen Sie dies einmal selbst, indem Sie diesen Teil zur Serveradresse hinzufügen, also <http://127.0.0.1:8000/admin/>. Sie gelangen dadurch zur Django-Verwaltungsseite, in der Sie sich mit Ihrem Benutzernamen und Passwort anmelden können. Mehr dazu in Abschnitt 17.5.1.

Sie können sich die Serveradresse [127.0.0.1:8000](http://127.0.0.1:8000) in unserer Testumgebung wie einen Platzhalter für eine Homepage, also die Startseite einer Website – beispielsweise [www.djangoproject.com](http://www.djangoproject.com) – vorstellen, auf der jede weitere Verzweigung durch einen Slash und den Pfadnamen – beispielsweise `/start` – ausgewiesen ist.

Um unsere Homepage mit einem selbstgewählten Inhalt zu füllen, können wir in `urlpatterns` folgende Zeile hinzufügen:

```
1  path('', views.home),
```

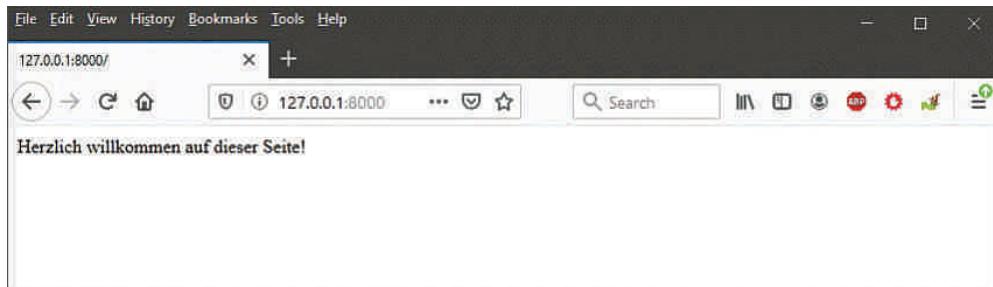
Die leeren Anführungszeichen bedeuten dabei, dass es sich um die Homepage handelt. Damit das `views`-Paket innerhalb von `urlpatterns` verwendet werden kann, müssen Sie in Ihrer IDE innerhalb des `erstewebsite`-Ordners zunächst eine Datei namens `views.py` anlegen und das Paket sodann in `urls.py` mit `from . import views` importieren. Der Punkt nach `import` gibt an, dass sich das `views`-Paket im selben Verzeichnis wie die `urls.py`-Datei befindet.

`home` bezieht sich auf eine Funktion, die das Aussehen der Homepage definiert und die in `views.py` definiert werden muss. Schreiben Sie also in diese Datei beispielsweise den Code

```
1  from django.http import HttpResponse
2
3  def home(request):
4      return HttpResponse("Herzlich willkommen auf dieser Seite!")
```

und aktualisieren Sie die Webserver-Ansicht in Ihrem Browser. Das neue Fenster sollte sofort angezeigt werden:

## 17 Webseiten mit Django erstellen: Ein Einstieg



**Abb. 17.5** Die Anzeige der ersten Webseite

In der ersten Zeile importieren wir die `HttpResponse`-Funktion, mithilfe derer sich Strings als HTML-Inhalt auf der Webseite anzeigen lassen. Dies geschieht durch die Definition der Funktion `home`, die den Begrüßungs-String zurückgibt. Das `request`-Objekt wird bei jeder Anfrage über die Homepage gesendet und enthält Metadaten zur Anfrage, beispielsweise über die Anfragemethode, Adresse oder Cookies.

Nach demselben Schema können wir in `urlpatterns` einen weiteren URL-Pfad hinzufügen, beispielsweise

```
1     path('seite2/', views.seite2),
```

Beachten Sie dabei, dass am Ende jedes Pfads ein Komma stehen muss. Der vollständige Inhalt der `urls.py`-Datei ist somit:

```
1 from django.contrib import admin
2 from django.urls import path
3 from . import views
4
5 urlpatterns = [
6     path('admin/', admin.site.urls),
7     path('', views.home),
8     path('seite2/', views.seite2),
9 ]
```

Gleichermaßen definieren wir in `views.py` eine weitere Funktion namens `seite2`, die beim Aufrufen des URL-Pfads `/seite2` ausgeführt wird:

```
1 def seite2(request):
2     return HttpResponse("<h1>Dies ist eine andere Seite.</h1>")
```

Geben Sie nun im Browser als Adresse <http://127.0.0.1:8000/seite2/> ein, so sollte die neue Seite angezeigt werden. Wie Sie möglicherweise bemerkt haben, ist die Schrift im Vergleich zur Schrift auf unserer Homepage nun größer gewählt. Dies liegt an der im zurückgegebenen String enthaltenen HTML-Formatierung durch `<h1>`-Tags. Wie diese Formatierung bei längeren Textinhalten effektiver vorgenommen werden kann, ist Bestandteil des nächsten Unterkapitels.

### 17.2.1 Templates anlegen und verwenden

In Kapitel 16 hatten wir beim Versenden einer E-Mail mit HTML-Inhalt bereits einige HTML-Formatierungsoptionen kennen gelernt. Selbstverständlich könnten Sie den gesamten HTML-Text einer Webseite als String innerhalb einer `HttpServletResponse`-Klammer eingeben, was jedoch nicht sonderlich praktisch wäre. Stattdessen verwendet man in Django zu diesem Zweck separate Textdateien, sogenannte *Templates* (engl. für *Schablone*), die den kompletten einzufügenden Textinhalt einer Webseite enthalten.

Templates bestehen in der Regel aus HTML-Text, können jedoch auch andere textbasierte Dateiformate wie etwa E-Mail- oder CSV-Dateien verarbeiten. Wir werden uns hier auf das HTML-Format beschränken. Neben den üblichen HTML-Formatierungen lässt sich in einem Template dabei außerdem Django-spezifischer Code verwenden, den wir im nächsten Unterkapitel genauer betrachten werden. Zunächst wollen wir jedoch unser erstes Template anlegen, mit Inhalt versehen und daraus im Schnellverfahren eine Webseite erstellen.

Eine Template-Datei enthält jeweils den vollständigen HTML-Inhalt für eine einzelne Webseite. Alle Template-Dateien werden direkt innerhalb Ihres PyCharm-Webseiten-Projekts in einem separaten Template-Ordner angelegt. Im vorliegenden Beispiel erstellen wir also im Ordner *erstewebsite-projekt* einen Unterordner namens *templates*. Legen Sie nun in diesem Ordner ein neues Template namens *home.html* an, indem Sie einen Rechtsklick auf den *templates*-Ordner machen und die Option **New → HTML File** wählen.

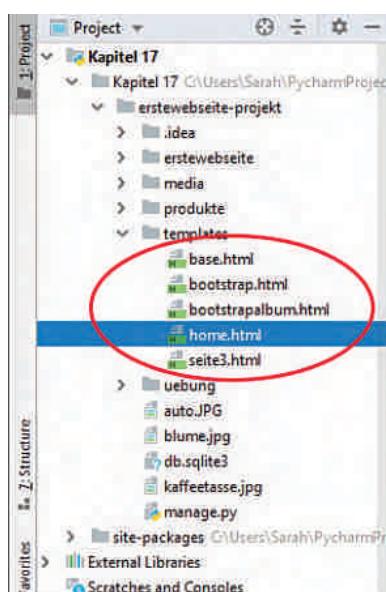


Abb. 17.6 Projektstruktur mit mehreren Templates

## 17 Webseiten mit Django erstellen: Ein Einstieg

Dieses Template wird den HTML-Code für unsere Homepage enthalten und ist in den meisten IDEs standardmäßig bereits mit einer minimalen HTML-Struktur versehen, die die folgende Gestalt hat:

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <title>Title</title>
6  </head>
7  <body>
8
9  </body>
10 </html>
```

Sie können hier innerhalb der `<title>`-Tags beispielsweise den Titel der Webseite anpassen, oder innerhalb der `<body>`-Tags weiteren Text für die Webseite, etwa eine Hauptüberschrift mit `<h1>`-Tags, einfügen. Für deutschsprachige Webseiten sollten Sie das `lang`-Attribut in der zweiten Zeile von "en" in "de" abändern.

Suchen Sie anschließend in der Datei `settings.py` nach dem Wort `TEMPLATES` und ändern Sie den Eintrag mit '`DIRS`' zu

```
1  'DIRS': ['templates'],
```

Hierdurch geben wir an, in welchem Ordner Django nach den Templates suchen soll. Anschließend muss das Template ebenfalls in der gewünschten Funktion innerhalb von `views.py` aufgeführt sein. Möchten wir beispielweise unsere vorige `home`-Funktion mit unserem neuen Template anpassen, verwenden wir dort anstelle von `HttpResponse` nun `render` nach dem folgenden Schema:

```

1  from django.shortcuts import render
2
3  def home(request):
4      return render(request, 'home.html')
```

Beachten Sie dabei, dass für die Verwendung der `render`-Funktion ebenfalls der Import von `render` aus `django.shortcuts` nötig ist. Als erstes Argument der `render`-Funktion muss hier stets ein `request`-Objekt stehen, während das zweite Argument den Namen des zu verwendenden Templates als String enthält. Sollte sich die HTML-Datei dabei in einem weiteren Unterordner des Templates-Ordners befinden – etwa `app_1` –, so müssen Sie diesen Unterordner ebenfalls folgendermaßen angeben: '`app_1/home.html`'.

Damit haben Sie alle nötigen Einstellungen vorgenommen. Um die neue Homepage tatsächlich anzeigen zu können, muss in die Datei `home.html` noch der gewünschte Inhalt der Homepage eingefügt werden. Bevor wir einzelne einfache HTML-Formatie-

## 17.2 Eine erste Webseite erstellen

rungen kennen lernen, können Sie dies mit einer beliebigen existierenden Webseite testen, da grundsätzlich alle Webseiten auf HTML-Code aufbauen.

In Ihrem Browser klicken Sie hierfür mit der rechten Maustaste auf eine beliebige Stelle innerhalb der gewünschten Webseite und wählen anschließend **Seitenquelltext anzeigen (View Page Source)** aus oder geben alternativ die Tastenkombination **Strg + U** ein. Der Quellcode wird in einem neuen Fenster angezeigt. Kopieren Sie den kompletten Code und fügen Sie ihn in das Template *home.html* ein. Aktualisieren Sie anschließend die Ansicht des Webservers. Wenn Sie alle Schritte richtig ausgeführt haben, sollte die Webseite nun in Ihrem Browserfenster angezeigt werden:

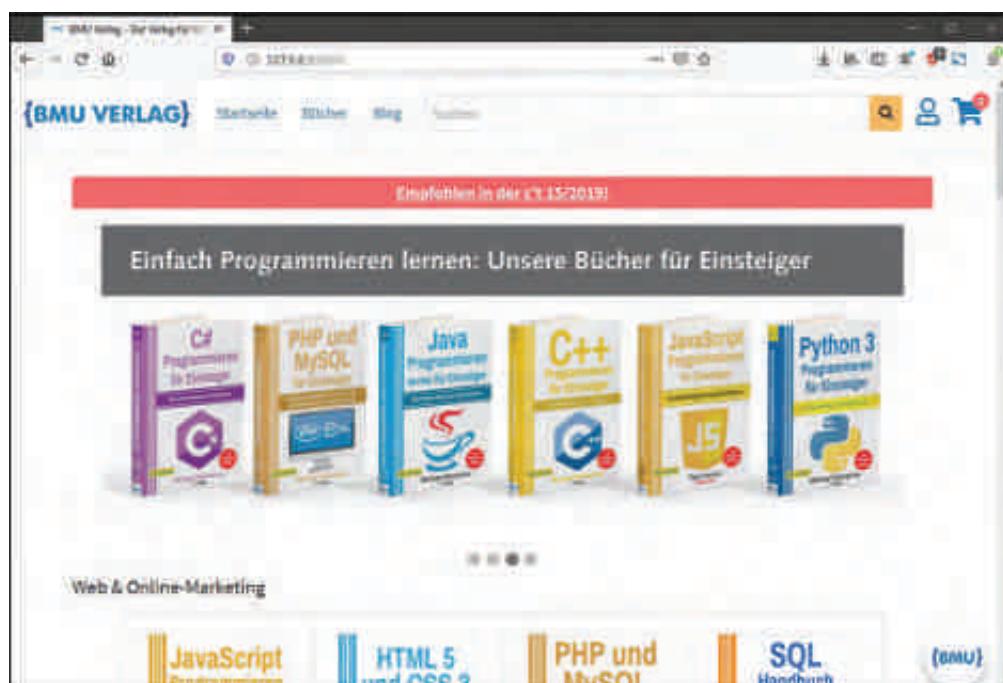


Abb. 17.7 Anzeigen einer Webseite aus einem HTML-Template

Beim Anklicken einzelner Links werden diese selbstverständlich nicht funktionieren, da hierfür der oben beschriebene Vorgang für die verlinkte Seite mitsamt einer entsprechenden Verlinkung durch spezielle HTML-Tags ebenfalls neu vorgenommen, eine Funktion in *views.py* definiert und ein Eintrag unter *urlpatterns* in der Datei *urls.py* angelegt werden muss.

Wie eine vereinfachte Verlinkung der beiden bereits angelegten Seiten *Homepage* und *Seite 2* anhand von speziellen HTML-Tags gelingt, werden wir am Ende des nächsten Unterkapitels untersuchen.

## 17 Webseiten mit Django erstellen: Ein Einstieg

### 17.2.2 Templates: Variablen, Filter und Tags

Innerhalb von Templates wird Django-spezifischer Code, auch *Django Template Language (DTL)* genannt, separat durch geschweifte Klammern ausgewiesen. Er ermöglicht einige Python-Befehle, die bei der Auswertung des HTML-Inhalts des Templates durch Django mit den entsprechenden dynamischen Werten ersetzt werden. Django-Code kann aus Variablen, Filtern und Tags bestehen und erlaubt ebenso das Einfügen von Kommentaren. Ein Template lässt damit unter anderem die Verwendung von Python-Elementen wie Dictionaries, Schleifen und anderen Kontrollstrukturen zu.

**Variablen** im Django-Code, die bei der Erstellung der Webseite zum Einsatz kommen, werden mit doppelten geschweiften Klammern ausgewiesen, etwa: `{ { buch1 } }`. Template-Variablen beziehen sich meistens auf ein Dictionary im Python-Code, können jedoch auch komplexere Datenstrukturen handhaben, darunter etwa Objektattribute nach dem Schema `{ { sachbuch.buch1 } }`.

Beispielsweise könnten in einem Online-Shop eines Verlags Bücher angezeigt werden, deren Titel und Preise jeweils in der separaten Datei `views.py` als Python-Code in einem Dictionary namens `buchbestand` gespeichert sind. Innerhalb des Templates wird die jeweilige Django-Variable für ein spezifisches Buch dann durch den Dictionary-Schlüssel `{ { buch1 } }` referenziert, wodurch auf der Webseite ihr Wert, also der entsprechende Preis, angezeigt wird.

Das Dictionary verwenden Sie, indem Sie dieses in der Datei `views.py` derjenigen Funktion hinzufügen, die auf der Webseite zum Einsatz kommen soll:

```

1 def home(request):
2     buchbestand = {'buch1': '25',
3                     'buch2': '17',
4                     'buch3': '18'}
5     return render(request, 'home.html', buchbestand)

```

Der Name des Dictionaries muss dabei ebenfalls in der `render`-Funktion angegeben werden. Anstatt des Namens `buchbestand` können Sie dort selbstverständlich auch das gesamte Dictionary eintragen. Anschließend lassen sich einzelne Einträge aus dem Dictionary im Django-Code Ihres `home.html`-Templates durch die Nennung des Schlüssels – etwa `{ { buch1 } }` – aufrufen. Auf der Webseite wird daraufhin der Wert des Dictionaries, im konkreten Fall 25 als Preis von `buch1`, angezeigt.

**Filter** werden auf Django-Variablen angewandt, um deren Anzeige einem gewünschten Format entsprechend anzupassen. Sie werden durch zwei geschweifte Klammern und einen senkrechten Strich gekennzeichnet. Beispielsweise können wir das Dictionary `{ 'name': 'DONALD' }` mithilfe der `lower`-Funktion durch das Tag `{ { name|lower } }` auf der Webseite in ein Wort verwandeln, das ausschließlich aus Kleinbuchstaben besteht: `donald`.

## 17.2 Eine erste Webseite erstellen

Eine weitere nützliche Filteroption ist `length`. Analog zur `len()`-Funktion in Python kann hierbei sowohl für Strings als auch für Listen die Anzahl der Elemente des Strings bzw. der Liste angegeben werden. Für das Dictionary `{'buch': "Python 3"}` würde `{{ buch|length }}` etwa den Wert 8 ergeben.

Django verfügt über etwa 60 unterschiedliche Filteroptionen, die auf der Webseite der Django-Dokumentation in einer Übersicht aufgeführt sind.

Ein Template-**Tag** mit einem spezifischen `inhalt` hat die allgemeine Form `{%inhalt%}`. Diese Tags dienen dazu, eine bestimmte Aktion auszuführen, beispielsweise durch logische Ausdrücke wie `{%if%} ... {%endif%}` oder Schleifen wie `{%forxiny%} ... {%endfor%}`. Wie Sie sehen, wird in diesem Fall jeweils ein Start- und Endtag benötigt um genau zu markieren, auf welchen Inhalt sich das Tag innerhalb des Django-Codes bezieht. Im folgenden Code werden beispielsweise durch ein `for`-Tag jeweils der Titel aller in `buchliste` abgespeicherten Bücher ausgegeben, wobei `buchliste` in `views.py` durch das Dictionary `{'buchliste': ["Hamlet", "Python 3", "Frankenstein"]}` definiert ist.

```

1 <ul>
2 {% for buch in buchliste %}
3   <li>{{ buch }}</li>
4 {% endfor %}
5 </ul>
```

Das `<ul>`-Tag bedeutet dabei, dass es sich um eine ungeordnete Liste handelt, deren einzelne Einträge jeweils in `<li>`-Tags stehen und mit Aufzählungspunkten angezeigt werden. Geordnete Listen werden hingegen durch `<ol>`-Tags ausgewiesen, die eine Nummerierung der einzelnen Einträge vornehmen.

Eine `if-else`-Anweisung können wir für `buchliste` etwa folgendermaßen einsetzen:

```

1 {% if buchliste %}
2   Anzahl Bücher:<p>{{ buchliste|length }}</p>
3 {% else %}
4   Keine Bücher
5 {% endif %}
```

Die `<p>`-Tags definieren dabei jeweils einen neuen Abschnitt. Zusätzlich zu den `if-else`-Klauseln können Sie für weitere Fälle ebenfalls das `elif`-Tag mit `{%elif%}` verwenden.

Auch **Kommentare** werden in geschweifte Klammern gesetzt. Dabei wird der auskommentierte Text wie in Python-Code durch ein Raute-Symbol gekennzeichnet, das hier auch abschließend gesetzt werden muss:

## 17 Webseiten mit Django erstellen: Ein Einstieg

```
1  {# Dies ist ein Kommentar #}
```

Eine sehr wichtige Funktionalität einer Website besteht in der Verlinkung zwischen einzelnen Seiten, wodurch etwa das Navigieren durch den gesamten Webauftritt ermöglicht oder auch auf externe Webseiten verwiesen werden kann. Möchten wir beispielsweise von unserer Homepage aus auf /seite2 verlinken, können wir hierfür auf der Homepage einen anklickbaren Text, *Seite 2*, vorsehen. Die Verlinkung wird durch sogenannte <a>-Tags bewerkstelligt, die wir im vorliegenden Beispiel in der HTML-Datei *home.html* für die Homepage einfügen:

```
1  <a href="{% url 'seite2' %}">Seite 2</a>
```

Bevor die Verlinkung wirksam ist, muss eine weitere Änderung in *urls.py* vorgenommen werden, indem die Namensbezeichnung des *seite2*-Pfads entsprechend durch '*seite2*' angegeben wird:

```
1      path('seite2/', views.seite2, name='seite2'),
```

Wenn Sie die Server-Homepage nun in Ihrem Browser aktualisieren, sollte sich darauf eine anklickbare Verlinkung zur gewünschten Seite befinden.

Um auf externe Webseiten zu verweisen, genügt der folgende Code ohne weitere Angaben in *urls.py*:

```
1  <a href="https://docs.djangoproject.com/en/3.0/ref/templates/
2  builtins/">Django-Dokumentation: Template-Tags & -Filter</a>
```

Unter dieser Adresse finden Sie übrigens eine nützliche Übersicht über alle in Django verfügbaren Tags und Filter.

### 17.2.3 Templates-Erweiterungen

Oftmals ist es erwünscht, Teile eines Templates, die wie ein Gerüst die Grundstruktur der Webseite ausmachen, auf mehreren Seiten einzusetzen. Beispielsweise wäre es denkbar, dass die Ober- oder Unterzeile einer Webseite, die ein Navigationsmenü, weitere Links oder Kontaktinformationen enthalten, auf jeder Seite des Webauftritts sichtbar sein sollen. Hierzu muss man nicht jedes Mal ein komplett neues Template anlegen, sondern kann – ähnlich wie bei der Vererbung von Klassen in der objekt-orientierten Programmierung – bereits bestehende Basis-Templates ohne Wiederholungen und platzsparend zu einem neuen Template kombinieren und mit den für die aktuelle Seite gewünschten spezifischen Inhalten versehen. Dies hat auch den Vorteil, dass nachträgliche Änderungen etwa am Navigationsmenü nicht in jedem einzelnen Template vorgenommen werden müssen, sondern sich direkt aus dem Basis-Template auf alle daraus abgeleiteten Templates auswirken. Bei einer Webseite, die aus 20

## 17.2 Eine erste Webseite erstellen

verschiedenen Seiten – und somit Templates – besteht, spart dies automatisch eine Menge fehleranfälliger Arbeit.

Das Basis-Template im Template-Ordner, das alle sich wiederholenden Teile der Webseite beinhaltet, trägt dabei meist den Namen *base.html*. Das Template einer spezifischen Seite – nennen wir es hier *seite3.html* – befindet sich ebenfalls im Template-Ordner und beginnt demgemäß stets mit der Zeile

```
1  {% extends 'base.html' %}
```

Darunter steht der jeweilige Inhalt der individuellen Seite. Somit lassen sich aus dem Basis-Template die unterschiedlichen Seiten für den gesamten Webauftritt ableiten. Diejenigen Teile des Basis-Templates, die für jede Seite individuell angepasst werden sollen, lassen sich darin durch unterschiedliche Blöcke kennzeichnen. Sie haben die allgemeine Form

```
1  {% block inhalt %}  
2  {% endblock %}
```

und stehen stellvertretend für den Inhalt des Templates einer spezifischen Seite, den Python beim Auslesen der Dateien sodann einfügt.

So könnte unser Template *seite3.html* etwa die folgende Gestalt haben:

```
1  {% extends 'base.html' %}  
2  
3  {% block inhalt %}  
4      <h1>Herzlich willkommen auf dieser Seite!</h1>  
5      <h2>Auf der anderen Seite ist das Gras viel grüner.</h2>  
6      <h3>Aber auf dieser Seite scheint die Sonne.</h3>  
7      <h4>Einerseits...</h4>  
8      <h5>... und andererseits.</h5>  
9  {% endblock %}
```

Beachten Sie im obigen Code ebenfalls die HTML-Tags für unterschiedliche Überschriften.

Das Basis-Template hat nun dementsprechend die folgende einfache Form:

```
1  <!DOCTYPE html>  
2  <html lang="de">  
3  <head>  
4      <meta charset="UTF-8">  
5      <title>Meine Webseite</title>  
6  </head>  
7  <body>  
8      {% block inhalt %}  
9      {% endblock %}  
10 </body>
```

## 17 Webseiten mit Django erstellen: Ein Einstieg

```
11 </html>
```

Innerhalb der <body>-Tags wird beim Auswerten des Templates für die Seite 3 nun jeweils im Basis-Template der gesamte `inhalt`-Block eingefügt. Somit lassen sich für mehrere Seiten an den gewünschten Stellen des Basis-Templates unterschiedliche Inhalte verwenden.

Legen Sie nun nach dem in 17.2. beschriebenen Verfahren in `urls.py` einen neuen Pfad namens `/seite3` an. In `views.py` genügt es, in der neu zu definierenden Funktion `seite3` auf das `seite3.html`-Template zu verweisen, da dieses bereits den Verweis auf das Basis-Template enthält:

```
1 def seite3(request):
2     return render(request, 'seite3.html')
```

Wenn die Vererbung richtig durchgeführt wurde, sollte unter <http://127.0.0.1:8000/seite3/> nun Folgendes angezeigt werden:



**Abb. 17.8** Anzeigen einer Webseite aus einem vererbten Template

Zusätzlich zur Ableitung von Templates aus einem Basis-Template besteht die Möglichkeit, an einer bestimmten Stelle innerhalb eines Templates weitere Templates einzubinden. Hierfür verwendet man ein Template-Tag mit `include` nach dem folgenden Schema:

```
1 { % include 'neuestemplate.html' %}
```

### 17.3 App vs. Projekt

Apps befinden sich von der Organisationsstruktur her unterhalb eines Django-Webseiten-Projekts. Die Idee dahinter ist, Projekte in separate Apps einzuteilen, die jeweils verschiedene Funktionalitäten innerhalb der Webseite separat handhaben. Ein

Projekt für eine Webseite kann somit mehrere Apps beinhalten. Durch diese Einteilung wird das Projekt übersichtlicher und klarer strukturiert. Ein weiterer Vorteil der Einteilung in separate Apps ist zudem, dass sich die Apps dadurch flexibel in unterschiedlichen Projekten einsetzen lassen.

Hierfür können Sie sich beispielsweise vorstellen, dass innerhalb einer Webseite für einen Online-Shop drei verschiedene Apps jeweils für Login- und Registrierungsvorgänge von Kundenkonten, für das Produktangebot mit Bildern und Informationen sowie zur Anzeige ausgewählter Produkte in einem Warenkorb eingesetzt werden. Dabei liegen Dateien, die nur für die spezifischen Apps und deren Einstellungen gelten, jeweils im Unterordner der App, während die Einstellungen und Dateien für das gesamte Projekt – darunter etwa auch gemeinsame Template-Dateien – im übergeordneten Projektordner gespeichert sind. In einem App-Ordner wird sich daher wiederum eine *views.py*- und eine *urls.py*-Datei befinden.

Bevor Sie ein neues Webseiten-Projekt starten, sollten Sie einen konkreten Plan – gerne in Form einer Skizze – erstellen, in dem genau angegeben ist, wie Sie die Webseite aufbauen wollen. Hierdurch wird leichter ersichtlich, welche Funktionalitäten Sie in separate Apps verlagern können.

Im Folgenden werden wir das Beispiel einer Produkte-App untersuchen, in der das gesamte Angebot eines Online-Shops angezeigt werden soll. Um eine neue App anzulegen, geben Sie im Kommandozeilenfenster aus dem *erstewebsite-projekt*-Ordner heraus (`cd erstewebsite-projekt`) den folgenden Befehl ein:

```
1 python manage.py startapp produkte
```

Die neue Produkte-App ist somit Bestandteil des *erstewebsite-projekt*-Ordners. Damit sie innerhalb Ihres Projekts verwendet werden kann, müssen Sie sie in den Projekt-Einstellungen unter *erstewebsite-projekt\erstewebsite\settings.py* in *INSTALLED\_APPS* angeben:

```
1 INSTALLED_APPS = [
2     'django.contrib.admin',
3     'django.contrib.auth',
4     'django.contrib.contenttypes',
5     'django.contrib.sessions',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8     'produkte', # hier steht die neue App
9 ]
```

Im neu erstellten *produkte*-Ordner wurde unter anderem bereits automatisch eine neue Datei namens *views.py* angelegt, die einen `render`-Import enthält. Fügen Sie dieser Datei nun den folgenden Code hinzu:

## 17 Webseiten mit Django erstellen: Ein Einstieg

```

1  from django.shortcuts import render
2
3  def produkte(request):
4      return render(request, 'produkte.html', {})

```

Beim Aufrufen der `produkte`-Funktion wird die HTML-Datei `produkte.html` ausgewertet, die bislang noch nicht existiert. Legen Sie dazu also im `produkte`-Ordner einen neuen `templates`-Ordner und darin die HTML-Datei `produkte.html` mit dem Code `<h1>Dies ist das Angebot</h1>` im Textkörper (`<body>`) an.

Im Falle von App-Templates müssen Sie in `settings.py` unter `TEMPLATES` bei `'DIRS'` keine zusätzlichen Angaben zu neuen Templates-Ordnern machen, da Templates-Ordner innerhalb von Apps im Gegensatz zu Templates-Ordnern im Hauptprojekt-Ordner standardmäßig automatisch erkannt werden.

Auch für Apps innerhalb eines Webseiten-Projekts muss die `.url`-Pfadangabe vorgenommen werden. Hierzu ändern Sie die `urls.py`-Datei im `erstewebsite`-Ordner an den folgenden zwei kommentierten Stellen:

```

1  from django.contrib import admin
2  from django.urls import path, include # Modul 'include' importieren
3  from . import views
4
5  urlpatterns = [
6      path('admin/', admin.site.urls),
7      path('', views.home),
8      path('seite2/', views.seite2, name='seite2'),
9      path('seite3/', views.seite3, name='seite3'),
10     path('', include('produkte.urls')), # neuen Pfad angeben
11 ]

```

Beim Aufrufen Ihrer Webseite sucht Django nun innerhalb der Produkte-App nach dem Modul `urls.py` und verarbeitet die darin angegebenen URL-Pfade. Legen Sie dieses Modul nun also ebenfalls im `produkte`-Ordner an und versehen Sie es mit dem folgenden Inhalt:

```

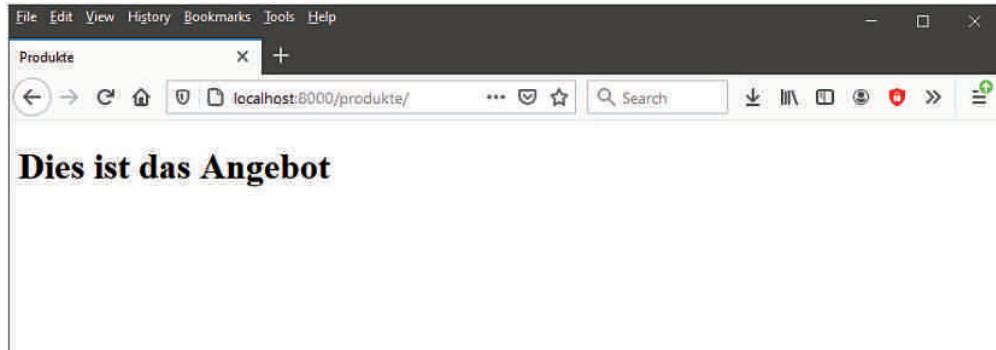
1  from django.urls import path
2  from . import views
3
4  urlpatterns = [
5      path('produkte/', views.produkte, name='produkte'), # neuen Pfad angeben
6 ]

```

Es mag etwas verwirrend erscheinen, dass zwei Dateien mit dem Namen `urls.py` existieren – eine Datei für das gesamte Webseiten-Projekt und eine für die Produkte-App. Erstere übernimmt dabei URL-Verlinkungen zu einzelnen Seiten und Apps, während letztere ausschließlich die URL-Verlinkungen innerhalb der erstellten App vollzieht.

## 17.4 Das Layout einer Webseite mit Bootstrap anpassen

Vergewissern Sie sich nun, dass Ihr Server läuft, und besuchen Sie in Ihrem Browser die Seite [localhost:8000/produkte](http://localhost:8000/produkte). Wenn Sie alle Einstellungen und Änderungen richtig vorgenommen haben, sollte dort nun der HTML-Inhalt der Template-Datei *produkte.html* zu sehen sein:



**Abb. 17.9** Teile einer Webseite aus einer App erstellen

## 17.4 Das Layout einer Webseite mit Bootstrap anpassen

Bislang haben wir uns vordergründig mit den technischen Details der Webseitengestaltung befasst, da Django in erster Linie die für das Erstellen der Webseite benötigte Backend-Infrastruktur bereithält, während andere Sprachen und Frameworks dem Frontend-Design dienen. Letzteres soll hier jedoch nicht gänzlich unbeachtet bleiben. In diesem Kapitel werden wir Ihnen daher eines der derzeit beliebtesten Frontend-Frameworks, *Bootstrap*, im Rahmen eines kurzen Einstiegs vorstellen.

Die Auszeichnungssprache HTML (Abk.: *Hypertext Markup Language*) ist zwar auch heute noch die Grundlage jeder Webseite, war aber ursprünglich nur für die Handhabung des Inhalts und der allgemeinen Struktur von Webseiten gedacht. Für das Layout von modernen Webseiten finden heutzutage üblicherweise andere Sprachen Gebrauch. So kommt zur Webseitengestaltung in der Regel ergänzend die Stylesheet-Sprache CSS (Abk.: *Cascading Style Sheets*) zum Einsatz, die das Einbinden von Layout-Vorgaben ermöglicht – sowohl direkt im HTML-Dokument als auch aus externen Dateien. Mit CSS lassen sich HTML-Elemente beliebig anpassen, sodass sich auf einem simplen HTML-Gerüst aufbauend optisch sehr ansprechende Webseiten kreieren lassen. Für zusätzliche dynamische Funktionalitäten wie Dialogfenster, die Validierung von Formulardaten, für Animationen und interaktive Inhalte wie Spiele und Videos wird heute darüber hinaus für gewöhnlich JavaScript verwendet.

Doch auch ohne vertiefte HTML-, CSS- oder JavaScript-Kenntnisse können Sie ansprechende Web-Inhalte erstellen. Das kostenlose Bootstrap-Framework – derzeit in Version 4 in Gebrauch – ermöglicht das Zusammenführen von HTML, CSS und JavaScript

## 17 Webseiten mit Django erstellen: Ein Einstieg

in einem einzigen Template. Es enthält viele vorgefertigte Layout-Vorlagen, die sich ganz einfach in ein bestehendes HTML-Dokument einbinden und anschließend je nach Bedarf anpassen lassen. Mithilfe von Bootstrap kann man somit schneller und mit weniger Aufwand eine hochwertige, professionelle Webseite erstellen. Ein weiteres Plus von Bootstrap ist, dass sich damit erstellte Webseiten aufgrund seines responsiven Designs auf Bildschirmen unterschiedlicher Auflösung und Größen – von Desktop-PCs bis hin zu Tablets und Smartphones – optimal anzeigen lassen.

Legen Sie nun zunächst ein neues Template *bootstrap.html* innerhalb des Template-Ordners Ihres Projekts an und nehmen Sie in *urls.py* durch

```
1 path('bootstrap/', views.bootstrap),
```

die nötige Verlinkung zu *bootstrap/* in *urlpatterns* vor. Legen Sie danach mit

```
1 def bootstrap(request):
2     return render(request, 'bootstrap.html')
```

einen Eintrag in *views.py* an. Die Template-Datei *bootstrap.html* kann nun beispielsweise die folgende Grundgestalt haben:

```
1 <!DOCTYPE html>
2 <html lang="de">
3 <head>
4     <meta charset="UTF-8">
5     <title>Bootstrap-Beispieleseite</title>
6 </head>
7 <body>
8     <h1>Meine erste Bootstrap-Seite</h1>
9     <p>Hier steht Text.</p>
10    </body>
11 </html>
```

Überprüfen Sie nun zuerst, dass die URL [localhost:8000/bootstrap](http://localhost:8000/bootstrap) funktioniert. Öffnen Sie sodann die Bootstrap-Webseite [www.getbootstrap.com](http://www.getbootstrap.com). Hier finden Sie eine ausführliche Dokumentation sowie zahlreiche Beispiele und weitere nützliche Informationen zum Framework. Um Bootstrap-Code in Ihrer HTML-Datei verwenden zu können, müssen Sie diesen entweder herunterladen oder den Zugang über das Bootstrap-Netzwerk *CDN* (engl. für *Content Delivery Network*) herstellen, wodurch die benötigten Inhalte bereitgestellt werden. Im Folgenden werden wir letztere Option verwenden.

Suchen Sie hierfür auf der Bootstrap-Startseite nach *BootstrapCDN* und kopieren Sie den Link für das CSS-Stylesheet innerhalb des *<head>*-Tags Ihres HTML-Codes und die drei JavaScript-Zeilen am Ende des HTML-Texts vor dem schließenden *</body>*-Tag.

## 17.4 Das Layout einer Webseite mit Bootstrap anpassen



**Abb. 17.10** Die vier benötigten CSS- und JS-Zeilen (rechts) aus BootstrapCDN

**Achtung:** Die benötigten Zeilen sind für neuere Versionen eventuell unter jsDelivr zu finden. Die hier verwendeten Codezeilen funktionieren jedoch weiterhin.

Ihre HTML-Datei *bootstrap.html* sollte nun folgendermaßen aussehen:

```

1  <!DOCTYPE html>
2  <html lang="de">
3  <head>
4      <meta charset="UTF-8">
5      <title>Bootstrap-Beispieleseite</title>
6      <!-- CSS only -->
7      <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/css/bootstrap.min.css" integrity="sha384-aIt2nRpC12Uk9gS9baDl411NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk" crossorigin="anonymous">
8
9
10 </head>
11 <body>
12     <h1>Meine erste Bootstrap-Seite</h1>
13     <p>Hier steht Text.</p>
14     <!-- JS, Popper.js, and jQuery -->
15     <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamOVi38MVBN+E+bVYUew+OrCxArkfj" crossorigin="anonymous"></script>
16     <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UKsdQRVvoxMfooAo" crossorigin="anonymous"></script>
17     <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.0/js/bootstrap.min.js" integrity="sha384-OgVRvuATP1z7JjHLkuOU7Xw704+h835Lr+6QL9UvYjZE3Ip" u6Tp75j7Bh/kR0JKI" crossorigin="anonymous"></script>
18
19
20 </body>
21 </html>

```

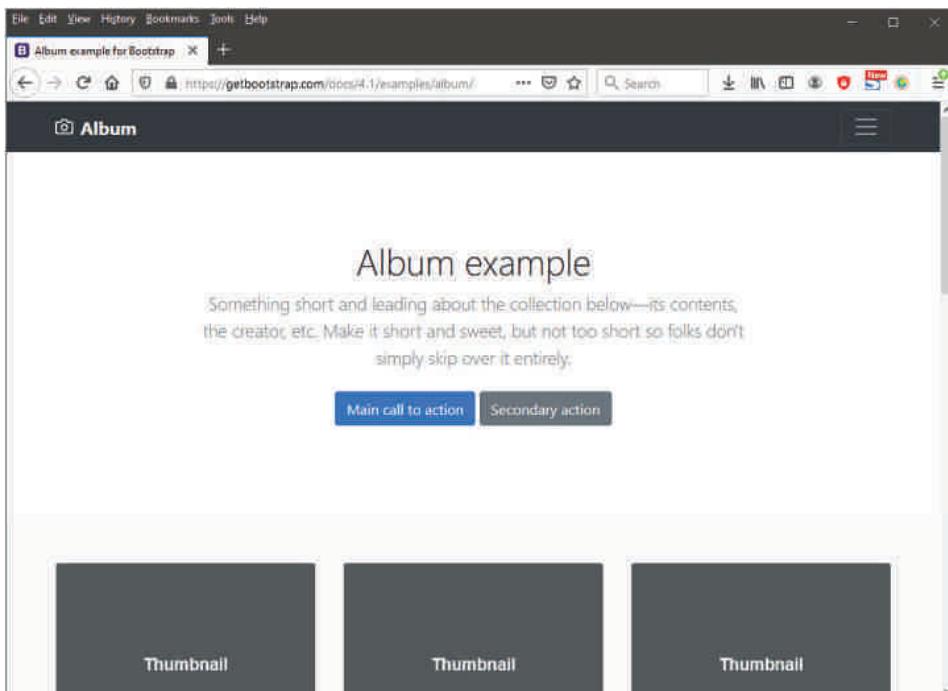
Aktualisieren Sie nun die URL [localhost:8000/bootstrap](http://localhost:8000/bootstrap). Der Seitentext sollte sich nun dem Bootstrap-Layout entsprechend geändert haben.

## 17 Webseiten mit Django erstellen: Ein Einstieg



**Abb. 17.11** Durch Bootstrap-Code formatierter Text

Unter <https://getbootstrap.com/docs/4.1/examples/> finden Sie viele Beispielvorlagen, die Sie für Ihre Webseite nutzen und bei Bedarf individuell anpassen können. Suchen Sie sich darunter eines aus – wir wählen hier *Album* –, klicken Sie darauf und lassen Sie sich wie in 17.2.1. beschrieben den Seitenquelltext anzeigen. Kopieren Sie diesen Inhalt nun in Ihre HTML-Datei. Wenn Sie nun in Ihrem Browser `localhost:8000/bootstrap` aufrufen, werden Sie feststellen, dass die Seite lediglich in reinem HTML-Format angezeigt wird. Um das gewünschte Bootstrap-Layout darstellen zu können, müssen Sie auch hier die obigen vier CSS- bzw. JavaScript-Zeilen an den entsprechenden Stellen im Code einfügen. Löschen Sie dabei zuvor die bereits vorhandenen Referenzen. Die gewählte Bootstrap-Vorlage sollte nun korrekt dargestellt werden.



**Abb. 17.12** Die Bootstrap-Beispielvorlage *Album*

Sie können nun selbst versuchen, die angezeigten Textstellen im Code anzupassen oder nicht benötigte Teile daraus zu löschen. Weitere Informationen zu den vielfältigen Gestaltungsmöglichkeiten von Bootstrap finden Sie auf <https://getbootstrap.com/>.

## 17.5 Mit Datenbanken arbeiten

Wie die meisten Web-Frameworks ist auch Django auf die sogenannte MVC-Architektur gestützt. *MVC* steht für *Model-View-Controlling* und bezeichnet einen speziellen Aufbau eines Frameworks, bei dem unterschiedliche Funktionalitäten jeweils auf Modell-, View- und Controller-Dateien verteilt sind. Mit dem *View*-Teil, der die unterschiedlichen Ansichten der Webseite übernimmt, haben Sie sich bereits durch das Arbeiten mit den *views.py*-Dateien vertraut gemacht. Die *Modelle* definieren die grundlegende Datenstruktur und werden durch Datenbank-Dateien gehandhabt. Der *Controller* wiederum koordiniert das Zusammenspiel von Datenbank und View.

Im Unterschied zu anderen Web-Frameworks übernimmt Django den Controller-Teil jedoch selbst im Hintergrund, ohne dass Sie dazu Vorgaben machen müssen. Präziser ausgedrückt funktioniert Django nämlich nach dem Schema *Model-View-Template (MVT)*, in dem View und Template zusammen den entsprechenden View-Teil der MVC-Architektur ausmachen. Der letzte große Thementeil dieses Kapitels wird sich daher der Verwendung von Datenbanken widmen, die den Modell-Part der MVT-Architektur in Django darstellen.

Jedes Modell enthält diejenigen Daten, auf denen ein spezifischer Teil Ihrer Webseite basiert, in Form einer Datenbank. Üblicherweise existiert ein Modell pro App. Ein Modell wird mithilfe der beim Erstellen der App automatisch angelegten Datei *models.py* gemanagt und darin als Unterklasse der Python-Klasse `django.db.models.Model` angelegt. Jedes Klassenattribut steht dabei für ein Feld in der Datenbank, das jeweils einer Tabellenspalte entspricht. Für den Zugang zur Datenbank sind dabei keine datenbankspezifischen Befehle nötig, wie Sie diese aus Kapitel 14 kennen. Stattdessen werden die CRUD-Operationen über die Objekte der Klassen veranlasst, die die Datenbank beschreiben.

Standardmäßig verwendet Django die in der Python-Standardsbibliothek enthaltene SQLite-Datenbank. Wenn Sie ein anderes Datenbankmanagementsystem verwenden wollen, müssen Sie entsprechende Angaben in *settings.py* machen und die Vorgänge zum Einbinden der gewünschten Datenbank durchführen. Für fortgeschrittenere Zwecke ist meist PostgreSQL die Datenbank der Wahl, für unsere einfachen Testbeispiele wird jedoch die Standard-Datenbank SQLite genügen.

Legen wir nun beispielsweise in *models.py* eine neue Klasse `Produkte` für unsere `Produkte`-App an, die von der `Model`-Klasse erbt und deren Objekte die in `Produkte`

## 17 Webseiten mit Django erstellen: Ein Einstieg

enthalteten Artikel sind. Wir versehen sie mit den Attributen `bezeichnung`, `artnr`, `beschreibung`, und `preis`:

```

1  from django.db import models
2
3  class Produkte(models.Model):
4      bezeichnung = models.CharField(max_length=50)
5      artnr = models.IntegerField()
6      beschreibung = models.TextField()
7      preis = models.FloatField()
8
9      class Meta:
10         verbose_name_plural = "Produkte"
11
12     def __str__(self):
13         return self.bezeichnung

```

Die unterschiedlichen Feld-Klassentypen beschreiben dabei das Format, in dem die Daten in der Datenbank abgespeichert werden sollen. Beispielsweise definiert `CharField` einen String und benötigt zudem die Angabe einer Maximallänge durch `max_length`, während `TextField` Textinhalte von unbegrenzter Länge aufnehmen kann. Jedes Feld Ihres Modells ist somit eine Instanz einer bestimmten `Field`-Klasse, die unter anderem auch Zahlen- oder Datumformate darstellen kann. Eine Übersicht aller vorhandenen `Field`-Klassentypen finden Sie in der Django-Dokumentation unter <https://docs.djangoproject.com/en/3.0/ref/models/fields/#model-field-types>.

Durch `class Meta` innerhalb der `Produkte`-Klassendefinition wird die standardmäßige Pluralbildung des Klassennamens mit „s“ in der Anzeige der Django-Verwaltung, die wir im nächsten Unterkapitel untersuchen werden, verhindert.

Modelle können unterschiedliche Methoden beinhalten, etwa zum Anzeigen bestimmter Einträge auf der Webseite. Zum mindesten sollte Ihr Modell die Standard-Klassenmethode `__str__` enthalten, die einen String mit dem gewünschten Inhalt – hier ist es die Artikelbezeichnung – für jedes definierte Objekt ausgibt.

Durch diesen Code wird bereits eine Datenbank für unsere App definiert, wie wir sie beispielsweise in SQLite mit dem Befehl `CREATE TABLE` angelegt hätten. Denken Sie daran, dass der Name der App, die Ihr Modell enthält – in unserem Fall `'produkte'` – unbedingt in `settings.py` unter `INSTALLED_APPS` angegeben sein muss, damit Django das neue Modell verwenden kann.

Um unsere Datenbank mit dem neuen Modell zu aktualisieren, müssen wir anschließend durch

```
1  python manage.py makemigrations produkte
```

einen entsprechenden Befehl in der Kommandozeile ausführen. Die Angabe des App-Namens, `produkte`, ist dabei optional. Wird kein Name angegeben, wird die Migration für alle Apps ausgeführt. Python hat nun eine Migrationsdatei erstellt, die wir auf unsere Datenbank anwenden müssen, indem wir den folgenden Befehl ausführen:

```
1 python manage.py migrate
```

Auch in diesem Fall ist die zusätzliche Angabe des App-Namens optional. Wenn Sie die Migrationen zum ersten Mal ausführen, ist es jedoch eine gute Idee, diese für alle Apps vorzunehmen. Damit ist sichergestellt, dass Ihre Datenbank die notwendigen Tabellen für alle erforderlichen Funktionalitäten, beispielsweise Nutzerauthentifizierung und Sitzungsverwaltung, enthält.

Die beiden Migrations-Befehle müssen Sie jedes Mal ausführen, wenn Sie Änderungen an Ihrem Modell vornehmen, um dieses dementsprechend zu aktualisieren.

In der Kommandozeile sollte nun der folgende Output angezeigt werden:

```
(myenv) C:\Users\Sarah\PycharmProjects\erstewebsite-projekt>python manage.py makemigrations produkte
Migrations for 'produkte':
  produkte\migrations\0001_initial.py
    - Create model Produkte

(myenv) C:\Users\Sarah\PycharmProjects\erstewebsite-projekt>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, produkte, sessions
Running migrations:
  Applying produkte.0001_initial... OK

(myenv) C:\Users\Sarah\PycharmProjects\erstewebsite-projekt>
```

**Abb. 17.13** Migrationsbefehle zum Anlegen eines neuen Modells

Das neu angelegte `Produkte`-Modell ist somit in unserer Datenbank enthalten.

Es gibt unterschiedliche Möglichkeiten, wie Sie Ihrer Datenbank neue Objekte hinzufügen können. Beispielsweise können Sie den nachfolgenden Code in die Datei `views.py` innerhalb der `Produkte`-App aufnehmen und nach Belieben weitere Objekte nach demselben Schema anlegen. Eine weitere einfache Methode ist durch die interaktive Django-Konsole gegeben, die Sie durch den Kommandozeilenbefehl `python manage.py shell` aufrufen können. Um in der Django-Shell einen Artikel als neues `Produkte`-Objekt in Ihrer Datenbank anzulegen, genügt nun beispielsweise der folgende Code:

```
1 from produkte.models import Produkte
2
3 artikel_1 = Produkte(bezeichnung='Kaffeetasse', artnr=945687261,
4 beschreibung='Kaffeetasse in Schwarz, groß', preis=7.99)
5 artikel_1.save()
```

## 17 Webseiten mit Django erstellen: Ein Einstieg

Das Objekt wird erst durch die `save`-Methode in der Datenbank abgelegt, was hier demselben Schema entspricht, wie wenn man durch einen `INSERT`-Befehl in einer SQLite-Datenbank einen neuen Eintrag erstellt. Auch Änderungen an Einträgen bedürfen der `save`-Methode und folgen dann dem Schema des `UPDATE`-Befehls für Datenbanken. Eine Änderung des Preises von `artikel_1` lässt sich beispielsweise mit

```
1 artikel_1.preis = 7
2 artikel_1.save()
```

vornehmen, während durch `artikel_1.delete()` der gesamte Eintrag gelöscht wird. Der `print`-Befehl für ein bestimmtes Objekt gibt denjenigen String-Inhalt aus, den Sie in der Klassenmethode `__str__` des Modells angegeben haben. In diesem Fall ergäbe `print(artikel_1)` also den Output `Kaffeetasse`, da durch die definierte `__str__`-Methode die Bezeichnung des jeweiligen Artikels zurückgegeben wird.

Eine Übersicht über alle vorhandenen Objekte können Sie sich durch `Produkte.objects.all()` anzeigen lassen. Durch `exit()` können Sie die Shell wieder verlassen, um wie zuvor Django-Befehle in der Kommandozeile zu tätigen.

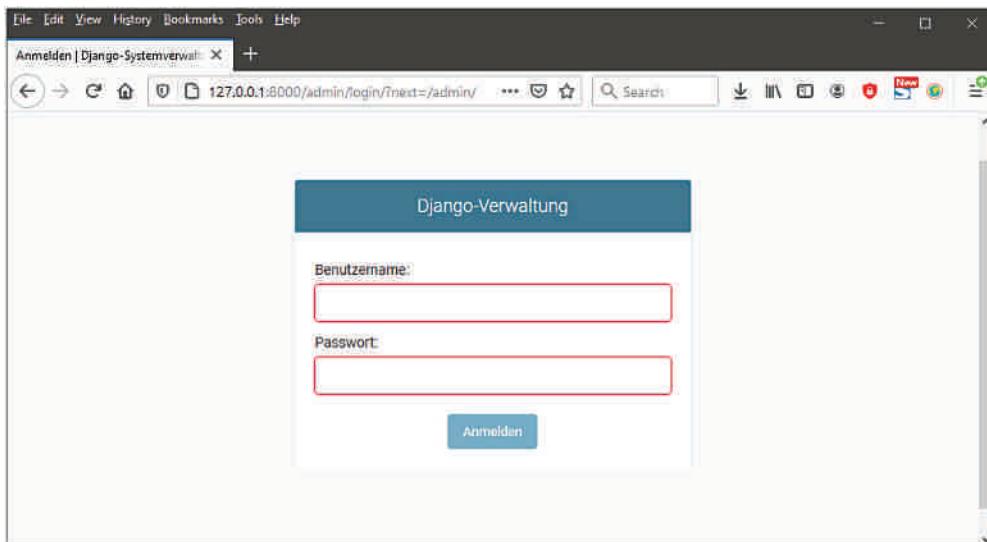
Mehr Informationen zu unterschiedlichen Datenbankabfragen finden Sie auf den Seiten der Django-Dokumentation.

Als Nächstes zeigen wir Ihnen eine sehr praktische Funktionalität in Django, mithilfe derer Sie die Einträge aus einer Datenbank ganz einfach verwalten und anpassen können.

### 17.5.1 Die admin-Seite

In Abschnitt 17.2. wurde im Zusammenhang mit den unterschiedlichen Pfaden einer Webseite bereits die admin-Seite genannt. Hierbei handelt es sich um eine sehr nützliche Verwaltungs-Funktionalität, mithilfe derer Sie die Daten Ihrer Webseite einsehen können. Zudem lassen sich Webseite-Daten über die admin-GUI aktualisieren, neu erzeugen oder löschen.

Um auf die admin-Seite zu gelangen, geben Sie als Webserver-Adresse <http://127.0.0.1:8000/admin> ein, nachdem Sie den Entwicklungsserver gestartet haben. Die Django-Verwaltung mit den zwei Eingabefeldern *Benutzername* und *Passwort* wird angezeigt.



**Abb. 17.14** Anzeige der admin-Seite unter [localhost:8000/admin](http://localhost:8000/admin)

Um sich hier erstmalig einloggen zu können, müssen Sie zunächst einen neuen Nutzer anlegen. Halten Sie dazu im Kommandozeilenfenster durch **Strg + C** den Entwicklungsserver an und erstellen Sie anschließend durch

```
1 python manage.py createsuperuser
```

ein neues Nutzerkonto. Sie werden nun aufgefordert, Nutzernamen, E-Mailadresse und Passwort anzugeben. Starten Sie den Server durch `python manage.py runserver` anschließend erneut und navigieren Sie in Ihrem Browser zur admin-Seite. Nun können Sie Ihre gewählten Anmeldedaten eingeben. Nach dem Login-Vorgang wird Djangos Standard-Verwaltungsseite angezeigt, auf der Sie unterschiedliche Aktionen veranlassen können.

Beispielsweise können Sie unter der Option **Benutzer** die Daten vorhandener Benutzer – darunter die Ihres soeben erstellten Superuser-Kontos – einsehen und ändern sowie neue Nutzer mit festlegbaren Zugangsrechten hinzufügen. Klicken Sie hierfür als Superuser auf **Hinzufügen** und legen Sie die neuen Nutzer an. Danach können Sie für jeden Nutzer unterschiedliche Rechte festlegen. Das Gleiche können Sie unter **Gruppen** übrigens auch für ganze Nutzergruppen tun.

Außerdem können Sie von innerhalb der admin-Seite Einträge zu Datenbanken hinzufügen und darauf die bekannten CRUD-Operationen ausführen. Hierfür müssen Sie das in Ihrem Projekt angelegte Datenbank-Modell jedoch für die Django-Verwaltungsseite registrieren. Dazu sollte in der Datei `admin.py` für die Modell-Klasse innerhalb der jeweiligen App – hier `Produkte` – folgender Inhalt stehen:

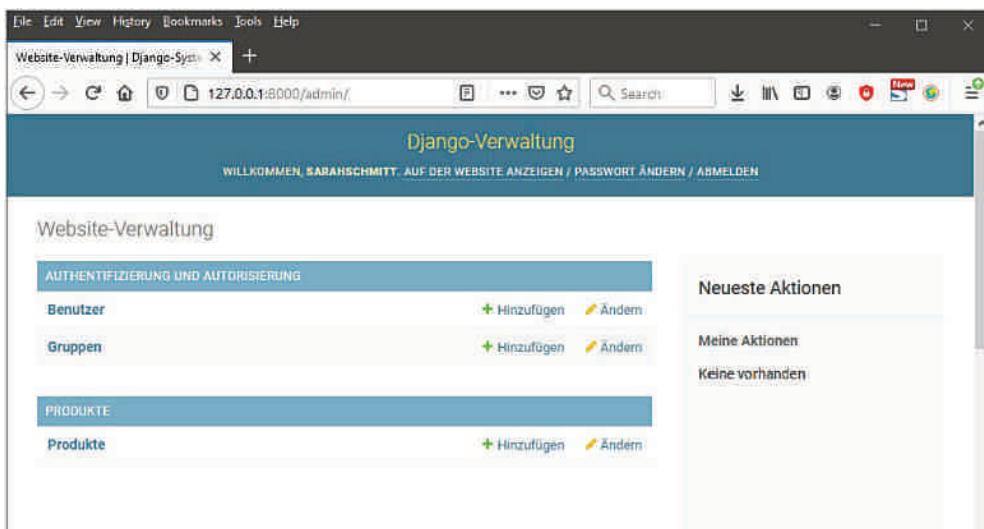
## 17 Webseiten mit Django erstellen: Ein Einstieg

```

1 from django.contrib import admin
2 from .models import Produkte
3
4 admin.site.register(Produkte)

```

Wenn Sie die Verwaltungsseite aktualisieren, sollte unterhalb des Menüs **Authentifizierung und Autorisierung** nun ebenfalls eine weitere Kategorie für unsere Produkte-App angezeigt werden. Klicken Sie darauf – wenn Sie im vorigen Abschnitt Instanzen in Ihrer Datenbank angelegt haben, sollten diese hier nun sichtbar sein. Sie können an dieser Stelle weitere Artikel hinzufügen, indem Sie auf **Hinzufügen** klicken und in den Feldern, die unseren gewählten Klassenattributen entsprechen, die jeweiligen neuen Eingaben machen. Legen Sie hier nun probeweise zwei oder drei Artikel an. Sie können die Angaben zu den Artikeln auch nachträglich ändern, indem Sie unter **Produkte** dasjenige Objekt auswählen, dessen Informationen Sie anpassen möchten. Hier besteht ebenfalls die Möglichkeit, bereits angelegte Produkt-Instanzen wieder zu löschen oder weitere neue Einträge anzulegen. Bei größeren Datenmengen können Sie Ihr Modell auch dahingehend abändern, dass Einträge in der admin-Seite alphabetisch sortiert oder durchsucht werden können.



**Abb. 17.15** Anzeige der Django-Verwaltungsseite mit der Produkte-Datenbank

Wie Sie sehen, ist die admin-Seite sehr praktisch, um Nutzerdaten zu verwalten und Datenbankeinträge direkt hierüber anstatt über Python-Code zu handhaben.

Glückwunsch! Sie haben nun die wichtigsten Grundlagen zur Erstellung von Webseiten mit Django erlernt. Bislang haben wir unsere Webseite jedoch nur auf unserem lokalen Entwicklungsserver laufen lassen. Wie also stellt man ein fertiges Django-Webseitenprojekt online, sodass es im Internet gefunden werden kann? Dies werden wir im letzten Teil dieses Kapitels kurz erläutern.

## 17.6 Eine Webseite online stellen

Sobald Sie Ihre Webseite fertiggestellt haben, brauchen Sie eine reservierte Domain-Adresse, unter der man Ihre Webseite live im Internet finden kann. Hierfür müssen Sie bei einem geeigneten Anbieter – beliebt sind beispielsweise *Ionos*, *Strato* oder *United Domains* – gegen ein monatliches oder jährliches Entgelt einen freien Domain-Namen registrieren lassen.

Zusätzlich zu Ihrer registrierten Domain benötigen Sie einen Server, bei dem Sie die Dateien Ihres Webseitenprojekts hochladen können. Der Webhosting-Service *hostet* Ihre Webseite: Er stellt den Speicherplatz zur Verfügung, sorgt durch eine permanente Internetverbindung dafür, dass die Webseite jederzeit abrufbar ist, garantiert die Sicherheit und Aktualität aller Funktionalitäten und bietet zudem eine Benutzeroberfläche, über die Sie Ihre Internetpräsenz verwalten können. Einige der Serveranbieter sind kostenlos, während andere einen monatlichen oder jährlichen Beitrag verlangen, oftmals je nach Größe Ihrer Seite. Viele Domainanbieter bieten zusätzlich zur Webseiten-Reservierung gleichzeitig ein Webhosting-Paket an, sodass es oftmals eine gute Idee sein kann, beide Funktionalitäten über ein und denselben Anbieter laufen zu lassen.

Beliebte externe Serveranbieter sind dabei unter anderem *DigitalOcean*, *Heroku* oder *PythonAnywhere*. Die letzteren beiden sind für kleinere Webanwendungen mit wenigen Seitenaufrufen in der Regel kostenlos. Zudem haben Sie beispielsweise ebenfalls die Möglichkeit, Ihre Webseite ohne den Ankauf einer separaten Domain direkt auf [www.pythonanywhere.com](http://www.pythonanywhere.com) zu hosten. Gerade für Einsteiger ist dies meist eine gute und einfache Möglichkeit, ein Webseitenprojekt mit wenig Nutzeraufkommen erstmalig online zu stellen und in einer echten Live-Umgebung zu testen.

Eine weitere wichtige Funktionalität in Bezug auf das Hosten Ihres Django-Codes ist *GitHub*. Die meisten Programmierenden eröffnen früher oder später ein GitHub-Konto, um dort ihren Code abzulegen. Auf [www.github.com](http://www.github.com) können Sie Kopien Ihrer fertigen Projekte – für jegliche Programmiersprachen und Software-Anwendungen – in Form sogenannter *Repositories* hochladen und verwalten, etwa um sie einer breiteren Nutzerschaft zur Verfügung zu stellen, gemeinsam mit anderen daran zu arbeiten oder in ein größeres Gesamtprojekt zu integrieren. So können Sie auch den Code für Ihre Django-Webseite zunächst auf GitHub hochladen und stetig aktualisieren, um anschließend durch Ihren Serveranbieter darauf zuzugreifen und die Webseite unter der gewünschten Domain online zu stellen.

Mithilfe von GitHub können Sie zugleich den Versionsverlauf Ihrer Softwareentwicklung festhalten, sodass Sie bei Bedarf ebenfalls unterschiedliche Code-Versionen einsetzen können. Nachdem Sie ein GitHub-Konto erstellt und *Git* unter <https://git-scm.com/> heruntergeladen und installiert haben, können Sie sich über Ihre Kommandozeile bei Ihrem GitHub-Konto anmelden und das Git-Repository für Ihr aktuelles Projekt starten, wodurch Änderungen an Ihrem Code registriert werden können.

## 17 Webseiten mit Django erstellen: Ein Einstieg

Für die Liveschaltung Ihrer Webseite müssen Sie in der Datei `settings.py` zusätzlich einige Änderungen vornehmen. Meist legt man hierfür zudem eine neue, separate `settings.py`-Datei an. Unter anderem müssen Sie darin bei `DEBUG` auf `False` umstellen und unter `SECRET_KEY` zur kryptographischen Sicherung Ihrer Daten eine neue Angabe machen. Zudem müssen Sie unter `ALLOWED_HOSTS` die Daten des Servers angeben, den Sie damit autorisiert haben, Ihre Webseite zu hosten. Eventuell müssen Sie ebenfalls die Parameter für Ihre Datenbank abändern. Außerdem müssen Sie unter `STATIC_ROOT` dasjenige Verzeichnis angeben, in dem Sie Ihre statischen Dateien wie beispielsweise Bilddateien ablegen möchten. Je nachdem, welche Funktionalitäten Ihr Internetauftritt bereitstellen soll, kann es erforderlich sein, weitere Änderungen an den Einstellungen vorzunehmen.

## 17.7 Übung: Webseiten mit Django erstellen

Klingt *Django* nach diesem flotten Einstieg nicht wie Musik in Ihren Ohren? Sie sollten die grundlegenden Schritte zum Erstellen einer Webseite jedoch mindestens einmal selbst ausgeführt haben, um ein besseres Verständnis des Aufbaus sowie mehr Sicherheit im Erstellen von Webseiten zu erhalten. Legen Sie daher der Übung halber zunächst selbst einmal eine neue Webseiten-URL sowie eine eigene neue App an, mit denen Sie in den folgenden Übungsaufgaben arbeiten. Mithilfe der Aufgaben können Sie danach Ihr Wissen verfestigen und sich weitere Fertigkeiten aneignen. Die Bearbeitung erfordert eine zusätzliche Internetrecherche.

### Übungsaufgaben

1. Suchen Sie in der Django-Dokumentation nach einem Tag zur Darstellung von Zeitangaben und implementieren Sie dieses auf einer Webseite, um sich das heutige Datum und die aktuelle Uhrzeit anzeigen zu lassen.
2. Finden Sie in der Django-Dokumentation einen passenden Feld-Klassentyp für Ihr Datenbankmodell, der das Hinzufügen von Bilddateien erlaubt. Legen Sie das neue Modell anschließend an. Vergessen Sie dabei nicht, die nötigen Migrationsbefehle auszuführen.

**Tipp:** Um Bilddateien verwenden zu können, müssen Sie zunächst mit `pip` das externe Paket `pillow` installieren.

3. Lassen Sie sich die in Aufgabe 2 angelegten Objekte anschließend auf der Webseite durch Variablen anzeigen.

**Tipp:** Recherchieren Sie im Internet, welche zusätzlichen Einstellungen hierfür in `views.py` und `urls.py` nötig sind.

## 17 Webseiten mit Django erstellen: Ein Einstieg

### Lösungen

- Wir implementieren die Zeitangabe im Template `uebung.html` unserer neu angelegten Uebung-App, das unter der URL `/uebung` verwendet wird. Die Datum- und Zeitangabe lässt sich innerhalb des Templates im `<body>`-Tag ganz einfach durch die Zeile

```
1 <p>Heutiges Datum: { % now "d. F Y, H:i" %} Uhr</p>
```

realisieren. Eine Beschreibung der Funktionsweise des eingebauten Template-Tags `now` finden Sie unter <https://docs.djangoproject.com/en/3.0/ref/templates/builtins/#now>. Es gibt in Django weitere Formatierungsmöglichkeiten zur Darstellung von Zeitangaben. So lassen sich diese beispielsweise auch über Filter oder innerhalb eines Modells über vordefinierte Feldtypen implementieren.

- Der gewünschte Feld-Klassentyp heißt `ImageField` und lässt sich in der Datei `models.py` der Uebung-App folgendermaßen als Objektattribut implementieren:

```
1 from django.db import models
2
3 class Uebung(models.Model):
4     bezeichnung = models.CharField(max_length=50)
5     bild = models.ImageField(upload_to='bilder/')
6
7     class Meta:
8         verbose_name_plural = "Uebung"
9
10    def __str__(self):
11        return self.bezeichnung
```

Unter `upload_to` steht das Verzeichnis, in dem Sie die Bilddateien auf Ihrem Computer speichern möchten. Hierzu sind in `settings.py` zudem folgende zusätzliche Einträge nötig, die Sie unterhalb des Eintrags `STATIC_URL` einfügen können:

```
1 MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
2
3 MEDIA_URL = '/media/'
```

Sie müssen zuerst das `pillow`-Paket installieren, ehe der Entwicklungsserver gestartet werden kann. Installieren Sie also das `pillow`-Paket durch den Kommandozeilenbefehl `pip3 install pillow`. Anschließend können Sie durch

```
1 python manage.py makemigrations uebung
2 python manage.py migrate
```

die Migrationsdatei erstellen und auf die Datenbank anwenden. Starten Sie jetzt den Entwicklungsserver durch den üblichen `runserver`-Befehl. In der Django-Verwaltungsseite sollte das Modell nun sichtbar sein. Neue Einträge darin lassen sich wie in 17.5.1. beschrieben anlegen. Dies beinhaltet nun auch die Option, eine

## 17.7 Übung: Webseiten mit Django erstellen

Bilddatei für jeden Eintrag auszuwählen, die im ausgewählten Ordner abgelegt wird.

3. Nehmen Sie in der Datei `views.py` der Uebung-App den zusätzlichen Import `from .models import Uebung` vor und fügen Sie der `uebung`-Funktionsdefinition die Zeile `eintraege = Uebung.objects` hinzu. Hierdurch werden alle Datenbankinträge ausgelesen, damit sich diese in unserem HTML-Code als Python-Objekte verwenden lassen. Geben Sie im Dictionary der `render`-Funktion anschließend `{'eintraege': eintraege}` an. Die abgeänderte `views.py`-Datei beinhaltet somit den folgenden Code:

```

1 from django.shortcuts import render
2 from .models import Uebung
3
4 def uebung(request):
5     eintraege = Uebung.objects
6     return render(request, 'uebung.html', {'eintraege': eintraege})

```

Ändern Sie nun den Code in `urls.py` in der Uebung-App wie folgt:

```

1 from django.urls import path
2 from . import views
3 from django.conf import settings
4 from django.conf.urls.static import static
5
6 urlpatterns = [
7     path('uebung/', views.uebung, name='uebung'), # neuen Pfad angeben
8 ] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

```

In der HTML-Datei `uebung.html` lassen sich die Einträge jetzt folgendermaßen referenzieren und werden als Liste auf der Webseite angezeigt:

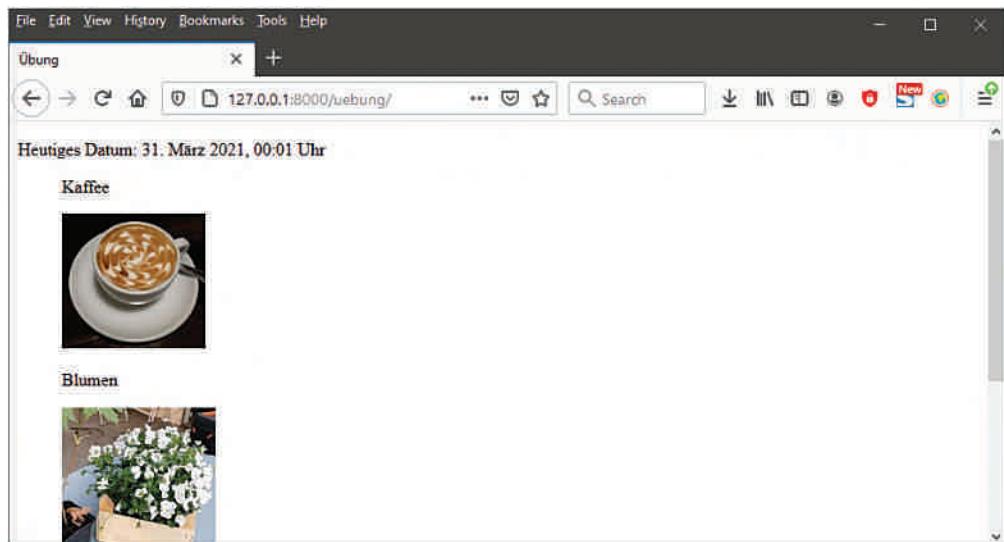
```

1 <ul>
2     {% for eintrag in eintraege.all %}
3         <p>{{ eintrag.bezeichnung }}</p>
4         
5     {% endfor %}
6 </ul>

```

Nach diesem vereinfachten Schema lassen sich auf der Webseite beliebig viele Objekte bzw. Attribute anzeigen.

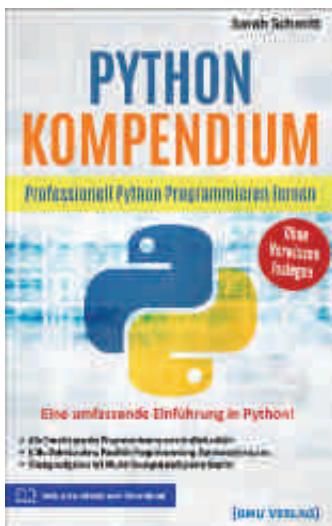
## 17 Webseiten mit Django erstellen: Ein Einstieg



**Abb. 17.16** Anzeige von in Django-admin angelegten Einträgen in einer for-Schleife

Downloadhinweis

Alle Programmcodes aus diesem Buch sind zum Download verfügbar. Dadurch müssen Sie sie nicht abtippen:  
<https://bmu-verlag.de/python-kompendium/>



Außerdem erhalten Sie die eBook-Ausgabe zum Buch kostenlos auf unserer Website:



<https://bmu-verlag.de/python-kompendium/>  
**Downloadcode:** siehe Kapitel 18

## Kapitel 18

# Datenanalyse mit Python

*Datenanalyse* ist ein recht weit gefasster Begriff. Allgemein bezeichnet er das computergestützte Arbeiten mit größeren Datenmengen und das Extrahieren von Informationen daraus mithilfe statistischer Methoden. Dabei gibt es verschiedene Ansätze und Techniken, wie sich neues Wissen aus Datenbeständen gewinnen lässt, sowie ein sehr breites Spektrum möglicher Einsatzgebiete – vom Gesundheitswesen, Ingenieur- und Rechtswissenschaften über Meinungsforschung bis hin zum Wirtschaftsbereich.

Nicht ohne Grund werden Daten heutzutage häufig als das „neue Gold“ unseres digitalen Zeitalters bezeichnet: Denn mit leistungsfähigeren Rechnern, einer zunehmenden Digitalisierung unseres Alltags und den damit verbundenen stetig wachsenden Datenmengen – auch *Big Data* genannt – steigen auch die Möglichkeiten, daraus äußerst lukrative Erkenntnisse über unser Verhalten, unsere Vorlieben oder unsere Kaufentscheidungen zu ziehen – und diese somit zu beeinflussen.

Wenig verwunderlich ist daher auch die Tatsache, dass Datenanalyse für mehr und mehr Menschen einen wichtigen Bestandteil ihrer Arbeit ausmacht. In vielen Tätigkeitsbereichen ist das Beherrschene grundlegender Datenanalyse-Techniken eine gern gesehene und gefragte Qualifikation, die in der Regel zudem sehr gut entlohnt wird.

Techniken der Datenanalyse umfassen dabei nicht nur die statistische Modellierung von Daten, sondern auch das Strukturieren und Ordnen derselben. Oftmals liegen diese nämlich lediglich in Form sogenannter Rohdaten vor, die zunächst in eine bereinigte, brauchbare Form gebracht werden müssen, ehe sich damit statistische Analysen durchführen lassen. Ebenso kann es erforderlich sein, die Daten nicht nur tabellarisch, sondern auch visuell darzustellen um wesentliche Zusammenhänge besser zu veranschaulichen.

Sicherlich überrascht es Sie nun nicht, dass Python ein für wissenschaftliche Berechnungen bestens geeignetes und sehr beliebtes Tool ist. Möchten Sie Python in erster Linie für Datenanalysezwecke einsetzen, ist tiefgründiges Wissen dieser Programmiersprache Ihrem Vorhaben in jedem Fall zuträglich. In Verbindung mit Python lässt sich eine Vielzahl an Paketen verwenden, die explizit für Datenanalysezwecke entwickelt wurden. Zu den beliebtesten darunter zählen *pandas*, *NumPy* und *SciPy*, zur Datenvisualisierung verwendet man insbesondere meist *Matplotlib* oder *Seaborn*.

Im nächsten Abschnitt werden wir Ihnen *Anaconda*, ein beliebtes Datenanalyse-Softwaretool, vorstellen. Danach werden wir die Datenanalyse-Bibliotheken *pandas* und *NumPy* sowie für Visualisierungszwecke die Bibliothek *Seaborn* behandeln. Pandas,

## 18.1 Anaconda installieren und JupyterLab starten

NumPy und andere nützliche Datenanalyse-Pakete werden häufig innerhalb eines Programms kombiniert, da sie jeweils spezifische Funktionalitäten für unterschiedliche Anwendungsbereiche bereithalten. So basiert die am häufigsten verwendete pandas-Datenstruktur, das sogenannte *Dataframe*, etwa ähnlich wie ein Excel-Spreadsheet auf Tabellen, während sich mit NumPys mehrdimensionaler *Array*-Datenstruktur fortgeschrittenere Matrizen-Operationen durchführen lassen.

### 18.1 Anaconda installieren und JupyterLab starten

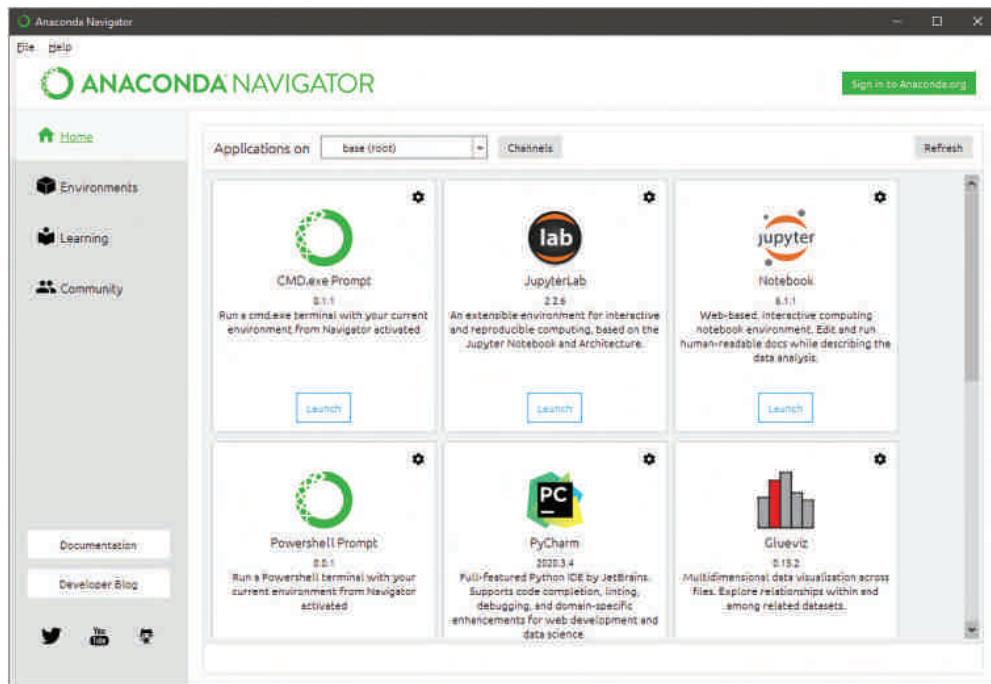
Anaconda ist eine plattformunabhängige Software-Distribution für Datenanalyse und wissenschaftliches Rechnen, die die beliebtesten Python-Pakete hierfür bereits enthält. Die Installation von Anaconda benötigt etwa 3 GB an Speicherplatz und dauert ca. 15 Minuten. Möchten Sie Zeit und Speicherplatz sparen oder nur sehr begrenzt Datenanalyse betreiben, können Sie anstelle von Anaconda die kleinere Miniconda-Distribution verwenden, mithilfe derer sich einzelne Pakete gezielt auswählen und installieren lassen. Der Einfachheit und Möglichkeit zur tiefergehenden Beschäftigung mit Datenanalyse halber werden wir hier jedoch die Anaconda-Vollversion nutzen.

Mit der Anaconda-Distribution werden automatisch 250 Pakete installiert. Darüber hinaus haben Sie mit der Paket- und Umgebungsverwaltung *conda* die Option, auf über 1.000 weitere Pakete aus dem Anaconda-Repository zuzugreifen. Anaconda lässt sich zudem in Verbindung mit der Programmiersprache *R* verwenden, die ebenfalls gerne für statistische Berechnungen genutzt wird. Anaconda beinhaltet auch den Kommandozeileninterpreter *IPython* und dessen webbasierte Benutzerschnittstelle *JupyterLab*, die immer häufiger für Datenanalysezwecke und wissenschaftliches Rechnen zum Einsatz kommt.

Um Anaconda zu installieren, müssen Sie zunächst unter <https://www.anaconda.com/products/individual> die Installationsdateien für Ihr Betriebssystem und Ihre Python-Version herunterladen. Wenn der Download fertig ist, klicken Sie auf die Datei. Wählen Sie sodann im Installationsprogramm die gewünschten Einstellungen und den Installationspfad aus. Im Fenster **Advanced Options** sollten Sie möglichst keine Änderungen vornehmen. Bestätigen Sie anschließend mit **Install**.

Wenn Sie Windows als Betriebssystem verwenden, sollte sich nun im Startmenü ein Eintrag für die Desktop-GUI *Anaconda Navigator* befinden. Auf macOS (mit dem *.pkg*-Installer) sollte das entsprechende Symbol in Ihrem Launchpad sichtbar sein. Auf Linux oder macOS (mit dem *.sh*-Installer) können Sie Anaconda Navigator aus einem Terminalfenster durch den Befehl `anaconda-navigator` aufrufen. Mithilfe des Anaconda Navigator lassen sich nun unterschiedliche Anwendungen starten sowie *conda*-Pakete und -umgebungen verwalten. Alternativ zum Navigator können Sie ebenfalls den *Anaconda Prompt* verwenden, mithilfe dessen sich zusätzlich zur Funktionalität eines Kommandozeilenfensters Anaconda- und *conda*-Befehle ausführen lassen.

## 18 Datenanalyse mit Python

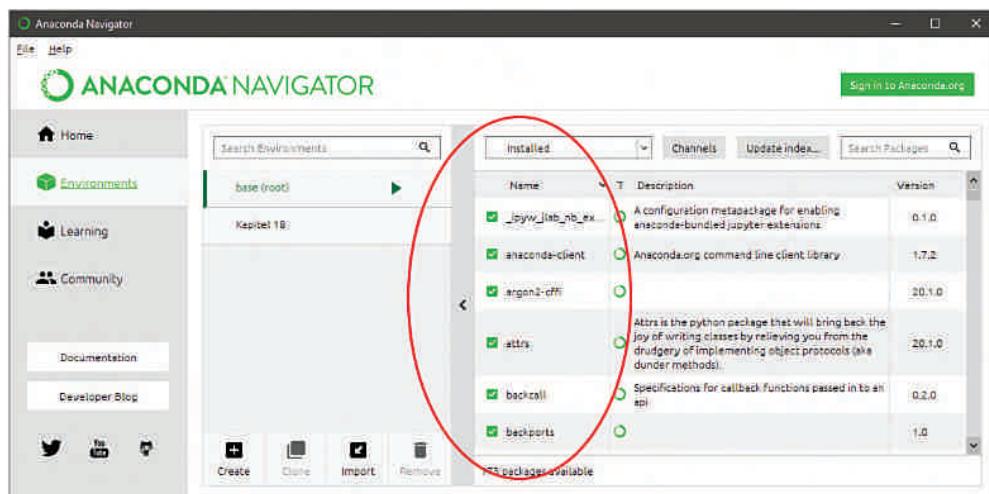


**Abb. 18.1** Erste Ansicht des Anaconda Navigator

In der Startansicht des Anaconda Navigator sehen Sie im **Home**-Tab alle bereits installierten oder verfügbaren Anwendungen. Wie Sie hier sehen, unterstützt PyCharm ebenfalls die Verwendung von Anaconda.

Im **Environments**-Tab sehen Sie unter **Installed** eine Übersicht über alle derzeit installierten Pakete, die Sie bei Bedarf aktualisieren, ändern oder löschen können. Im Suchbereich rechts davon können Sie ebenfalls nach bislang nicht installierten Paketen innerhalb der gewählten Umgebung suchen und diese installieren.

## 18.1 Anaconda installieren und JupyterLab starten



**Abb. 18.2** Paketverwaltung im Anaconda Navigator

Im **Learning**-Tab finden Sie unter anderem zusätzliches Trainingsmaterial mit Zugang zu den Dokumentationen unterschiedlicher Tools sowie weitere Informationen zum Navigator.

Sie sollten sich an dieser Stelle ein wenig mit den Möglichkeiten innerhalb des Anaconda Navigator vertraut machen. Wenn Sie für die nächsten Codebeispiele weiterhin PyCharm verwenden möchten, müssen Sie in den PyCharm-Einstellungen die nötigen Änderungen hierfür vornehmen, die Sie in der Anaconda-Dokumentation unter <https://docs.anaconda.com/anaconda/user-guide/tasks/pycharm/> beschrieben finden.

Wir werden im Folgenden *JupyterLab*, den flexibleren und mächtigeren Nachfolger des klassischen *Jupyter Notebook*, verwenden. Bei JupyterLab handelt es sich um ein sehr aktuelles Tool, das insbesondere für Anfänger bestens geeignet ist und das sich ebenfalls in Kombination mit C#, Java, JavaScript, PHP und vielen weiteren Sprachen einsetzen lässt. Mithilfe von JupyterLab lassen sich Datenanalyseprogramme direkt im Webbrowser durch sogenannte *Notebooks* erstellen, in die sich neben dem gewöhnlichen Code auch andere Dateien wie Texte, Videos und Bilder sowie Links einbauen lassen. Die Codezeilen lassen sich dabei zudem sehr praktisch zeilenweise ausführen.

Klicken Sie im **Home**-Tab des Anaconda Navigators nun also auf den entsprechenden Eintrag für JupyterLab und warten Sie, bis sich das Browserfenster geöffnet hat.

## 18 Datenanalyse mit Python

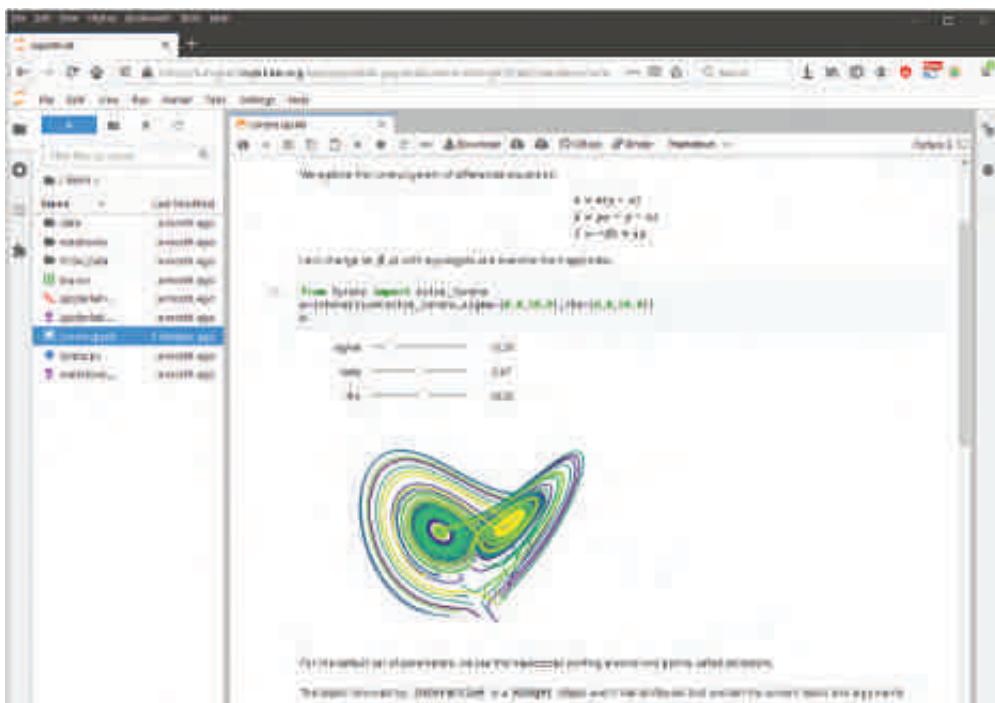


Abb. 18.3 Eine Beispielansicht von JupyterLab im Browser

Unter <https://jupyterlab.readthedocs.io/en/stable/> können Sie mehr über die zahlreichen Funktionalitäten von JupyterLab nachlesen. Im Folgenden werden wir in JupyterLab mit drei sehr beliebten Datenanalyse-Paketen für Python arbeiten: pandas, NumPy und Seaborn. Darüber hinaus gibt es jedoch noch viele weitere Pakete, die jeweils für spezifische Anwendungsbereiche entwickelt wurden oder teilweise auf den oben genannten aufbauen.

### 18.2 pandas

Pandas ist ein praxisorientiertes Datenanalyse-Paket, das insbesondere für die Arbeit mit großen Datenmengen geeignet ist. Es verfügt über verschiedene Datenstrukturen zur Handhabung der Daten, worunter die gängigste, das *Dataframe*, sehr häufig im Bereich der Datenanalyse und in anderen datenintensiven Bereichen Anwendung findet. Eine weitere grundlegende pandas-Datenstruktur ist die *Series*-Datenstruktur.

Auf den nächsten Seiten werden wir als Beispiel für die Arbeit mit Daten die folgende Tabelle über den Inhalt der bisherigen Kapitel dieses Buchs heranziehen, wobei wir die Kapitelüberschriften der besseren Darstellung halber ihrem Inhalt entsprechend abgekürzt haben:

	Kapiteltitel	Wörter	Teilkapitel
1	Einleitung	3027	5
2	Vorbereitung	2150	4
3	Der interaktive Modus	1031	2
4	Programme in eine Datei schreiben	2333	4
5	Variablen	3090	6
6	Datenstrukturen	6202	8
7	Entscheidungen treffen	2785	5
8	Schleifen	3203	6
9	Funktionen	3647	6
10	Module aus der Standardbibliothek	3338	3
11	Objektorientierte Programmierung	8735	7
12	Fehler und Ausnahmen	4985	6
13	Textdateien zur Datenspeicherung	2311	4
14	Datenbanken	4377	5
15	Desktop-GUIs	8928	7
16	E-Mails	2681	4
17	Django	8197	7

In der ersten Zeile stehen dabei die Spalten-Labels und in der ersten Spalte der jeweilige Zeilenindex, der die Kapitelzahl bezeichnet. Selbstverständlich werden Sie in den meisten Datenanalyseanwendungen weitaus größere Datenmengen verwenden; diese reduzierte Variante soll uns jedoch als einfaches Beispiel genügen.

### 18.2.1 Ein pandas-Dataframe erstellen

Dataframes ermöglichen das Speichern und Bearbeiten von Daten in zweidimensionaler Form, wie Sie dies von üblichen, aus Zeilen und Spalten bestehenden Tabellen gewohnt sind. Die Spalten können dabei jeweils unterschiedliche Datentypen enthalten. Zeilen und Spalten lassen sich zudem mit einem Index bzw. Labels versehen, die

## 18 Datenanalyse mit Python

in unserer obigen Buch-Tabelle beispielsweise den Kapitelzahlen und weiteren kategorischen Bezeichnungen entsprechen.

Um mit pandas arbeiten zu können, müssen Sie zunächst die Bibliothek importieren. Geben Sie dazu den entsprechenden `import`-Befehl in Ihr geöffnetes JupyterLab-Notebook ein:

```
1 import pandas as pd
```

Das Alias `pd` dient hier der Abkürzung.

Nun gibt es mehrere Möglichkeiten, wie Sie die Daten aus unserer Tabelle als Dataframe abspeichern können. Beispielsweise können Sie sie in einer Excel-Tabelle anlegen und durch den `read`-Befehl importieren. Diese Lese- und Schreiboptionen werden wir im nächsten Unterkapitel erläutern.

Eine zweite Möglichkeit ist es, die Daten direkt einzugeben und daraus ein Dataframe zu erstellen, was insbesondere bei kleineren Datensätzen die praktikablere Variante ist. Die Daten lassen sich dabei unter anderem als Liste, Tupel oder Dictionary definieren. Wir geben unsere Buch-Daten also folgendermaßen in einem Dictionary an:

```
1 buchdaten = {'thema' : ['Einleitung', 'Vorbereitung', 'Der interaktive
2 Modus', 'Programme in eine Datei schreiben',
3 'Variablen',
4 'Datenstrukturen', 'Entscheidungen treffen',
5 'Schleifen',
6 'Funktionen', 'Module aus der Standardbibliothek',
7 'Objektorientierte Programmierung', 'Fehler und
8 Ausnahmen', 'Textdateien zur Datenspeicherung',
9 'Datenbanken', 'Desktop-GUIs', 'E-Mails', 'Django'],
10 'woerter' : [3027, 2150, 1031, 2333, 3090, 6202, 2785, 3203, 3647,
11 3338, 8735, 4985, 2311, 4377, 8928, 2681, 8197],
12 'teilkapitel' : [5, 4, 2, 4, 6, 8, 5, 6, 6, 3, 7, 6, 4, 5, 7, 4,
13 7]
14 }
15
16 kapitel = range(1, 18)
```

Die `buchdaten`-Variable enthält dabei die Labels und Daten aller drei vorhandenen Spalten. Der Zeilenindex, in diesem Fall die Kapitelnummerierung, wird in der `kapitel`-Variablen automatisch durch die `range`-Funktion definiert. Sie können hier jedoch auch nicht aufeinanderfolgende Zahlen oder eine beliebige andere Beschriftung angeben.

Aus diesen Informationen lässt sich nun ein Dataframe erstellen:

```
1 df = pd.DataFrame(data=buchdaten, index=kapitel)
2 df
```

Sofern Sie die Codezeilen in der obenstehenden Menüauswahl durch `Code` (anstelle von `Markdown` oder `Raw`) als solche gekennzeichnet haben, gibt die Zeile `df` im JupyterLab-Notebook die im Dataframe angelegten Daten aus. Anstelle des in IDEs üblichen `print`-Befehls genügt für die Ausgabe in JupyterLab – ähnlich wie in der Python-Shell – also lediglich die Nennung des Dataframe-Namens. Sie können für die Anzeige jedoch ebenfalls wie gewohnt den `print`-Befehl verwenden. Der Code lässt sich in der Menüauswahl sodann durch Klicken auf das *Run*-Symbol (▶) ausführen.

	kapiteltitel	woerter	teilkapitel
1	Einleitung	3027	5
2	Vorbereitung	2150	4
3	Der interaktive Modus	1031	2
4	Programme in eine Datei schreiben	2333	4
5	Variablen	3090	6
6	Datenstrukturen	6202	8
7	Entscheidungen treffen	2785	5
8	Schleifen	3203	6
9	Funktionen	3647	6
10	Module aus der Standardbibliothek	3338	3
11	Objektorientierte Programmierung	8735	7
12	Fehler und Ausnahmen	4985	6
13	Textdateien zur Datenspeicherung	2311	4
14	Datenbanken	4377	5
15	Desktop-GUIs	8928	7
16	E-Mails	2681	4
17	Django	8197	7

Abb. 18.4 Anzeigen des Dataframes in JupyterLab

Oftmals kommt es vor, dass ein Datenbestand nicht vollständig ist, sodass einige Felder innerhalb einer Tabelle etwa fälschlich ausgefüllt oder leer sind. Dann müssen die Daten zunächst bereinigt und vorbereitet werden, bevor sich damit weiterarbeiten lässt.

## 18 Datenanalyse mit Python

Pandas hält zur Handhabung unvollständiger oder fehlerhafter Daten eine starke Funktionalität parat. Um dies zu untersuchen, fügen wir unserem Dataframe zunächst folgendermaßen eine neue Zeile mit den bislang unvollständigen Daten für das 18. Buchkapitel hinzu:

```
1 neueskapitel = pd.DataFrame({"kapiteltitel": ["Datenanalyse"]}, index=[18])
2 df2 = df.append(neueskapitel)
3 df2
```

Die fehlenden Daten werden in der Ausgabe von `df2` durch das Kürzel `NaN` angezeigt. Fehlende Daten im Dataframe können mitunter ebenfalls durch die Werte `np.nan`, `None`, `<NA>` oder `NaT` gekennzeichnet sein.

Die fehlenden Einträge lassen sich nun durch den Befehl `df2.isna()` auffinden, wodurch diese Felder in der Tabelle dem Wahrheitswert `True` zugeordnet werden, während vorhandene Einträge die Zuordnung `False` erhalten. Die Funktion `notna()` nimmt die umgekehrte Zuordnung vor.

14	<code>False</code>	<code>False</code>	<code>False</code>
15	<code>False</code>	<code>False</code>	<code>False</code>
16	<code>False</code>	<code>False</code>	<code>False</code>
17	<code>False</code>	<code>False</code>	<code>False</code>
18	<code>False</code>	<code>True</code>	<code>True</code>

Abb. 18.5 Fehlende Werte ausweisen

Mitunter sind die Felder einer Tabelle nicht leer, sondern enthalten fehlerhafte Zeichen wie `?`, `/` oder `--`. Hier besteht die Möglichkeit, die bekannten fehlerhaften Zeichen in einer Liste zu speichern und sodann dem Parameter `na_values` zuzuordnen, um sie somit als unvollständige Werte in pandas auszuweisen. Alternativ dazu lassen sich die Zeichen mithilfe der `replace()`-Funktion in entsprechende `NaN`-Werte überführen.

Ebenso kann es praktisch sein, `NaN`-Werte durch einen gewünschten Eintrag, beispielsweise den Durchschnitt oder einen String, zu ersetzen. Hierfür kommt die Funktion `fillna()` zum Einsatz. Durch den folgenden Code werden so beispielsweise alle `NaN`-Einträge mit dem String `fehlender Wert` versehen:

```
1 df.fillna("fehlender Wert")
```

Zeilen oder Spalten, die als `NaN` ausgezeichnete fehlende Werte enthalten, lassen sich schließlich mithilfe der Funktion `dropna()` entfernen.

### 18.2.2 Daten ein- und auslesen

In den Kapiteln 13 und 14 haben Sie bereits einige Optionen der Datenbehandlung und Datenspeicherung in Textdateien und Datenbanken kennen gelernt. Mit pandas können Sie Daten aus Excel-, .csv-, SQL- und vielen anderen Dateiformaten einlesen sowie umgekehrt in pandas erstellte Daten in diese Dateiformate schreiben und auf Ihrem Computer abspeichern.

Den df-Dataframe können wir somit ganz einfach durch

```
1 df.to_csv('buchdaten.csv')
```

in das aktuelle Anaconda-Arbeitsverzeichnis schreiben, das sich normalerweise in Ihrem Nutzerordner befindet, hier etwa *C:\Users\Sarah*. In diesem Verzeichnis wurde nun eine .csv-Datei mit den Buchdaten angelegt. Beim .csv-Format handelt es sich um ein reines Textformat, das sehr häufig für das Speichern großer Datenmengen verwendet wird. Die Werte der einzelnen Felder in der Textdatei sind dabei standardmäßig durch Kommata getrennt – daher der Name **comma separated values (.csv)** – und entsprechen den jeweiligen Zeilenwerten der Tabelle.

Befindet sich umgekehrt eine .csv-Datei in Ihrem Arbeitsordner, die Sie als Dataframe in JupyterLab verwenden möchten, so können Sie hierfür die Methode `read_csv()` verwenden:

```
1 pd.read_csv('buchdaten.csv', index_col=0)
```

Das letzte Argument in der Klammer gibt an, dass sich der gewünschte Zeilenindex in der ersten Spalte der Datei befindet.

Auf die gleiche Weise lassen sich mit dem Befehl `to_excel()` Excel-Tabellen erstellen, etwa durch

```
1 df.to_excel('buchdaten.xlsx')
```

sowie mit `read_excel()` Daten aus bereits vorhandenen Excel-Dateien lesen:

```
1 df = pd.read_excel('buchdaten.xlsx', index_col=0)
```

Je nach verwendeter IDE und gewünschter Funktionalität müssen Sie hierfür unter Umständen einige der zusätzlichen Pakete *xlwt*, *openpyxl*, *XlsWriter* oder *xlrd* installieren. Nach demselben Schema lassen sich ebenfalls andere Dateiformate wie *JSON*, *HTML* oder *SQL* lesen und schreiben.

## 18 Datenanalyse mit Python

### 18.2.3 Ein Dataframe bearbeiten

Auf die Zeilen- und Spaltenanzahl eines Dataframes lässt sich durch das `shape`-Attribut zugreifen, dessen Output ein Tupel mit der Zeilenanzahl als erstem Element und der Spaltenanzahl als zweitem Element ist. Die Ausgabe von `df.shape` ist in unserem Beispiel somit `(17, 3)` – die Indexspalte wird nicht mitgezählt. Insbesondere nach dem Einlesen großer Datenmengen aus Dateien kann man sich auf diese Weise also schnell einen Überblick über das vorliegende Datenvolumen verschaffen.

Anstatt sich den gesamten Inhalt eines Dataframes anzeigen zu lassen, besteht zudem die Möglichkeit, durch `df.head()` bzw. `df.tail()` lediglich eine bestimmte Anzahl der ersten bzw. letzten Zeilen auszugeben, die sich in der Klammer durch den Parameter `n` vorgeben lässt. So würde etwa `df.tail(n=3)` die letzten drei Tabellenzeilen anzeigen.

Möchten Sie sich die Bezeichnungen der Zeilen und Spalten eines Dataframes anzeigen lassen, so lässt sich dies mit den Befehlen `df.index` bzw. `df.columns` realisieren. Auf einzelne Bezeichnungen daraus lässt sich hierbei mittels eckiger Klammern zugreifen. Zum Beispiel haben `df.index[0]` bzw. `df.columns[0]` in unserem Fall die Ausgaben 1 bzw. 'kapiteltitel' zur Folge.

Spalteninhalte können Sie sich wie bei Dictionaries durch die Nennung des Schlüsselwerts ausgeben lassen. Beispielsweise würde `df['kapiteltitel']` die Anzeige der entsprechenden Spalte bewirken. Eine weitere interessante Möglichkeit der Spaltenanzeige ist durch die Punktenschreibweise gegeben, die wir von der Arbeit mit Klassen und den Attributen ihrer Instanzen kennen. So können Sie etwa die Wortanzahlenspalte dieses Buchs durch den einfachen Befehl `df.woerter` einsehen.

```
[12]: df.woerter
[12]: 1    3027
      2    2150
      3    1031
      4    2333
      5    3090
      6    6202
      7    2785
      8    3203
      9    3647
     10   3338
     11   8735
     12   4985
     13   2311
     14   4377
     15   8928
     16   2681
     17   8197
Name: woerter, dtype: int64
```

**Abb. 18.6** Eine Dataframe-Spalte durch Klassennotation anzeigen

Wie Sie sehen, wird auf der linken Seite ebenfalls die Zeilenbeschriftung angezeigt. Mithilfe dieses Labels können Sie wie im Falle von Schlüsseln bei Dictionaries auf einzelne Spaltenwerte zugreifen. So gibt etwa `df.woerter[17]` den Wert 8197 aus.

Der Inhalt einzelner Zeilen lässt sich durch die Zugriffsmethode `loc` ausgeben, wobei in der eckigen Klammer auch hier das Label der gewünschten Zeile stehen muss. So gibt `df.loc[11]` etwa die Daten zu Kapitel 11 aus:

[21]:	<code>df.loc[11]</code>						
[21]:	<table> <thead> <tr> <th>kapitelstitel</th> <th>Objektorientierte Programmierung</th> </tr> </thead> <tbody> <tr> <td>woerter</td> <td>8735</td> </tr> <tr> <td>teilkapitel</td> <td>7</td> </tr> </tbody> </table>	kapitelstitel	Objektorientierte Programmierung	woerter	8735	teilkapitel	7
kapitelstitel	Objektorientierte Programmierung						
woerter	8735						
teilkapitel	7						

Name: 11, dtype: object

Abb. 18.7 Anzeigen einer Dataframe-Zeile durch die `loc`-Methode

Möchten Sie einzelne Zeilen oder Spalten aus Ihrer Tabelle entfernen, so kommen hierfür die beiden Befehle `drop` bzw. `del` in Frage. Durch Nennung des Zeilen-Labels können Sie eine bestimmte Zeile – in diesem Fall die letzte – somit etwa durch

```
1 df.drop(labels=[17])
```

bzw. eine gewünschte Spalte wie gewöhnlich bei Dictionaries durch

```
1 del df['teilkapitel']
```

aus Ihrem Dataframe löschen.

Eine sehr häufige Praxis in der Datenanalyse stellt das *Sortieren* von Daten dar. Pandas verfügt hier über die einfache Methode `sort_values`, mithilfe derer sich die Daten einer gewünschten Spalte anhand ihrer Größe in auf- oder absteigender Reihenfolge sortieren lassen. So liefert etwa der Befehl

```
1 df.sort_values(by=['woerter'], ascending=False)
```

als Output eine nach Wörteranzahlen absteigend sortierte Tabelle:

## 18 Datenanalyse mit Python

```
[24]: df.sort_values(by=['woerter'], ascending=False)
```

	kapitelstitel	woerter	teilkapitel
15	Desktop-GUIs	8928	7
11	Objektorientierte Programmierung	8735	7
17	Django	8197	7
6	Datenstrukturen	6202	8
12	Fehler und Ausnahmen	4985	6
14	Datenbanken	4377	5
9	Funktionen	3647	6
10	Module aus der Standardbibliothek	3338	3
8	Schleifen	3203	6
5	Variablen	3090	6
1	Einleitung	3027	5
7	Entscheidungen treffen	2785	5
16	E-Mails	2681	4
4	Programme in eine Datei schreiben	2333	4
13	Textdateien zur Datenspeicherung	2311	4
2	Vorbereitung	2150	4
3	Der interaktive Modus	1031	2

Abb. 18.8 Sortieren einer gewünschten Spalte

Eine weitere wichtige Funktionalität der Datenbearbeitung ist das *Filtern* von Daten. Die gewünschten Ergebnisse lassen sich dabei mithilfe unterschiedlicher logischer Operatoren ermitteln. Um herauszufinden, welche Kapitel mehr als fünf Teilkapitel beinhalten, lässt sich beispielsweise ganz einfach der Operator `>` verwenden:

```
1 filter = df['teilkapitel'] > 5
2 filter
```

Im Output wird nun all denjenigen Kapiteln der Wahrheitswert `True` zugeordnet, die mehr als fünf Teilkapitel enthalten; dies sind die Kapitel 5, 6, 8, 9, 11, 12, 15 und 17. Durch `df[filter]` lassen sich die Daten dieser Kapitel nun abrufen:

```
[26]: df[filter]
```

	kapitelstitel	woerter	teilkapitel
5	Variablen	3090	6
6	Datenstrukturen	6202	8
8	Schleifen	3203	6
9	Funktionen	3647	6
11	Objektorientierte Programmierung	8735	7
12	Fehler und Ausnahmen	4985	6
15	Desktop-GUIs	8928	7
17	Django	8197	7

Abb. 18.9 Filtern einer Spalte nach einem gewünschten Kriterium

Einfache statistische Auswertungen lassen sich mit `describe()` vornehmen, wodurch unter anderem der Durchschnitt jeder Zahlenspalte gebildet wird. Mit `df.describe()` können wir so beispielsweise herausfinden, dass die durchschnittliche Wörteranzahl eines Kapitels dieses Buchs 4177,65 Wörter beträgt, während ein Kapitel durchschnittlich aus 5,24 Teilkapiteln besteht.

Die hier vorgestellten Beispiele sollten Ihnen einen einführenden Überblick über die zahlreichen Möglichkeiten der pandas-Bibliothek verschaffen. Die komplette pandas-Dokumentation finden Sie unter <https://pandas.pydata.org/>.

## 18.3 NumPy

Bei NumPy (Abk. für *Numerical Python*) handelt es sich um eine sehr häufig verwendete Bibliothek für numerische Berechnungen in Python, die auch gerne für maschinelles Lernen eingesetzt wird. NumPy beruht insbesondere auf der mehrdimensionalen Array-Datenstruktur `ndarray`, mithilfe derer sich Vektor- und Matrizenoperationen effizienter und schneller als mit gewöhnlichen Python-Listen ausführen lassen. NumPy hält daher sehr viele Funktionalitäten aus der linearen Algebra bereit.

### 18.3.1 Ein NumPy-Array erstellen

Um NumPy verwenden zu können, geben Sie zunächst den Import-Befehl mit dem Alias `np` ein:

```
1 import numpy as np
```

Ein einfaches Array lässt sich nun beispielsweise auf die folgende Art und Weise erstellen:

```
1 arr = np.array([1, 2, 3, 4, 5])
```

Anstelle einer Liste können Sie der `array`-Methode auch andere `array`-ähnliche Objekte wie zum Beispiel Tupel übergeben. Das obige Array ist eindimensional: Sie können sich vorstellen, dass die Zahlen dabei Werte auf einer Achse darstellen. Zur Beschreibung von Matrizen werden hingegen häufig zweidimensionale Arrays verwendet. So lässt sich ein Array der Dimension 2 etwa durch

```
1 arr2 = np.array([[1, 2, 3], [4, 5, 6]])
```

erzeugen, während ein dreidimensionales Array die Form

```
1 arr3 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
```

## 18 Datenanalyse mit Python

mit wiederum zwei darin eingebetteten 2D-Arrays als dessen Elemente haben kann.

```
[1]: import numpy as np

[2]: arr = np.array([1, 2, 3, 4, 5])
arr

[2]: array([1, 2, 3, 4, 5])

[3]: arr2 = np.array([[1, 2, 3], [4, 5, 6]])
arr2

[3]: array([[1, 2, 3],
           [4, 5, 6]])

[4]: arr3 = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
arr3

[4]: array([[[1, 2],
           [3, 4]],
           [[5, 6],
           [7, 8]]])
```

**Abb. 18.10** Arrays unterschiedlicher Dimensionen erstellen

Auf diese Weise lassen sich Arrays beliebiger Dimensionen und Komplexität erstellen. Die Dimension eines Arrays kann man dabei ganz einfach durch das `ndim`-Attribut herausfinden. So liefert `arr3.ndim` etwa den Output 3.

Für Arrays existieren die synonymen Bezeichnungen *Vektor* oder *Matrix*, wobei ein Vektor einfach ausgedrückt ein eindimensionales Array (`ndim = 1`) bezeichnet. Bei zweidimensionalen Matrizen gibt man oftmals deren Zeilen- und Spaltenanzahlen mit an; beispielsweise ist unser zweidimensionales `arr2` eine  $2 \times 3$ -Matrix.

Wie bei pandas-Dataframes zählt das `shape`-Attribut auch im Falle von NumPy-Arrays die Anzahl der in jeder Dimension enthaltenen Elemente. So liefert `arr.shape` beispielsweise den Output `(5,)`, da das Array eindimensional ist und aus fünf Elementen besteht, während unser zweidimensionales `arr2` mit drei Elementen pro Dimension durch `arr2.shape` den Wert `(2, 3)` ausgibt.

### 18.3.2 Arrays bearbeiten

Mithilfe der `reshape`-Funktion können Sie die Dimension eines Arrays beliebig ändern. Beispielsweise ließe sich ein durch `vektor = np.array([1, 2, 3, 4])` definierter eindimensionaler Vektor mit `vektor.reshape(2, 2)` in eine  $2 \times 2$ -Matrix umwandeln. Die einzige Voraussetzung für die Umwandlung ist dabei, dass das alte und das neue Array dieselbe Anzahl an Elementen haben.

Wie bei pandas-Dataframes lässt sich auch bei NumPy-Arrays mittels eckiger Klammern auf einzelne Elemente daraus zugreifen. Je nach Dimension des Arrays müssen dabei dementsprechend viele Indizes in den eckigen Klammern stehen, um ein spezifisches Element als Output zu erhalten. So liefern etwa `arr[0]` das erste und `arr2[1, 2]` das letzte Element unserer zuvor definierten ein- bzw. zweidimensionalen Arrays. Die erste Index-Zahl steht dabei für den Index der ersten Dimensionstiefe, die zweite für den Index der zweiten Dimensionstiefe, usw. Geben Sie für ein zweidimensionales Array jedoch nur einen Index an – zum Beispiel `arr2[1]` – so erhalten Sie als Output das komplette untergeordnete Array, in diesem Fall also `array([4, 5, 6])`.

*Slicing* bezeichnet die Möglichkeit, bestimmte Elemente innerhalb angegebener Indizes auszuwählen. Dies wird mithilfe einer Doppelpunktnotation bewerkstelligt, die Ihnen womöglich noch von der Arbeit mit Listen (Kapitel 6.2.) bekannt ist. Slicing funktioniert nach demselben Schema wie bei gewöhnlichen Python-Listen und lässt sich ebenfalls auf pandas-Dataframes anwenden. Sie können einmal selbst versuchen, in einem Jupyter-Notebook den Output der folgenden Zeilen, die verschiedene Slicing-Methoden beschreiben, nachzuvollziehen:

```

1 arr[1:4]
2 arr[:2]
3 arr[1:4:2] # die letzte Zahl gibt die Schrittweite an
4 arr2[0, 1:3]
5 arr2[0:2, 0]
6 arr2[0:2, 1:3]
```

Sie können über ein Array mittels `for`-Schleife iterieren, um sich einzelne Elemente daraus anzeigen zu lassen oder diese durch weitere Operationen bearbeiten zu können. Hierfür kommt zusätzlich das Iterationsobjekt `nditer` zum Einsatz. Im folgenden Beispiel erstellen wir zunächst ein Array, `arr4`, mithilfe der `arange`-Methode, die ähnlich zur `range`-Funktion in Python funktioniert, jedoch als optionalen vierten Parameter ebenfalls die Vorgabe des gewünschten Datentyps aufnehmen kann. Mit `nditer` gibt unser Code anschließend separat die vier Elemente in `arr4` aus:

```

1 arr4 = np.arange(2, 10, 2)
2 for x in np.nditer(arr4):
3     print(x)
```

```
[23]: arr4 = np.arange(2, 10, 2)
for x in np.nditer(arr4):
    print(x)

2
4
6
8
```

Abb. 18.11 Arrays unterschiedlicher Dimensionen erstellen

## 18 Datenanalyse mit Python

Möchten Sie jedoch eine Operation gleichermaßen auf alle Elemente eines Arrays ausüben, so gibt es hierfür anstelle der `for`-Schleife eine einfachere Methode, dies zu tun: das sogenannte *Broadcasting*. Hierbei wird ein und dieselbe arithmetische Operation auf alle Elemente eines Arrays angewandt. Zum Beispiel können wir zu jedem Element in `arr` ganz einfach durch `arr + 1` jeweils den Wert 1 addieren, was intern die Berechnung `arr + [1, 1, 1, 1, 1]` bewirkt und die Ausgabe `array([2, 3, 4, 5, 6])` zur Folge hat. Ebenso lassen sich Werte subtrahieren, multiplizieren oder teilen. Testen Sie hierzu beispielsweise einmal, welchen Output die folgenden Operationen bewirken:

```
1 arr*-1
2 arr/2
3 arr*4
4 arr**2
```

Broadcasting umfasst ebenfalls arithmetische Operationen wie Addition, Subtraktion, Multiplikation, Division etc. zwischen mehreren Arrays. Beispielsweise können Sie so ganz einfach `arr + np.arange(2, 12, 2)` addieren. Hierfür müssen die `shape`-Werte beider Arrays jedoch stets kompatibel sein. Anstelle der genannten arithmetischen Operatoren können Sie ebenfalls die Methoden `add`, `multiply`, `subtract` oder `divide` verwenden.

Um am Ende eines Arrays weitere Werte manuell hinzuzufügen, kommt die `append`-Methode nach dem folgenden Schema zum Einsatz:

```
1 np.append(arr, 6)
2 np.append(arr, [6,7])
3 np.append(arr[0], [0,-1])
```

Möchten Sie die Änderungen an einem Array speichern, so dürfen Sie nicht vergessen, die Eingabe einer neuen Bezeichnung für das abgeänderte Array gleichzusetzen.

Damit sich zwei Matrizen multiplizieren lassen, muss die Anzahl der Spalten der ersten Matrix gleich der Anzahl der Zeilen der zweiten Matrix sein. Die Multiplikation erfolgt mithilfe von `dot`:

```
1 arr4 = arr4.reshape(2,2)
2 arr5 = np.dot(arr4, arr2)
```

```
[34]: arr4 = arr4.reshape(2,2)
arr4
[34]: array([[2, 4],
       [6, 8]])

[35]: arr2
[35]: array([[1, 2, 3],
       [4, 5, 6]])

[36]: arr5 = np.dot(arr4, arr2)
arr5
[36]: array([[18, 24, 30],
       [38, 52, 66]])
```

**Abb. 18.12** Zwei Matrizen multiplizieren

Sie können Arrays in einzelne Teile zerlegen sowie umgekehrt verschiedene Arrays zu einem einzigen Array zusammenfügen. Dies geschieht mithilfe der Methoden `split` und `concatenate`:

```
1 np.split(arr5, 2) # teilt arr5 in zwei Arrays
2 np.split(arr5, 3, axis=1) # teilt arr5 vertikal in drei Arrays
3 np.concatenate((arr2, arr5)) # fügt arr2 und arr5 entlang der Zeilen zusammen
4 np.concatenate((arr2, arr5), axis=1) # fügt die Arrays entlang der Spalten
5 zusammen
```

Standardmäßig ist hierbei `axis = 0` angegeben, wodurch die Arrays entlang der Zeilen, also der horizontalen Achse, zerlegt bzw. zusammengefügt werden.

Als Nächstes wollen wir betrachten, wie sich Arrayelemente hinsichtlich eines bestimmten Inhalts durchsuchen lassen. Hierfür verwendet man die `where`-Methode, die die Indizes derjenigen Array-Elemente ausgibt, für die das Kriterium zutrifft:

```
1 x = np.where(arr > 2)
2 arr[x]
[41]: x = np.where(arr > 2)
x
[41]: (array([2, 3, 4], dtype=int64),)

[42]: arr[x]
[42]: array([3, 4, 5])
```

**Abb. 18.13** Arrays nach bestimmten Kriterien durchsuchen

Vorausgesetzt, Ihr Array liegt bereits in sortierter Form vor, sodass die Elemente darin in aufsteigender Reihenfolge angeordnet sind, so können Sie mit `searchsorted` überprüfen, an welcher Stelle neue Werte eingesetzt werden können, damit die Sortierung erhalten bleibt. Somit würde `y = np.searchsorted(arr, 2.5)` etwa den Output 2 erzeugen. Dies bedeutet, dass die Zahl 2,5 beim Einfügen in `arr` den Index 2

## 18 Datenanalyse mit Python

erhalten sollte. Diese Methode funktioniert ebenfalls im Falle mehrerer Elemente, die dann jedoch innerhalb einer Liste angegeben werden müssen. Testen Sie dies einmal selbst!

Mit `sort` können Sie die Elemente eines unsortierten Arrays in eine sortierte Reihenfolge bringen. Für ein durch `arr6 = np.array([1, 0, 3, 1, 2])` definiertes Array würde dies beispielsweise durch `np.sort(arr6)` realisiert. Diese Sortierung funktioniert übrigens ebenfalls für mehrdimensionale Arrays, boolesche Werte oder Strings, wobei letztere alphabetisch sortiert werden. Bei der Sortierung lässt sich zusätzlich die Achse, die Sortierungsmethode sowie das Feld, nach welchem sortiert werden soll, spezifizieren.

Beim *Filtern* werden spezifische Elemente einem Array entnommen, die anschließend in ein neues Array überführt werden können. So könnten wir in unserem `arr6` etwa ausschließlich Zahlen über 1 stehen haben wollen, wofür wir eine Zuordnung der jeweiligen erwünschten bzw. auszuschließenden Elemente zu `True` bzw. `False` vornehmen:

```
1 x = [False, False, True, False, True]
2 arr6neu = arr6[x]
```

Der obige Code veranschaulicht eine manuelle Zuschreibung der Wahrheitswerte. In der Praxis ist es jedoch üblicher, die Filterung nach bestimmten Kriterien automatisch auszuführen. Im behandelten Fall wird dies etwa nach dem folgenden Schema bewerkstelligt:

```
1 filter_array = []
2
3 for element in arr6:
4     if element > 1:
5         filter_array.append(True)
6     else:
7         filter_array.append(False)
8
9 arr6neu = arr6[filter_array]
```

Wichtig ist hierbei, zunächst eine leere Liste zu erstellen, der die Wahrheitswerte für jedes Element in einer `for`-Schleife mittels `if-else`-Klauseln und `append`-Methode hinzugefügt werden.

## 18.4 Datenvisualisierung mit Seaborn

```
[46]: x = [False, False, True, False, True]
arr6neu = arr6[x]
arr6neu

[46]: array([3, 2])

[47]: filter_array = []
for element in arr6:
    if element > 1:
        filter_array.append(True)
    else:
        filter_array.append(False)

arr6neu = arr6[filter_array]
arr6neu

[47]: array([3, 2])
```

**Abb. 18.14** Gewünschte Werte aus einem Array herausfiltern

Selbstverständlich hält NumPy ebenfalls zahlreiche statistische Methoden parat. Mit `amin` und `amax` lassen sich beispielsweise jeweils das kleinste bzw. größte Element eines Arrays ausgeben. Weiterhin können Sie den Durchschnitt aller Elemente mit `mean()` ermitteln. Optional kann man dabei jeweils die Achse auswählen, entlang derer nach den entsprechenden Elementen gesucht werden soll. Auch andere statistische Größen wie der Median, die Standardabweichung und die Varianz lassen sich nach diesem Schema ganz einfach berechnen.

Für die Arbeit mit Matrizen existieren in NumPy unter anderem Methoden zur Berechnung des Skalarprodukts, der Determinanten oder der Inversen einer Matrix sowie zum Lösen linearer Gleichungen.

Zusätzlich zu den hier vorgestellten gibt es noch viele weitere und fortgeschrittenere Optionen, wie sich Arrays bearbeiten lassen um Informationen daraus zu gewinnen. Diese finden Sie unter anderem in der NumPy-Dokumentation unter [www.numpy.org](http://www.numpy.org) beschrieben.

## 18.4 Datenvisualisierung mit Seaborn

Oftmals verwendet man im Python-Code eine Kombination mehrerer Datenanalyse-Bibliotheken, um von deren unterschiedlichen Funktionalitäten Gebrauch zu machen. Dabei lassen sich ebenfalls verschiedene Datenstrukturen ineinander überführen. Beispielsweise kann man ein pandas-Dataframe durch die Funktion `to_numpy()` in ein NumPy-Array umwandeln. Umgekehrt lässt sich etwa ein pandas-Dataframe durch die Angabe eines NumPy-Arrays im `data`-Argument erstellen. In den Übungen werden Sie die Chance haben, dies auszuprobieren.

In den folgenden Abschnitten wollen wir ebenfalls zwei Bibliotheken – pandas und Seaborn – gleichzeitig verwenden, um damit bestimmte Daten und Ergebnisse zu vi-

## 18 Datenanalyse mit Python

sualisieren und diese beispielsweise anhand von Punkt- und Liniendiagrammen oder Histogrammen zu veranschaulichen. Durch die optische Darstellung werden die Informationen aus den Daten oftmals leichter verständlich, als wenn man sie lediglich in Zahlen- oder Tabellenform betrachten würde.

Zur Visualisierung dienen in der Regel die Bibliotheken Matplotlib und Seaborn. Während Matplotlib die ältere, grundlegendere Bibliothek der beiden ist, kommt die intuitivere Seaborn-Bibliothek mit einer reduzierten Syntax aus, ist vielseitiger und liefert ästhetischere Plots. Seaborn baut auf Matplotlib auf und ist als Ergänzung, nicht als Ersatz dieser Bibliothek gedacht. Insbesondere für fortgeschrittenere Funktionalitäten werden Sie daher meist beide Bibliotheken benötigen.

Die Seaborn-Bibliothek ist eng mit pandas-Dataframes verbunden, sodass sich deren Plottingfunktionen ganz einfach auf Dataframes anwenden lassen. Genauso funktioniert Seaborn gut in Kombination mit NumPy-Arrays.

### 18.4.1 Daten für Visualisierungen einlesen

Wenn Sie mit Seaborn arbeiten möchten, gibt es zwei Möglichkeiten, wie sich Daten zu Visualisierungszwecken verwenden lassen: Sie können entweder von einem in der Seaborn-Bibliothek bereits enthaltenen Datenset Gebrauch machen oder ein selbst erstelltes pandas-Dataframe bzw. NumPy-Array einlesen. Während sich erstgenannte Möglichkeit vor allem für Lernzwecke eignet, wird man für fortgeschrittenere Anwendungen meist jedoch eigene Datenbestände verwenden und sich diese anzeigen lassen wollen. In dieser Einführung werden wir Ihnen beide Möglichkeiten vorstellen.

Die hier beschriebenen Codebeispiele wurden mit Seaborn 0.11 erstellt. Sollten einige der Funktionen oder Plots in Ihrem Notebook bzw. Ihrer IDE nicht funktionieren, müssen Sie nach dem Seaborn-Import mit `sns.__version__` überprüfen, welche Version auf Ihrem Rechner installiert ist und, wenn nötig, entsprechende Versions- oder Funktionsanpassungen vornehmen.

Seaborn importiert man in der Regel unter dem Kürzel `sns`. Zum Abrufen eines der in Seaborn eingebauten Datensets, die Sie allesamt unter <https://github.com/mwaskom/seaborn-data> aufgelistet finden, dient die Funktion `load_dataset()`. So können Sie sich beispielsweise folgendermaßen die ersten fünf Einträge des *Iris*-Datensets, das unterschiedliche Eigenschaften der Iris-Pflanze beschreibt, anzeigen lassen:

```

1 import seaborn as sns
2
3 iris = sns.load_dataset("iris")
4 iris.head()

```

## 18.4 Datenvisualisierung mit Seaborn

```
[1]: import seaborn as sns
[2]: iris = sns.load_dataset("iris")
      iris.head()
[3]:   sepal_length  sepal_width  petal_length  petal_width  species
[4]: 0       5.1        3.5         1.4        0.2    setosa
[5]: 1       4.9        3.0         1.4        0.2    setosa
[6]: 2       4.7        3.2         1.3        0.2    setosa
[7]: 3       4.6        3.1         1.5        0.2    setosa
[8]: 4       5.0        3.6         1.4        0.2    setosa
```

**Abb. 18.15** Die Anzeige der ersten Einträge eines Datensets

Wie Sie sehen, enthält das Iris-Datenset die Daten vierer Messgrößen (Länge und Breite von Kelch- und Blütenblatt der Schwertlilie), die in 150 Stichproben für drei verschiedene Iris-Arten (*Setosa*, *Virginica* und *Versicolor*) erhoben wurden.

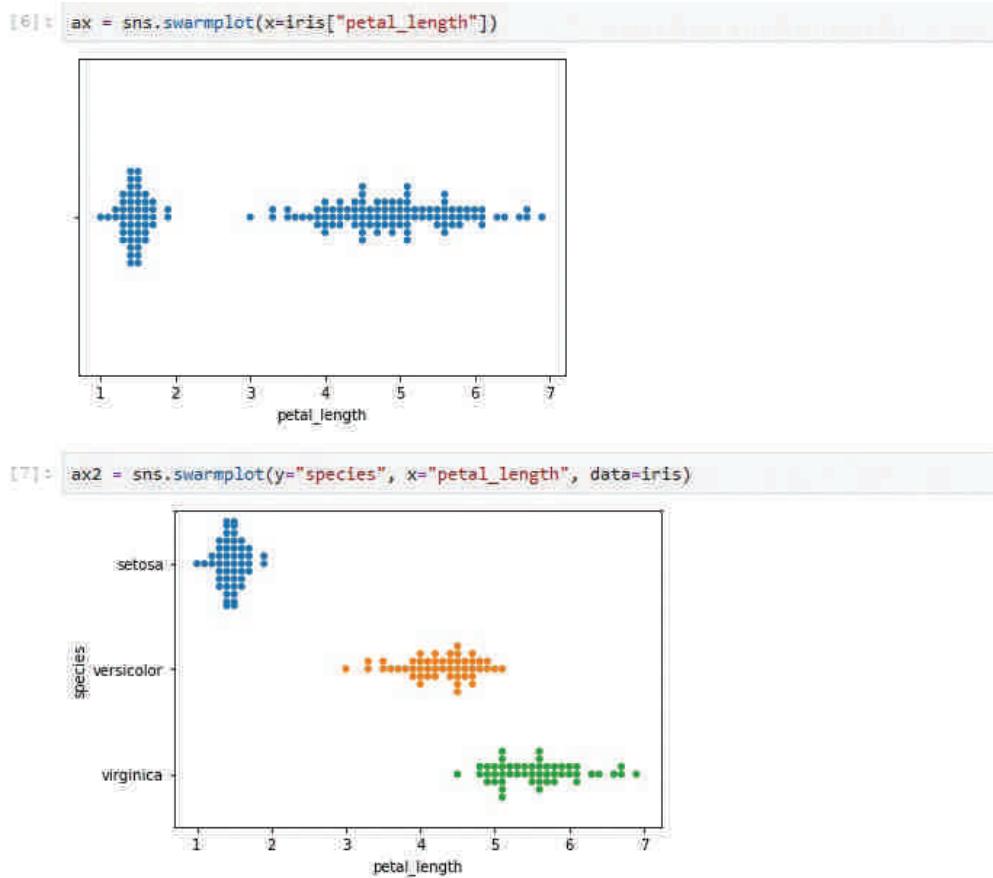
Eine einfache Visualisierung aller 150 Messpunkte für die Blütenblatlänge lässt sich daraus beispielsweise als Swarmplot erstellen, wobei die einzelnen Datenpunkte in ein Koordinatensystem eingetragen werden:

```
1 ax = sns.swarmplot(x=iris["petal_length"])
```

Der Inhalt der Klammern gibt dabei an, dass in der x-Achse die Messdaten von `petal_length` stehen sollen. Folgendermaßen können Sie eine Differenzierung der Messdaten in die drei unterschiedlichen Arten mit entsprechender farblicher Kennzeichnung vornehmen:

```
1 ax2 = sns.swarmplot(y="species", x="petal_length", data=iris)
```

## 18 Datenanalyse mit Python



**Abb. 18.16** Zwei unterschiedliche Visualisierungen der Daten als Swarmplot

Sie können selbst ausprobieren, im Code die Achsenzuordnungen umzukehren oder die gewünschte Messdatenvariable zu ändern.

Bevor wir Ihnen im nächsten Abschnitt einige weitere Plottingoptionen vorstellen, werden wir noch untersuchen, wie Sie eigene Datensets mithilfe von Dataframes in Seaborn einlesen können.

Vergessen Sie dabei zunächst nicht, ebenfalls die pandas-Bibliothek zu importieren, wenn Sie in Seaborn mit einem pandas-Dataframe arbeiten möchten. Angenommen, Sie haben nun bereits ein pandas-Dataframe definiert – entweder direkt im Code oder durch einen Datenimport, etwa mit `read_csv` oder `read_excel`. Wir werden hierfür das bereits in 18.2.1. definierte Buchdaten-Dataframe `df` verwenden. Ein Swarmplot mit der Kapitelnummerierung auf der x-Achse sowie den Wortanzahlen auf der y-Achse lässt sich daraus folgendermaßen durch eine einzige Zeile erstellen, wobei Sie die Dataframe-Labels sogleich als Achsenbeschriftung wählen können:

## 18.4 Datenvisualisierung mit Seaborn

```
1 ax3 = sns.swarmplot(x=df.index, y="woerter", data=df)
```

```
[1]: import pandas as pd
[1]: buchdaten = {'kapitteltitel': ['Einleitung', 'Vorbereitung', 'Der interaktive Modus',
[1]:     'Programme in eine Datei schreiben', 'Variablen', 'Datenstrukturen',
[1]:     'Entscheidungen treffen', 'Schleifen', 'Funktionen',
[1]:     'Module aus der Standardbibliothek', 'Objektorientierte Programmierung',
[1]:     'Fehler und Ausnahmen', 'Textdateien zur Datenspeicherung', 'Datenbanken',
[1]:     'Desktop-GUIs', 'E-Mails', 'Django'],
[1]:     'woerter' : [3027, 2150, 1031, 2333, 3090, 6202, 2785, 3203, 3647, 3338, 8735,
[1]:     4985, 2311, 4377, 8928, 2081, 8197],
[1]:     'teilkapitel' : [5, 4, 2, 4, 6, 0, 5, 6, 6, 3, 7, 6, 4, 5, 7, 4, 7]
[1]: }
[1]: kapitel = range(1, 18)
[1]: df = pd.DataFrame(data=buchdaten, index=kapitel)
[1]: ax3 = sns.swarmplot(x=df.index, y="woerter", data=df)
```

**Abb. 18.17** Ein Swarmplot der woerter-Daten unseres Buchdaten-Dataframes pro Kapitel

Wie im Abschnitt 18.2.3. *Ein Dataframe bearbeiten* beschrieben, lässt sich mittels Punktschreibweise auf einzelne Spalten des Dataframes – hier: `df.index` für die Kapitelnummern – zugreifen.

### 18.4.2 Verschiedene Plottingfunktionen

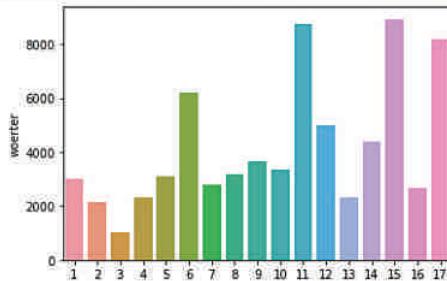
Es gibt mehrere Möglichkeiten, Datenmengen in Seaborn abzubilden. Beispielsweise können Sie eine andere graphische Darstellung erzeugen, wenn Sie anstelle von `swarmplot` die Methoden `barplot`, `violinplot` oder `displot` verwenden.

Ein *Barplot* ähnelt im Aufbau dem bereits vorgestellten Swarmplot. Anstelle der Datenpunkte stehen hier jedoch Balken:

```
1 ax4 = sns.barplot(x=df.index, y="woerter", data=df)
```

## 18 Datenanalyse mit Python

```
[14]: ax4 = sns.barplot(x=df.index, y="woerter", data=df)
```

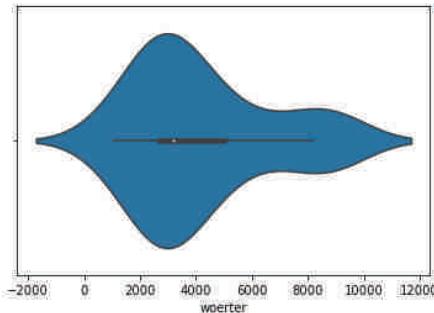


**Abb. 18.18** Ein Barplot der woerter-Daten unseres Buchdaten-Dataframes pro Kapitel

Ein *Violinplot* verschafft insbesondere ein gutes Verständnis der Verteilung von Daten, wie Sie anhand des untenstehenden Plots erkennen können. Daraus wird unter anderem ersichtlich, dass die meisten Kapitel dieses Buchs Wortanzahlen zwischen 2000 bis 4000 Wörtern aufweisen. Zusätzlich zur Verteilung wird in der Mitte des Violinplots ebenfalls der Median – derjenige Messwert, der in der Mitte aller Werte liegt – als weißer Punkt, die Quartile – die beiden Punkte mit einem Viertel aller Werte ober- bzw. unterhalb – sowie die Whisker – die den Minimum- bzw. Maximumwert auswiesen – als Boxplot angezeigt.

```
1 ax5 = sns.violinplot(x="woerter", data=df)
```

```
[15]: ax5 = sns.violinplot(x="woerter", data=df)
```



**Abb. 18.19** Ein Violinplot der woerter-Daten unseres Buchdaten-Dataframes

Reine Boxplots lassen sich übrigens mit der `boxplot`-Methode erstellen. Eine andere optische Darstellung können Sie erzeugen, indem Sie der Plottingfunktion die Codezeile `sns.set_theme(style = "darkgrid")` voranstellen. Durch die Wahl unterschiedlicher Farbschemata lassen sich statistische Zusammenhänge mitunter leichter erkennen. Durch die Angabe der gewünschten Farbpalette können Sie zudem die Farbe des Plots auswählen, zum Beispiel mit `palette="rocket"` als weiterem Argument in der `set_theme`-Methode. Die unterschiedlichen Optionen in Bezug auf

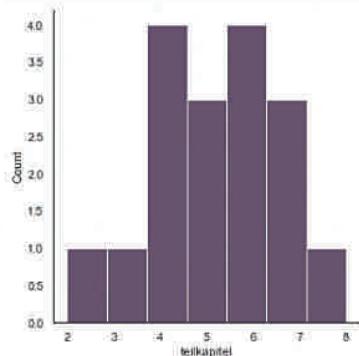
## 18.4 Datenvisualisierung mit Seaborn

Ästhetik und Farbschemata finden Sie allesamt auf den Tutorialseiten der Seaborn-Dokumentation (<https://seaborn.pydata.org/tutorial.html>) unter *Plot aesthetics* aufgelistet.

Zum Erstellen von Histogrammen dienen die Plottingfunktionen `histplot` oder `displot`. Damit können Sie leicht veranschaulichen, wie häufig bestimmte Werte innerhalb eines Datensets vorkommen:

```
1 ax6 = sns.displot(df.teilkapitel, bins=7)
```

```
[22]: sns.set_theme(style="white", palette="rocket")
ax6 = sns.displot(df.teilkapitel, bins=7)
```



**Abb. 18.20** Das Histogramm zeigt die Häufigkeit der `teilkapitel`-Daten unseres Buchdaten-Dataframes an

Damit die möglichen Datenwerte in der richtigen Einteilung dargestellt werden, müssen Sie deren Anzahl mitunter durch die zusätzliche Angabe der `bins`-Klassen festlegen, wie hier mit `bins=7`. Aus dem obigen Histogramm wird ersichtlich, dass die häufigsten Teilkapitel-Anzahlen 4 und 6 sind und jeweils viermal vorkommen.

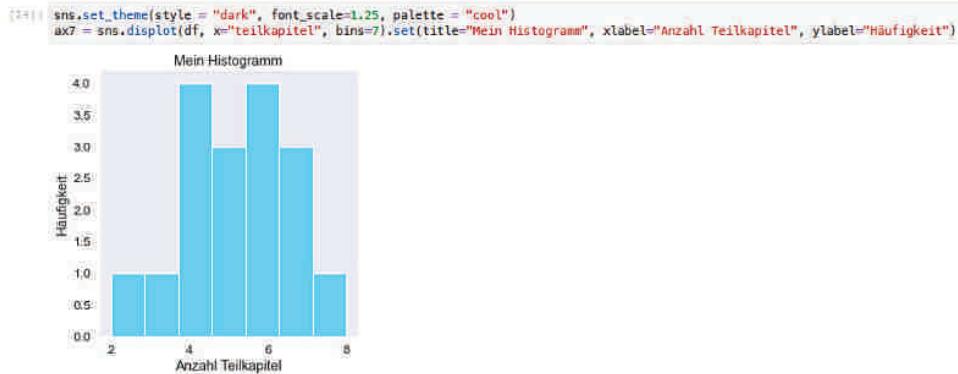
Im folgenden Code haben wir einige Details an unserem vorigen `displot` geändert:

```
1 sns.set_theme(style = "dark", font_scale=1.25, palette = "cool")
2 ax7 = sns.displot(df, x="teilkapitel", bins=7).set(title="Mein Histogramm",
3 xlabel="Anzahl Teilkapitel", ylabel="Häufigkeit")
```

In der ersten Zeile wurde hier ein anderes Design samt Schriftgröße und Farbpalette ausgewählt. Beachten Sie zudem in der zweiten Zeile eine alternative Möglichkeit, die x-Achse anzugeben und vergleichen Sie diese Methode mit der Angabe in `ax6`. In der zweiten Codezeile wird außerdem durch die `set`-Methode ein Titel für das Histo-

## 18 Datenanalyse mit Python

gramm angegeben sowie eine Beschriftung der x- und y-Label vorgenommen. Das Ergebnis dieser Änderungen sieht folgendermaßen aus:



**Abb. 18.21** Histogramm mit geänderter graphischer Darstellung

Eine weitere interessante Plottingfunktion, mithilfe derer Sie statistische Analysen mittels linearer Regression durchführen können, ist `regplot`. Darüber hinaus bietet Seaborn viele weitere Plottingfunktionen, die Sie in der Beispieldokumentation der Seaborn-Dokumentation unter <http://seaborn.pydata.org/examples/> einsehen können.

Aus den Techniken und Möglichkeiten der Datenanalyse haben sich ebenfalls die derzeit sehr beliebten Begriffe *Deep Learning* und *maschinelles Lernen* sowie deren Oberbegriff *künstliche Intelligenz (KI)* entwickelt, die dem menschlichen Denken ähnelnde, fortgeschrittenere Algorithmen bezeichnen. Allgemein gesprochen definiert ein Algorithmus dabei eine Abfolge von Rechenschritten zur Lösung eines mathematischen Problems. Sobald Sie sich ein wenig in der Welt der Datenanalyse zurechtgefunden haben, sollten Sie nicht verpassen, mehr über diese spannenden fortgeschrittenen Themen zu lernen.

## 18.5 Übung: Datenanalyse mit Python

Da der Inhalt der folgenden Übungsaufgaben teilweise über den hier behandelten Stoff hinausgeht, ist zur Bearbeitung der Aufgaben mitunter eine eigenständige Internetrecherche nötig.

### Übungsaufgaben

1. Definieren Sie ein eindimensionales NumPy-Array, das zehn Hauptstädte dieser Welt enthält. Wenden Sie die Attribute `ndim` und `shape` darauf an. Sortieren Sie das Array anschließend alphabetisch. Filtern Sie nun diejenigen Einträge heraus, die mehr als sieben Buchstaben enthalten und speichern Sie diese in einem neuen Array ab. Wenden Sie schließlich die Methoden `amin` und `amax` auf das neue Array an und erklären Sie den Output.
2. Erstellen Sie mithilfe des NumPy-Moduls `random` eine aus Zufallszahlen bestehende  $6 \times 6$ -Matrix. Definieren Sie daraus anschließend ein pandas-Dataframe, dessen Zeilenindex aus Strings besteht. Wandeln Sie das Dataframe danach mithilfe der Funktion `to_numpy()` wieder in ein NumPy-Array um.
3. Plotten Sie das in Übung 2 erstellte Array als Heatmap in Seaborn.

## 18 Datenanalyse mit Python

### Lösungen

1. Wir definieren zunächst unser Array. Die einzelnen Elemente darin können wir uns separat mithilfe von `nditer` ausgeben lassen. Um die Methoden `amin` und `amax` auf unser gefiltertes Array anwenden zu können, müssen wir dessen Datentyp in `object` umwandeln. Die beiden Methoden geben sodann den ersten bzw. letzten Eintrag in alphabetischer Reihenfolge aus:

```
1 import numpy as np
2
3 arr = np.array(["Riad", "Mexico City", "Oslo", "Berlin", "Addis Abeba",
4 "Canberra", "Teheran", "Thimphu", "Colombo", "Ottawa"])
5
6 for x in np.nditer(arr):
7     print(x)
8
9 arr.ndim
10 arr.shape
11
12 np.sort(arr) # Sortierung vornehmen
13
14 # Gewünschte Elemente herausfiltern:
15 filter_array = []
16
17 for element in arr:
18     if len(element) > 7:
19         filter_array.append(True)
20     else:
21         filter_array.append(False)
22
23 arrneu = arr[filter_array]
24
25 arrneu = arrneu.astype(np.object) # Datentyp ändern
26
27 np.amin(arrneu)
28 np.amax(arrneu)
```

## 18.5 Übung: Datenanalyse mit Python

```

1 np.sort(arr) # Sortierung vornehmen
2 array(['Addis Abeba', 'Berlin', 'Canberra', 'Colombo', 'Mexico City',
3       'Oslo', 'Ottawa', 'Riad', 'Teheran', 'Thimphu'], dtype='<U11')
4 # Gewünschte Elemente herausfiltern:
5 filter_array = []
6
7 for element in arr:
8     if len(element) > 7:
9         filter_array.append(True)
10    else:
11        filter_array.append(False)
12
13 arrneu = arr[filter_array]
14 arrneu
15 array(['Mexico City', 'Addis Abeba', 'Canberra'], dtype='<U11')
16 arrneu = arrneu.astype(np.object) # Datentyp ändern
17 arrneu
18 array(['Mexico City', 'Addis Abeba', 'Canberra'], dtype=object)
19 np.amin(arrneu)
20 'Addis Abeba'
21 npamax(arrneu)
22 'Mexico City'

```

Abb. 18.22 Der Output des obigen Codes ab Zeile 12

2. Mithilfe des random-Moduls lassen sich Arrays mit Pseudozufallszahlen generieren, durch die Methode `rand` beispielsweise aus dem halboffenen Intervall [0.0, 1.0). Daraus kann man sehr einfach ein Dataframe erstellen, das sich wiederum durch die `to_numpy()`-Funktion in ein Array umwandeln lässt:

```

1 import pandas as pd
2
3 arr2 = np.random.rand(6,6)
4 df = pd.DataFrame(data=arr2, index=["a", "b", "c", "d", "e", "f"])
5 df.to_numpy()
6

```

## 18 Datenanalyse mit Python

```
[322]: import pandas as pd
arr2 = np.random.rand(6,6)
arr2

[322]:
array([[0.55581443, 0.66184608, 0.76984488, 0.64275588, 0.58959394,
       0.62820227],
       [0.82104112, 0.77944888, 0.86747294, 0.62829377, 0.88577824,
       0.89223195],
       [0.98173685, 0.39757863, 0.93186528, 0.0625327 , 0.86431251,
       0.92457258],
       [0.36376645, 0.0883708 , 0.34220428, 0.96805044, 0.40542451,
       0.72351951],
       [0.43639488, 0.09601509, 0.49269658, 0.49187625, 0.57472797,
       0.56430057],
       [0.66936313, 0.81236131, 0.97743557, 0.16118831, 0.75171713,
       0.08231825]])
```

```
[323]: df = pd.DataFrame(data=arr2, index=("a", "b", "c", "d", "e", "f"))
df
```

	0	1	2	3	4	5
a	0.555814	0.661846	0.769845	0.642756	0.589594	0.628202
b	0.821041	0.779449	0.867473	0.628294	0.885778	0.892232
c	0.981737	0.397579	0.931865	0.062533	0.864313	0.924573
d	0.363766	0.088371	0.342204	0.968050	0.405425	0.723520
e	0.436394	0.096015	0.492697	0.491876	0.574728	0.564301
f	0.669363	0.812361	0.977436	0.161188	0.751717	0.082318

```
[324]: df.to_numpy()
```

```
[324]:
array([[0.55581443, 0.66184608, 0.76984488, 0.64275588, 0.58959394,
       0.62820227],
       [0.82104112, 0.77944888, 0.86747294, 0.62829377, 0.88577824,
       0.89223195],
       [0.98173685, 0.39757863, 0.93186528, 0.0625327 , 0.86431251,
       0.92457258],
       [0.36376645, 0.0883708 , 0.34220428, 0.96805044, 0.40542451,
       0.72351951],
       [0.43639388, 0.09601509, 0.49269658, 0.49187625, 0.57472797,
       0.56430057],
       [0.66936313, 0.81236131, 0.97743557, 0.16118831, 0.75171713,
       0.08231825]])
```

**Abb. 18.23** Der Output des obigen Codes

3. Auf <https://seaborn.pydata.org/generated/seaborn.heatmap.html> finden Sie die Beschreibung des Heatmap-Plots. Da dieser Plot auf jegliche zweidimensionale Datensets anwendbar ist, können Sie anstelle des arr2-Arrays übrigens genauso gut das df-Dataframe aus Übung 2 verwenden. In diesem Fall werden bei der Visualisierung ebenfalls die angegebenen Zeilen- und Spaltenbeschriftungen berücksichtigt.

Beim Heatmap-Plot werden den Zahlen entsprechende Farben zugeordnet. Je näher zwei Zahlen beieinander liegen, umso ähnlicher werden deren Farben sein. Anhand der Heatmap lässt sich das Ergebnis des Algorithmus, der die Pseudozufallszahlen für unsere 6x6-Matrix generiert hat, somit sehr leicht vergleichen.

```
1 import seaborn as sns
2
3 ax = sns.heatmap(arr2)
```

## 18.5 Übung: Datenanalyse mit Python

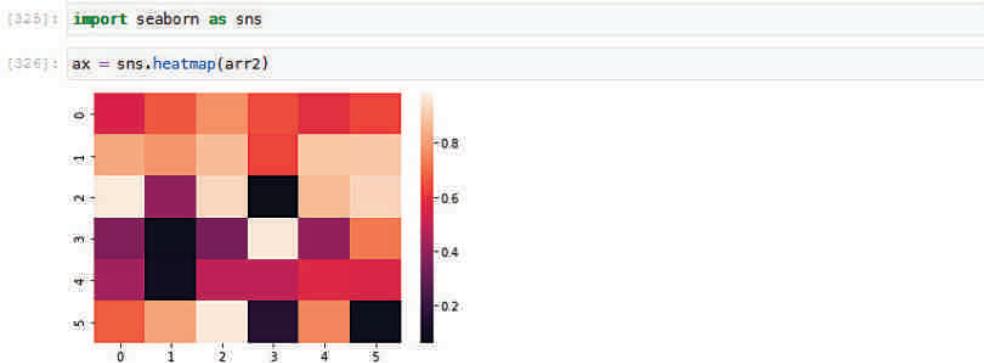


Abb. 18.24 Die Visualisierung von arr2 als Heatmap

**Herzlichen Glückwunsch!** Sie haben den ersten Einstieg in Pythons vielfältige Anwendungsbereiche erfolgreich gemeistert und sich eigenständig einen umfassenden Überblick über diese faszinierende Programmiersprache verschafft! Mit diesem fundierten Wissen können Sie nun selbst entscheiden, wohin die weitere Reise gehen soll. Wir wünschen Ihnen auch in Zukunft viel Spaß beim Lernen und Anwenden von Python.

Und sollten Sie auf Ihrem weiteren Weg doch einmal verzagen und verzweifelt Ihren Rechner verfluchen, denken Sie an Monty Pythons *Galaxy Song*, machen Sie eine Pause und tanzen Sie eine Runde zum ihrem altbewährten Hit *Always Look on the Bright Side of Life!*

## Downloadhinweis



<https://bmu-verlag.de/python-kompendium/>

**Downloadcode:** zrx4a6cfs

## Glossar

**Anaconda:** Freie Open-Source-Software-Distribution, die zahlreiche Python-Pakete und Tools für Datenanalyse und wissenschaftliches Rechnen bereithält.

**Array:** Mehrdimensionale Datenstruktur der NumPy-Bibliothek für Datenanalysezwecke, insbesondere geeignet für Vektor- und Matrizenoperationen sowie weitere Funktionalitäten aus der linearen Algebra.

**Attribut:** Spezifische Eigenschaft, die den Objekten einer Klasse zugeordnet ist und diese genauer beschreibt. Die Attribute der Objekte einer Klasse bilden die charakteristische Struktur der Objekte ab. Ein Attribut besteht jeweils aus einem festen Attributnamen sowie einem Attributwert, der sich von Objekt zu Objekt unterscheiden kann.

**Ausnahme:** Ein Objekt der Klasse `Exception`. Eine Ausnahme im Programmcode beschreibt einen Fehler, der Ursachen unterschiedlicher Art haben kann und durch eine entsprechende Ausnahmebehandlung behoben werden muss. Hierfür existieren in Python die Kontrollstrukturen `try ... except` sowie `try ... finally`.

**Bootstrap:** Freies Web-Framework für das Erstellen von Webseiten anhand von Layout-Vorlagen, in denen HTML-, CSS- und JavaScript-Elemente in einem einzigen Stylesheet zusammengeführt sind. Die Vorlagen lassen sich in Django in ein HTML-Dokument einbinden und je nach Bedarf anpassen.

**Compiler:** Computerprogramm, das den Quellcode eines Programms in Maschinencode umwandelt, damit das Programm ausgeführt werden kann. Der Kompilier-Prozess ist nicht plattformunabhängig.

**CRUD** (engl. für *Create, Read, Update & Delete*): Das Akronym beschreibt die vier wichtigsten Operationen auf Daten: Erzeugen, Lesen, Aktualisieren & Löschen.

**Dataframe:** Zweidimensionale, tabellenähnliche Datenstruktur der pandas-Bibliothek für Datenanalysezwecke, die sich entlang der Zeilen durch einen Index sowie entlang der Spalten mit Labels versetzen lässt und vielfältige Rechenoperationen ermöglicht.

**Datenbank:** Strukturierte Form der elektronischen Datenspeicherung, die der Verwaltung und Bearbeitung von Daten durch ein Datenbankmanagementsystem (DBMS) dient. Für die Abfrage der in einer Datenbank abgespeicherten Daten wird eine Datenbanksprache, beispielsweise SQL, MySQL oder PostgreSQL, eingesetzt.

**Datenstruktur:** Grundlegendes Programmkonstrukt, das der Speicherung und Organisation von Daten dient und spezifische Operationen darauf ermöglicht. Eine Datenstruktur kann gleiche oder verschiedene Datentypen enthalten. Beispiele für Datenstrukturen in Python sind Listen, Dictionaries, Tupel oder Mengen.

## Glossar

**Datentyp:** Ein fundamentaler Begriff in der Programmierung. Der Datentyp einer Variablen gibt an, in welcher Form diese abgespeichert ist, welche Wertebereiche für sie gelten und welche Operationen darauf möglich sind. Die gängigsten Python-Datentypen sind ganze Zahlen, Strings, Fließkommazahlen sowie boolesche Wahrheitswerte. Der Datentyp einer Variablen lässt sich durch die Funktion `type()` bestimmen.

**Debugger:** Werkzeug, das dem Lokalisieren und Beheben von Fehlern (sog. *Bugs*) im Programmcode dient. Heutzutage sind die meisten IDEs mit einem Debugger ausgestattet.

**Dictionary:** Eine der wichtigsten Datenstrukturen in Python, die aus Schlüssel-Wert-Paaren in geschweiften Klammern besteht. Dictionaries sind unsortiert und änderbar. Der Zugriff auf die jeweiligen Werte eines Dictionaries geschieht durch die Angabe des Schlüssels, der in der Regel durch eine Zeichenkette definiert ist.

**Django:** Ein freies, quelloffenes Web-Framework für Python, mithilfe dessen sich professionelle, datenbankgestützte Webseiten erstellen lassen. Django ermöglicht die Verwendung pythonspezifischer Elemente im Code, darunter beispielsweise Variablen, Funktionen und Kontrollstrukturen. Das Framework ist nach dem Schema *Model-View-Template (MVT)* aufgebaut.

**Docstring:** Kommentar zur Dokumentation und Beschreibung von u. a. Klassen, Modulen oder Funktionen. Docstrings werden durch drei doppelte Anführungszeichen (""""") eingeleitet und auf dieselbe Weise abgeschlossen.

**Dynamische Typisierung:** Im Gegensatz zu manch anderen Programmiersprachen müssen Variablentypen in Python anfangs nicht deklariert werden. Sie lassen sich auch im Laufe des Programms festlegen oder nachträglich ändern.

**Fließkommazahl** (auch *Gleitkommazahl*): Eine Zahl, die einen Dezimalanteil enthält. Gemäß der englischen Schreibweise ist das Dezimaltrennzeichen in Python stets ein Punkt (.) und kein Komma wie im Deutschen.

**Funktion:** In Python besteht eine Funktion aus zusammenhängenden Codezeilen, die eine bestimmte Aufgabe ausführen. Werden einer Funktion konkrete Eingabewerte (sog. *Argumente*) übergeben, so erzeugt die Funktion eine entsprechende Ausgabe. Pythons Standardbibliothek verfügt über eine Vielzahl eingebauter Funktionen, die sich im Programmcode anhand ihres Namens aufrufen lassen. Eigene Funktionen lassen sich zudem durch das Schlüsselwort `def` definieren.

**Getter:** Methode innerhalb einer Klasse, die den Wert eines Attributs liest bzw. abfragt.

**GitHub:** Auf [www.github.com](https://www.github.com) lassen sich Kopien fertiger Entwicklungsprojekte – für jegliche Programmiersprachen und Software-Anwendungen – in Form sogenannter *Repositories* hochladen und verwalten. Somit können sie etwa einer breiteren Nutzerschaft zur Verfügung gestellt, gemeinsam mit anderen bearbeitet oder in ein größeres Gesamtprojekt integriert werden.

**GUI (Graphical User Interface):** Grafische Benutzeroberflächen stellen ein Computerprogramm in einem bedienbaren Fenstersystem durch geeignete Bedienelemente grafisch dar. GUIs erlauben somit als interaktive Schnittstelle unterschiedliche Eingabe- und Aktionsmöglichkeiten zwischen den Nutzerinnen bzw. Nutzern und dem Programm.

**IDE (Integrated Development Environment):** Eine integrierte Entwicklungsumgebung dient dem Verfassen, Speichern, Testen und Ausführen von Code in einer einzigen Anwendung. Beispiele für IDEs sind u. a. PyCharm, Visual Studio Code oder IDLE.

**Instanz:** Ein konkretes Objekt einer Klasse, dem Attribute mit spezifischen Werten zugeordnet sind. Jede Instanz eines Objekts ist mit einem eigenen Namen versehen.

**Interpreter:** Programm, das den Code während seiner Ausführung einliest, Zeile für Zeile in Maschinensprache übersetzt und anschließend direkt ausführt. Dieser Vorgang ist plattformunabhängig: Interpretierte Programme können auf jedem beliebigen Rechner laufen, der mit einem Interpreter ausgestattet ist.

**Klasse:** Die wesentliche Grundlage objektorientierter Programme, die der vereinfachten Nachbildung eines Ausschnitts aus der Wirklichkeit dient. Eine Klasse bezeichnet eine übergeordnete Struktur, in der Objekte mit ähnlicher Charakteristik zusammengefasst sind. Diese objektspezifischen Eigenschaften werden durch Attribute beschrieben, die von Objekt zu Objekt unterschiedliche Ausprägungen bzw. Werte annehmen können.

**Kommentar:** Gewöhnlicher Text, der in den Quellcode eingefügt wird, jedoch keinerlei Auswirkung auf die Ausführung des Programms hat. Kommentare stehen an denjenigen Stellen im Code, die einer genaueren Erklärung bedürfen. Sie werden durch das Raute-Zeichen (#) eingeleitet.

**Liste:** Eine grundlegende Datenstruktur in Python. Die Listeneinträge stehen innerhalb eckiger Klammern und sind durch Kommas voneinander getrennt. Der Zugriff auf einzelne Elemente einer Liste erfolgt durch die Angabe ihres Index. Listen können Daten unterschiedlichen Typs aufnehmen und lassen sich auch nachträglich ändern.

**Menge:** Eine weitere wichtige Python-Datenstruktur, die dem Mengenbegriff in der Mathematik ähnelt. Die einzelnen Elemente einer Menge sind innerhalb geschweifter Klammern durch Kommas voneinander getrennt aufgelistet. Duplikate werden nicht gezählt. Die Elemente einer Menge bilden keine bestimmte Reihenfolge und können nicht mittels Indizes referenziert werden.

**Methode:** Bei Methoden handelt es sich um Funktionen, die Bestandteil einer bestimmten Klasse sind. Die Methoden einer Klasse ermöglichen Operationen auf den einzelnen Objekten. Eine spezielle Methode zur Initialisierung von Objekten innerhalb einer Klasse ist der Konstruktor. Weitere spezifische Methodentypen sind statische Methoden, Instanzmethoden und magische Methoden.

**Modul:** Pythons Standardbibliothek enthält etwa 200 Kernmodule, die für die wichtigsten Anwendungsbereiche benötigt werden. Sie lassen sich durch den

## Glossar

**import**-Befehl in das eigene Programm aufnehmen. Zusätzlich lassen sich externe Module installieren und importieren sowie eigene Module für den Gebrauch im Rahmen eines Hauptprogramms definieren.

**NumPy:** Die externe Python-Bibliothek NumPy (Abk. für *Numerical Python*) wird sehr häufig für Datenanalysezwecke und numerische Berechnungen eingesetzt. NumPy beruht insbesondere auf der mehrdimensionalen Array-Datenstruktur *ndarray*, mithilfe derer sich schnell und effizient Vektor- und Matrizenoperationen ausführen lassen. Die Bibliothek hält daher sehr viele Funktionalitäten aus der linearen Algebra bereit.

**Objekt:** Objekte bilden den Grundstein objektorientierter Programmiersprachen wie Python. Grundsätzlich hat man es in Python immer mit einem konkreten Objekt einer übergeordneten Klasse zu tun. So handelt es sich nicht nur bei explizit definierten Klassen, sondern auch bei Datentypen, bei Funktionen und Modulen sowie bei eingebauten Datenstrukturen im Wesentlichen um Klassen mit entsprechenden spezifischen Objekten. Allen Objekten einer Klasse ist gemeinsam, dass ihnen gewisse Eigenschaften (sog. Attribute) und Methoden zugeordnet sind. Dabei drücken die Attribute die klassenspezifische Charakteristik aus, während die Methoden – ganz wie Funktionen – Operationen mit den einzelnen Objekten ermöglichen.

**Objektorientierte Programmierung (OOP):** Objektorientierte Programmierung ist ein sehr umfassender Begriff, der in vielen Programmiersprachen wie

C++, C#, Java und Python von zentraler Bedeutung ist. Objektorientierung beschreibt einen fundamentalen Programmierstil – ein sogenanntes Programmierparadigma –, bei dem ein Ausschnitt der Wirklichkeit als System von Objekten betrachtet und innerhalb eines Programms in vereinfachter Form nachgebildet wird. Die Architektur des jeweiligen Programms ist jedoch nicht bloß beschreibend, sondern stets auf eine konkrete Aufgabe ausgerichtet, die das Programm handhaben soll.

**Open Source:** Der Quellcode einer Open-Source-Programmiersprache oder -Software ist nicht urheberrechtlich oder durch eine Lizenz geschützt, sondern kann von der Öffentlichkeit kostenlos eingesehen, modifiziert und weiterverbreitet werden. Open-Source-Projekte fördern infolgedessen die freiwillige Unterstützung, Verbesserung und Weiterentwicklung einer Programmiersprache oder Software durch eine aktive User-Community.

**Operator:** Mithilfe von Operatoren lassen sich in Python unterschiedliche Rechenoperationen auf Objekten wie Zahlen, Strings oder Variablen durchführen. Der Operator ist dabei derjenige Teil des Ausdrucks, der auf die Operanden ausgeübt wird. Die verschiedenen Typen umfassen u. a. den Zuweisungsoperator, arithmetische Operatoren, Vergleichsoperatoren, boolesche Operatoren und Zugehörigkeitsoperatoren.

**pandas:** Pandas ist eine praxisorientierte, externe Datenanalyse-Bibliothek für Python, die insbesondere für die Arbeit mit großen Datenmengen geeignet ist.

Zu diesem Zweck verfügt Pandas über verschiedene Datenstrukturen, darunter etwa das tabellenbasierte *Dataframe*. Eine weitere gängige pandas-Datenstruktur ist *Series*.

**PyQt:** Python-Sprachanbindung für das auf C++ basierende Qt-Framework zur Erstellung graphischer Benutzeroberflächen (GUIs). Das externe GUI-Modulpaket PyQt – derzeit in Version 5 verfügbar – erfordert eine zusätzliche Installation.

**PyPI (Python Package Index):** Sammlung verschiedener externer Pakete, die nicht Teil der Python-Standardbibliothek sind. Die Installation erfolgt normalerweise über den Kommandozeilenbefehl `pip install Paketname`. Eine Übersicht über alle für Python erhältlichen externen Pakete samt Informationen dazu finden Sie unter [www.pypi.org](http://www.pypi.org).

**Qt Designer:** Praktisches Tool zur PyQt-basierten GUI-Erstellung mit zahlreichen Layout- und Widget-Optionen.

**Schleife:** Ein wesentlicher Bestandteil vieler Programme. Durch diese zirkulären Ausführungsfolgen lassen sich spezifische Aufgaben automatisch wiederholen. Während die `while`-Schleife bis zum Erreichen eines bestimmten Ereignisses läuft, ist die Anzahl der Wiederholungen einer `for`-Schleife bereits von Anfang an festgelegt. Zum Verlassen bzw. frühzeitigen erneuten Ausführen von Schleifen dienen die Schlüsselwörter `break` bzw. `continue`. Schleifen erlauben zudem Verschachtelungen.

**Seaborn:** Seaborn ist eine externe Python-Bibliothek für Datenvisualisie-

rungszwecke. Seaborn basiert auf der älteren, grundlegenden Matplotlib-Bibliothek, kommt jedoch mit einer reduzierten Syntax aus, ist intuitiver im Gebrauch und liefert ästhetischere Plots. Seaborn wird meist als Ergänzung, nicht als Ersatz für Matplotlib verwendet.

**Setter:** Methode innerhalb einer Klasse, mithilfe derer sich der Wert eines Attributs überschreiben bzw. ändern lässt.

**Stack Overflow:** [Stackoverflow.com](http://Stackoverflow.com) ist eine Internetplattform für Softwareentwicklung, auf der Nutzerinnen und Nutzer Fragen stellen und in beantworteten Beiträgen recherchieren können.

**Standardbibliothek:** Pythons Standardbibliothek vereint wesentliche Komponenten, Funktionalitäten und Module für die wichtigsten Anwendungsbereiche des Programmierens. Die über 200 Kernmodule lassen sich durch den `import`-Befehl innerhalb eines Python-Programms verwenden.

**String** (auch *Zeichenkette*): Der String-Datentyp wird in Python mit `str` angegeben. Strings sind stets in einfache oder doppelte Anführungszeichen gesetzt und können einzelne Buchstaben, Wörter, ganze Sätze oder längere Texte aufnehmen.

**Syntax:** Bezeichnet die den Satzbau regelnden formalen Vorschriften einer Programmiersprache. Die fest vorgegebene Syntax-Struktur gibt an, welche Kombinationen von Symbolen (Zeichenketten, Satzzeichen, Sätzen, etc.) einen korrekten Ausdruck innerhalb der Programmiersprache darstellen.

## Glossar

**Tupel:** Eine wichtige Datenstruktur, in der die einzelnen Elemente in runden Klammern angegeben sind. Tupel haben große Ähnlichkeit mit Listen, sind im Unterschied zu diesen jedoch unveränderlich, was sich insbesondere bei größeren Datenmengen vorteilhaft auf die Rechenleistung auswirkt.

**Variable:** Ähnlich wie Variablen in der Mathematik enthalten Variablen beim Programmieren spezifische Informationen, auf die mittels ihres Namens zugegriffen werden kann. Somit lassen sich darauf im Laufe des Programms weitere Operationen und Berechnungen ausführen. Variablen können nicht nur aus unterschiedlichen Zahlentypen, sondern ebenso aus Buchstaben oder anderen Datenstrukturen bestehen.

**Vererbung:** Ein wichtiges Prinzip der objektorientierten Programmierung, bei dem die Eigenschaften und Methoden einer sogenannten Oberklasse bei der Definition einer neuen Unterklassse automatisch auf diese übertragen werden. Die abgeleitete Unterklassse „erbt“ sozusagen die Struktur der übergeordneten Oberklasse.

**Wahrheitswert:** Auch boolesche Variablen genannt. Die beiden möglichen Werte, True oder False, dienen insbesondere der Formulierung und Auswertung logischer Ausdrücke. In Python ist allen Ausdrücken und Objekten automatisch ein bestimmter Wahrheitswert zugeordnet.

# Stichwortverzeichnis

## A

Absolutfunktion (fabs).....	162
Anaconda.....	377
Navigator .....	377
ArithmetError.....	221
arithmetischer Operator.....	87
Array	
append .....	392
bearbeiten.....	390
Broadcasting.....	392
Dimension (ndim) .....	390
durchsuchen mit where .....	393
erstellen.....	389
filtrn.....	394
shape-Attribut.....	390
Slicing.....	391
sortieren.....	394
statistische Methoden .....	395
as-Klausel.....	228
Attribut .....	175, 182
Datenkapselung .....	197
Klassenattribut.....	177, 186
Name.....	182
Objektattribut .....	185
Wert.....	182
Ausnahme.....	214
Ausnahmebehandlung.....	218
eigene Ausnahmen definieren .....	225
else.....	221
raise .....	226
unbekannte Ausnahmen .....	224

## B

Binärkode.....	16
Bit.....	16
boolescher Operator .....	90
Bootstrap .....	359
break-Anweisung.....	122

## C

Compiler.....	17
continue-Anweisung.....	124
CRUD (Create, Read, Update & Delete) .....	252

## D

Dataframe	
bearbeiten.....	386
describe.....	389
filtrn.....	388
head.....	386
index.....	386
loc .....	387
shape-Attribut.....	386
sortieren.....	387
Datenanalyse.....	376
Datenbank .....	252
aktualisieren .....	264
erstellen.....	256
Joins.....	262
lesen.....	259
löschen .....	265
NoSQL-Datenbank.....	267
relationale .....	253
SQL.....	253
SQLite .....	254, 256, 258
XML-Datenbank .....	270
Datenbankmanagementsystem (DBMS) .....	252
relationales (RDBMS) .....	254
Datenkapselung .....	196
Datenspeicherung in Textdateien.....	237
Daten lesen .....	237
Daten schreiben .....	243
weitere Optionen .....	246
Datenstruktur.....	65
Dictionary .....	72
Liste .....	66
Menge.....	78
Tupel.....	75

## Stichwortverzeichnis

Datentyp .....	58	<b>F</b>	
bool (boolescher Wahrheitswert) .....	58, 91	Fakultät .....	147
float (Fließkommazahl) .....	58	factorial() .....	162
int (ganze Zahl) .....	58	Fehler .....	214
str (String) .....	58	Laufzeitfehler .....	215
überprüfen mit type() .....	60	lexikalische .....	215
umwandeln.....	59	Syntaxfehler.....	215
Debugger .....	228	File (Klasse).....	237
Dictionary.....	72	FileNotFoundException .....	238, 248
erstellen (dict).....	74	finally.....	224
Felder löschen (del).....	74	Fließkommazahl .....	55
Schlüssel-Werte auflisten (list) .....	74	for-Schleife .....	118
zusammenfügen (update) .....	73	from modul import .....	149
Disjunktion (or).....	90	Funktion .....	134
Django		append() .....	70
admin-Seite .....	366	Argument .....	134, 135
App .....	356	Aufbau .....	135
Bootstrap .....	359	clear().....	69, 74
Datenbank .....	363	connect().....	254
erste Webseite erstellen.....	347	definieren.....	135
installieren.....	343	del() .....	69
MVT-Architektur .....	363	dir() .....	192
Template .....	349, 352, 354	eingebaute.....	134
Webserver starten.....	344	eval() .....	53
Django Template Language (DTL) .....	352	fabs() .....	162
Docstring .....	179	factorial() .....	162
anzeigen .....	180	globale Variable .....	139
doc-Attribut .....	181	help() .....	181
dynamische Typisierung.....	13, 48, 55	in eigene Datei schreiben .....	148
<b>E</b>		input() .....	52
Eingabeaufforderung .....	31	len() .....	86, 123, 141
einseitige Verzweigung .....	100	lokale Variable.....	139
elif-Anweisung.....	108	lower() .....	60
else-Anweisung.....	106, 117	max() .....	140
bedingter Ausdruck .....	107	mehrere Parameter.....	139
E-Mail		min() .....	140
empfangen .....	332	open() .....	238
Funktionsweise .....	326	optionale Parameter .....	143
mit Anhang versenden .....	331	Parameter .....	135
versenden.....	327	Potenz (pow) .....	163
Endlosschleife .....	116	print() .....	36, 49
Exception (Klasse).....	215, 225	property() .....	202
		range() .....	120

rekursive .....	146
remove() .....	69, 247
render() .....	350
Rückgabewert (return) .....	143
strip() .....	239
type().....	60
update() .....	73
upper().....	59
variable Anzahl an Übergabewerten....	142
variable Parameteranzahl.....	140
verbindliche Parameter.....	143
<b>G</b>	
Getter .....	198
GitHub .....	369
Gleichheitsoperator.....	102, 104
GUI.....	277
aus exe-Datei starten.....	297
Bibliotheksprogramm.....	308
Dialog.....	280
Widget.....	277, 298
<b>H</b>	
Hallo-Welt-Programm.....	36
Heatmap .....	406
HTML .....	359
<b>I</b>	
IDE	
siehe Integrierte Entwicklungsumgebung...	
.....	19
if-Anweisung.....	100
Bedingungen verknüpfen.....	104
Vergleiche mit.....	101
import-Anweisung.....	149
import this.....	167
Instanz.....	182
Integrated Development and Learning Environment (IDLE) .....	19
Integrierte Entwicklungsumgebung (IDE)	18
interaktiver Modus.....	30
Interpreter	
Programm ausführen.....	38
interpretierte Programmiersprache .....	17
<b>J</b>	
JupyterLab .....	379
<b>K</b>	
Klasse .....	175, 177
Definition .....	178
Konstruktor.....	178
Methode.....	188
Oberklasse .....	203
Unterklasse .....	203
Klassenattribut.....	177, 185
Kommandozeileninterpreter .....	31
Kommentar .....	39
Docstring.....	179
kompilierte Programmiersprache .....	17
Konjunktion (and).....	90
Konstruktor .....	178
<b>L</b>	
len() .....	123
Liste .....	66
Elemente ändern.....	69
Elemente hinzufügen (append).....	70
Elemente löschen (del) .....	69
Elemente löschen (remove).....	69
Index .....	67
Inhalt löschen (clear) .....	69
leere .....	68
Operationen mit .....	69
Teilbereich .....	68
Verschachtelungen von.....	71
<b>M</b>	
magische Methode .....	193
__add__ .....	194
__str__ .....	194
Maschinencode.....	16
math-Modul .....	162
Matplotlib .....	396
Mehrfachvererbung.....	206
Member	
geschütztes.....	197
öffentliches .....	197
privates.....	197

## Stichwortverzeichnis

Menge.....	78	Käse-Sketch .....	208
Differenz.....	79	Spam.....	326
Element entfernen (remove).....	81	<b>N</b>	
Element hinzufügen (add) .....	81	Negation (not).....	90
erstellen (set).....	78	NumPy.....	376, 389
frozenset .....	79	random-Modul.....	405
größtes Element (max).....	79	to_numpy .....	405
kleinstes Element (min).....	79		
Schnittmenge .....	79	<b>O</b>	
Vereinigung.....	79	Objekt .....	174
Methode		Attribut .....	175, 185
close().....	238	Instanz.....	182
execute() .....	256	instantiiieren.....	182
fetch().....	334	Methode.....	175
read().....	239	objektorientierte Programmierung (OOP) .....	174
readline() .....	240	Klasse .....	176
sendmail() .....	329	Vererbung.....	176, 203
write() .....	244	Open Source.....	15
Methode (einer Klasse).....	188	Operator .....	87
Getter .....	198	arithmetischer.....	87
__init__().....	178	boolescher .....	90
Instanzmethode .....	190	Überladung .....	195
Konstruktormethode .....	178	Vergleichsoperator .....	88
magische .....	193	Zugehörigkeitsoperator .....	92
Setter .....	198	OverflowError.....	221
statische .....	188		
Model-View-Controlling (MVC) .....	363	<b>P</b>	
Modul.....	148	pandas .....	376, 380
eigene Module verwenden .....	149	Dataframe	
email.....	329	bearbeiten.....	386
imaplib .....	332	Dataframe erstellen .....	381
math .....	162	Daten ein- und auslesen .....	385
os.....	247	pass-Anweisung.....	127
PyQt5 .....	280	PEP .....	38
random .....	164	pip install .....	168
smtplib .....	327	Polymorphie.....	195, 205
sqlite3 .....	254	PostgreSQL.....	254
Standardbibliothek .....	156	Programmierparadigma .....	174
sys .....	224, 228	PyCharm.....	19
Teile importieren .....	149	anpassen.....	27
time.....	158	Debugger .....	228
xml.....	271	Eigenschaften .....	22
Modulo-Operation.....	125	Einrückung einstellen .....	101
fmod() .....	164	Installation.....	24
Monty Python .....	13, 15, 212		
Argument .....	136		

neues Projekt anlegen .....	35
PyQt .....	277
Fenster erstellen .....	281
installieren .....	278
Python .....	9
Bibliothek .....	156
Interpreter .....	17
Interpreter installieren .....	19
Prompt .....	21, 30
Python Package Index (PyPI) .....	168
Python Virtual Machine (PVM) .....	17
Shell .....	30
Standardbibliothek .....	156
The Zen of Python .....	14, 167
Versionen .....	13
Python Package Index (PyPI) .....	168
<b>Q</b>	
Qt Designer .....	284
Fenstereigenschaften .....	288
Fenster in PyCharm verwenden .....	291
Fensterlayout .....	286
Widget .....	298
<b>R</b>	
raise .....	226
random-Modul .....	164
range() .....	120
Rekursion .....	146
return-Anweisung .....	143, 145
<b>S</b>	
Schleife .....	115
Endlosschleife .....	116
for .....	118
Verschachtelung .....	125
while .....	115, 127
Zählschleife .....	121
Seaborn .....	376, 395
Daten einlesen .....	396
Plottingfunktionen .....	399, 401
Setter .....	198
Signale und Slots .....	293
SMTP (Simple Mail Transfer Protocol) .....	327
Spyder .....	19
SQL .....	253
SQLite .....	254, 256, 258
Stackoverflow .....	11
Standardbibliothek .....	156, 157, 158
String .....	13, 82
Anzahl der Elemente (len) .....	86
formatieren .....	84
Großbuchstaben (upper) .....	87
Kleinbuchstaben (lower) .....	87
Konkatenation .....	84
löschen (del) .....	83
strip() .....	87
Zugehörigkeitsoperator .....	85
Syntax .....	12
<b>T</b>	
Tag (XML) .....	270
Template .....	349
Texteditor .....	18
time-Modul .....	158
Tkinter .....	277
try ... except .....	218
Tupel .....	75
leeres .....	77
Werte hinzufügen .....	77
<b>U</b>	
Überladung .....	195
<b>V</b>	
ValueError .....	217, 220
Variable .....	12, 32
Name .....	47, 48
Namensgebung .....	48
Wert durch Nutzereingabe .....	52
Variablentyp .....	55
Vererbung .....	176, 203
Mehrfachvererbung .....	206
Vergleichsoperator .....	88, 103
virtuelle Umgebung .....	341
aktivieren .....	342
von Rossum, Guido .....	13

## Stichwortverzeichnis

### W

Wahrheitswert.....	49
Web-Framework.....	340
while-Schleife.....	115, 127
with.....	240
World Wide Web (WWW) .....	340

### Z

Zählschleife .....	121
ZeroDivisionError.....	217, 220
Zugehörigkeitsoperator .....	92
Zuweisungsoperator....	47, 49, 51, 102
zweiseitige Verzweigung.....	106

## Arduino-Kompendium: Elektronik, Programmierung und Projekte (572 Seiten)



Die Arduino-Plattform bietet Bastlern die Möglichkeit auch ohne umfassende Elektronik- und Programmierkenntnisse eigene intelligente Mikrocontrollerprojekte umzusetzen. Mit diesem Buch lernen Sie praxisnah die notwendigen Grundlagen im Bereich des Programmierens und der Elektronik, um bald eigene spannende Projekte mit dem Arduino basteln zu können. Neben allen relevanten Zubehörteilen wie Sensoren, Aktoren, Displays und Shields werden auch fortgeschrittene Themen wie moderne MQTT Smart Home Systeme, Arduino-Roboter und die Datenverarbeitung und Steuerung über Processing-Programme am PC behandelt. So können Sie die Möglichkeiten des Arduino voll ausnutzen!

### Inhalte des Buchs

- ▶ Grundlagen der Elektronik verständlich erklärt
- ▶ Alle wichtigen Sensoren, Aktoren, Displays, Shields und Zubehörteile umfassend vorgestellt
- ▶ Arduino-Programmierung über die Arduino IDE
- ▶ Datenverarbeitung und Steuerung über Processing-Programme am PC
- ▶ Arduino mit dem Internet verbinden, MQTT-Internetanwendungen
- ▶ Erstellung eigener Platinen, um Projekte für den professionellen Einsatz zu entwickeln
- ▶ Effizientes Debugging, um Fehler schnell zu beheben

### Vorteile dieses Buchs

- ▶ Solides Hintergrundwissen für eigene Projekte durch Erläuterung aller Elektronik- und Programmiergrundlagen
- ▶ Einfache, praxisnahe Erklärungen tragen zum schnellen Verständnis bei
- ▶ Einsatzbeispiele helfen, das Gelernte anzuwenden und sichern den nachhaltigen Lernerfolg
- ▶ Umfangreiche Praxisprojekte wie Arduino-Roboter, Smart Home Anwendungen und Arduino-Wecker dienen als Vorlagen für eigene Projekte
- ▶ Alle Schaltpläne und Quellcodes kostenfrei zum Download verfügbar

Hier informieren: <https://bmu-verlag.de/arduino-kompendium/>

## Weitere Bücher vom BMU Verlag

### Raspberry Pi Kompendium: Linux, Programmierung und Projekte (516 Seiten)



Der Raspberry Pi hat die Welt der Bastler revolutioniert: Trotz seiner nur kreditkartengroßen Form ist er ein vollwertiger Mini-Computer und bietet nahezu unbegrenzte Möglichkeiten für spannende Projekte in verschiedensten Bereichen. Mit diesem Buch lernen Sie praxisnah die notwendigen Grundlagen des Betriebssystems Linux zur effizienten Arbeit mit dem Pi, die Programmierung mit Python sowie die Grundlagen der Elektronik. Den Abschluss des Buchs bilden umfassende Beispielprojekte zum Nachbauen. So können Sie die Möglichkeiten des Raspberry Pi voll ausnutzen und eigene Ideen umsetzen!

#### Inhalte des Buchs

- ▶ Setup und Inbetriebnahme des Raspberry Pi
- ▶ Anschauliche Einführung in die Arbeit mit dem Betriebssystem Linux
- ▶ Grundlagen der Programmierung in Python verständlich erklärt, inklusive Programmen mit grafischen Oberflächen und Webanwendungen
- ▶ Umfassende Behandlung der Grundlagen der Elektronik
- ▶ Vorstellung vieler verschiedener Bauteile wie Sensoren, Motoren, Displays und Zusatzboards
- ▶ Anwendung des Gelernten in zehn großen Praxisprojekten

#### Vorteile dieses Buchs

- ▶ Solides Hintergrundwissen für eigene Projekte durch Erläuterung aller Elektronik-, Programmier- und Linux-Grundlagen
- ▶ Einfache, praxisnahe Erklärungen tragen zum schnellen Verständnis bei
- ▶ Einsatzbeispiele helfen, das Gelernte anzuwenden und sichern den nachhaltigen Lernerfolg
- ▶ Umfangreiche Praxisprojekte wie ein Roboter, Smart Mirror oder eine Wetterstation mit Webinterface dienen als Vorlagen für eigene Projekte
- ▶ Alle Schaltpläne, Quellcodes und eBook-Version kostenfrei zum Download verfügbar

Hier informieren: <https://bmu-verlag.de/raspi-kompendium/>

**C++ Programmieren für Einsteiger: Der leichte Weg zum C++-Experten (278 Seiten)**



Beginnend mit den Grundlagen der Programmierung wird die Programmiersprache C++ vermittelt, ohne, dass dabei Vorkenntnisse vorausgesetzt werden. Besonderer Fokus liegt dabei auf Objektorientierter Programmierung und dem Erstellen grafischer Oberflächen mit Hilfe von MFC.

Auch auf C++ Besonderheiten, wie die Arbeit mit Zeigern und Referenzen, wird ausführlich eingegangen. Jedes Kapitel beinhaltet Übungsaufgaben, durch die man das Gelernte direkt anwenden kann. Nach dem Durcharbeiten des Buches kann der Leser eigene komplexe C++ Anwendungen inklusive grafischer Oberflächen erstellen.

2. Auflage: aktualisiert und erweitert

Hier informieren: [http://bmu-verlag.de/cpp\\_programmieren/](http://bmu-verlag.de/cpp_programmieren/)

Weitere Bücher vom BMU Verlag

**Python 3 Programmieren für Einsteiger: Der leichte Weg zum Python-Experten  
(310 Seiten)**



Python ist eine weit verbreitete, universell einsetzbare und leicht zu erlernende Programmiersprache und eignet sich daher bestens zum Programmieren lernen!

In diesem Buch wird das Programmieren in Python beginnend mit den Grundlagen leicht und verständlich erklärt, ohne dass dabei Vorkenntnisse vorausgesetzt werden. Ein besonderer Fokus wird dabei auf die Objektorientierte Programmierung (OOP) und das Erstellen von grafischen Oberflächen gelegt.

Jedes Kapitel beinhaltet Übungsaufgaben, durch die man das Gelernte direkt anwenden kann. Nach dem Durcharbeiten des Buches kann der Leser eigene komplexere Python Anwendungen inklusive grafischer Oberfläche programmieren.

2. Auflage: aktualisiert und erweitert

Hier informieren: <http://bmu-verlag.de/python/>