

HANSER



Leseprobe

zu

„Python 3“

von Heiko Kalista

ISBN (Buch): 978-3-446-45469-9

ISBN (E-Book): 978-3-446-45689-1

Weitere Informationen und Bestellungen unter
<http://www.hanser-fachbuch.de/978-3-446-45469-9>

sowie im Buchhandel

© Carl Hanser Verlag, München

Inhalt

Vorwort	XV
1 Grundlagen	1
1.1 Hältst Du das richtige Buch in den Händen?	1
1.2 Dieses Buch bricht mit einigen Konventionen!	2
1.3 Die Arbeit mit diesem Buch	3
1.4 Das Kapitel zu Minecraft	4
1.5 Das Begleitmaterial zum Buch	4
1.6 Anregungen? Kritik? Forum?	5
1.7 Die Geschichte von Python in 120 Wörtern	5
1.8 Was kann man mit Python machen (und was nicht)?	6
1.9 Interpreter vs. Compiler	7
1.10 Python 2.7 oder 3.7?	8
1.11 Die Entwicklungsumgebung PyCharm	9
1.11.1 Alternativen zu PyCharm	10
1.12 Python-Interpreter installieren	10
1.12.1 Python-Interpreter unter Windows installieren	11
1.12.2 Python-Interpreter unter macOS installieren	12
1.12.3 Python-Interpreter unter Linux installieren	12
1.12.4 PyCharm unter Windows installieren	13
1.12.5 PyCharm unter macOS installieren	13
1.12.6 PyCharm unter Linux installieren	14
1.12.7 PyCharm einrichten	15
1.13 Genug geredet, los gehts!	17
1.13.1 Das erste Programm eingeben	18
1.13.2 Ausführen des ersten Beispiels	20
1.13.3 Laden der Beispiele	20
1.13.4 Der Quelltext im Detail	22
1.13.5 Kommentare im Detail	23
1.14 PEP8 – um sie ewig zu binden	25

1.15	Python-Programme ohne Entwicklungsumgebung starten	27
1.15.1	Python-Programme unter Windows starten	27
1.15.2	Python-Programme unter Linux oder macOS starten	28
1.16	Python interaktiv	29
1.17	Aufgabenstellung	31
1.18	Kurz & knapp	31
2	Variablen	33
2.1	Was sind Variablen?	33
2.2	Statisch typisiert vs. dynamisch typisiert	34
2.3	Einige Details vorab – Variablen sind auch nur Namen	36
2.4	Richtlinien für Variablennamen	38
2.4.1	Von Kamelen, Schlangen und Pascal	38
2.4.2	Sinnvolle Variablennamen	39
2.5	Rechnen mit Variablen	40
2.5.1	Verkürzte Schreibweise bei Rechenoperationen.	41
2.5.2	Gibt es noch weitere Datentypen?	42
2.5.3	Konvertierung von Datentypen.	43
2.5.4	Besonderheiten bei der Konvertierung von Datentypen	44
2.6	Formatierte Ausgabe mit print()	45
2.7	Genauigkeit von Fließkommazahlen	47
2.7.1	Formatierte Ausgabe von Fließkommazahlen.	48
2.8	Den ganzzahligen Rest einer Division bestimmen	49
2.9	Konstanten, Konventionen und Restriktionen	50
2.10	Fehlerquelltext	51
2.10.1	Was ist die Aufgabe des Programms?	52
2.10.2	Lösungsvorschlag.	52
2.10.3	Die gemeinen Kleinigkeiten	54
2.11	Aufgabenstellung	55
2.11.1	Einige Tipps	55
2.11.2	Lösung zur Aufgabenstellung.	56
2.12	Kurz & knapp	57
3	Schleifen und Bedingungen	59
3.1	Was sind Schleifen und Bedingungen?	59
3.2	Die if-Bedingung	60
3.2.1	Blöcke und Einrückungen.	61
3.2.2	Arbeit sparen mit else.	62
3.2.3	elif	64
3.3	Der Platzhalter pass	66
3.4	Blöcke im interaktiven Modus	67
3.5	Logische Operatoren	67

3.6	PyCharm – Korrekturvorschläge übernehmen	70
3.7	while-Schleifen	71
3.7.1	Eine Schleife mit break vorzeitig beenden	72
3.7.2	continue in while-Schleifen.....	74
3.7.3	while-else	75
3.8	for-Schleifen	76
3.8.1	Eine einfache Zählschleife.....	77
3.8.2	Ein Wort zu Schleifenvariablen	78
3.8.3	Die Funktion range() im Detail.....	79
3.8.4	break und continue in for-Schleifen.....	80
3.8.5	Sonderfall Unterstrich: die Wegwerfvariable	81
3.9	Ein kurzer Abstecher: Exceptions	82
3.10	Fehlerquelltext	83
3.10.1	Was ist die Aufgabe des Programms?	84
3.10.2	Lösung zum Fehlerquelltext.....	85
3.11	Aufgabenstellung	86
3.11.1	Einige Tipps	87
3.11.2	Lösungsvorschlag.....	88
3.12	Ein umfangreicheres Beispiel: Zahlenraten	89
3.12.1	Die Hauptschleife.....	92
3.12.2	Ein neues Spiel starten	92
3.12.3	Abfrage des Schwierigkeitsgrades und Beenden des Spiels	93
3.13	Kurz & knapp	93
4	Funktionen	97
4.1	Was sind Funktionen?	97
4.2	Eine einfache Funktion definieren	97
4.3	Parameter, Argumente und Rückgabewert	99
4.4	Weitere Möglichkeiten, Funktionen aufzurufen	101
4.5	Globale und lokale Variablen	103
4.5.1	Ein bisschen mehr Verwirrung, bitte!	105
4.5.2	Das Schlüsselwort global.....	106
4.5.3	Auch Parameter sind lokal	108
4.5.4	Ein Wort zu globalen Variablen	109
4.5.5	Von Schatten und Hauptprogrammen	109
4.6	Standardwerte für Parameter	113
4.7	Schlüsselwortparameter	114
4.8	Wann sollten Funktionen zum Einsatz kommen?	116
4.9	Aufgabenstellung	117
4.9.1	Lösung zur Aufgabenstellung.....	118
4.10	Vorgehensweise bei größeren Projekten	121
4.11	Rekursion	122

4.12	Fehlerquelltext	123
4.12.1	Was ist die Aufgabe des Programms?	124
4.12.2	Lösung zum Fehlerquelltext	125
4.13	Kurz & knapp	127
5	Klassen	131
5.1	Was ist Objektorientierung?	131
5.2	Eine einfache Klasse definieren	132
5.2.1	Aufbau einer Klasse	133
5.2.2	Erzeugen von Objekten	134
5.2.3	Verwenden der Objekte	135
5.3	Kontrollierter Zugriff auf Attribute: Properties	136
5.3.1	Getter	136
5.3.2	Setter	139
5.3.3	Wann sollten Attribute und wann Properties verwendet werden?	140
5.3.4	Nachträgliches Umstellen auf Properties	141
5.4	Dynamische Attribute	142
5.5	Klassenattribute	145
5.5.1	Stolpersteine bei Klassenattributen	147
5.6	Statische Methoden	149
5.7	Zwischenstand: Braucht man das wirklich?	151
5.8	Aufgabenstellung	152
5.8.1	Einige Tipps	153
5.8.2	Lösungsvorschlag	154
5.9	Das Gleiche ist nicht dasselbe	156
5.9.1	Für Fortgeschrittene: Parameterübergabe im Detail	159
5.10	None	161
5.11	Vererbung	163
5.11.1	Ein einfaches Beispiel	164
5.11.2	Überschreiben von Methoden	166
5.12	Wie das Smartphone erfunden wurde – oder: Mehrfachvererbung	171
5.13	Für Fortgeschrittene: Binden von Methoden	174
5.13.1	Ein Blick hinter die Kulissen	174
5.13.2	Klassen um eigene Funktionen erweitern	176
5.13.3	Statische Methoden und instanzbezogene Bindung	178
5.14	Überschreiben der Methode <code>__str__()</code>	182
5.15	Und wo ist der Fehlerquelltext?	183
5.16	Kurz & knapp	183
6	Container	189
6.1	Was sind Container?	189
6.1.1	Eine Anmerkung, bevor es losgeht	189

6.2	Listen	190
6.2.1	Listen erzeugen und auf Elemente zugreifen	190
6.2.2	Listen dynamisch erzeugen	191
6.2.3	Elemente löschen oder ersetzen	193
6.2.4	Aufgabenstellung	196
6.2.4.1	Einige Tipps	197
6.2.4.2	Lösung zur Aufgabenstellung	197
6.2.5	Sortieren von Listen	198
6.2.5.1	Eigene Sortierfunktionen	200
6.2.5.2	Eine weitere Möglichkeit der Sortierung: sorted	202
6.2.5.3	Für Fortgeschrittene: lambda – anonyme Funktionen	203
6.2.6	Listen verknüpfen	206
6.2.7	Nicht nur für Fortgeschrittene: tiefes und flaches Kopieren	207
6.3	Tupel	209
6.3.1	Unterschiede zu Listen	209
6.3.2	Vorsicht: Instanzen in Tupeln können geändert werden	211
6.3.3	Mehrere Rückgabewerte mit Tupeln	212
6.3.4	Für Fortgeschrittene: namedtuple	214
6.4	Strings	216
6.4.1	Für Fortgeschrittene: Darum sind Strings unveränderlich	217
6.4.2	Slicing	219
6.4.3	Arbeiten mit Strings	221
6.4.4	Strings verbinden	221
6.4.5	Zerlegen und wieder zusammensetzen: split und join	223
6.4.6	Strings „aufbereiten“	225
6.4.7	Ändern der Groß- und Kleinschreibung	227
6.4.8	Strings durchsuchen	228
6.4.9	Aufgabenstellung	230
6.4.9.1	Einige Tipps	231
6.4.9.2	Lösungsvorschlag	231
6.4.10	Ersetzungen durchführen	233
6.5	Dictionaries	235
6.5.1	Grundlagen von Dictionaries	235
6.5.2	Vorsicht beim Zugriff!	236
6.5.3	Dictionaries durchlaufen und verändern	238
6.5.4	Elemente aus einem Dictionary entfernen	240
6.6	Fehlerquelltext	242
6.6.1	Was ist die Aufgabe des Programms?	242
6.6.2	Lösung zum Fehlerquelltext	243
6.7	Sets und Frozensets	245
6.7.1	Einfache Sets/Frozensets erzeugen	245
6.7.2	Sets: Elemente hinzufügen und entfernen	247
6.7.3	Mengenoperationen	248
6.8	Kurz & knapp	252

7	Exceptions	261
7.1	Was sind Exceptions?	261
7.2	Fehler? Die passieren mir doch nicht!	261
7.3	Die Mechanik von Exceptions	263
7.4	Abfangen unterschiedlicher Exceptions	265
7.5	Alle Exceptions fangen	267
7.6	Eigene Exceptions	269
7.7	else und finally	272
7.8	Einige allgemeine Tipps	275
7.9	Kurz & knapp	279
8	Module und Pakete	283
8.1	Module	283
8.1.1	Grundlagen	284
8.1.2	Dokumentation von Modulen	286
8.1.3	Umbenennen eines Namensraums	288
8.1.4	Selektives Importieren	289
8.1.5	Namensräume haben ihren Sinn	290
8.1.6	Batteries included!	290
8.1.7	Reihenfolge beim Importieren	291
8.1.8	Eigene Module in mehreren Projekten nutzen	292
8.1.9	Tipps für das Schreiben von Modulen	292
8.1.10	Automatische Ausführung beim Import verhindern	294
8.2	Pakete	295
8.2.1	Importieren von Modulen aus Paketen	296
8.2.2	Reguläre Pakete	297
8.2.3	Namespace Packages	298
8.2.4	Unterscheidung der Paketarten	299
8.3	Kurz & knapp	300
9	Dateien und Dateisystem	303
9.1	Lesen und Schreiben von Dateien	303
9.1.1	Eine einfache Textdatei auslesen	304
9.1.2	Fehler beim Öffnen von Dateien	305
9.1.3	Eine Datei schrittweise auslesen	306
9.1.4	Zeilenweises Auslesen von Textdateien	308
9.1.5	Textdateien schreiben	309
9.1.6	Übersicht möglicher Modi der Funktion open()	311
9.1.7	Aufgabenstellung	312
9.2	Dateien und Dateisystem	312
9.2.1	Verzeichnisse	313
9.2.2	Pfade und Prüfung auf deren Existenz	314
9.2.3	Pfade und Plattformunabhängigkeit	316

9.2.4	Verzeichnisse und Dateien erzeugen und löschen	317
9.2.5	Verzeichnisstrukturen erzeugen und löschen	320
9.2.6	Umbenennen von Verzeichnissen und Dateien	322
9.2.7	Kopieren und Verschieben	324
9.3	Kurz & knapp	326
10	GUI-Programmierung mit tkinter	331
10.1	Warum gerade tkinter?	331
10.2	Die Layout-Manager	332
10.2.1	Absolute Positionierung	332
10.2.2	Der pack-Manager	333
10.2.3	Der grid-Manager	338
10.2.4	Welcher Manager soll es sein?	340
10.3	Flexible Fenstergröße	341
10.4	Konfiguration von Widgets	344
10.5	Steuerelemente im Detail	346
10.5.1	Buttons	346
10.5.2	Kontrollvariablen	348
10.5.3	Eingabefelder	349
10.5.4	Textboxen	350
10.5.5	Checkboxen	354
10.5.6	Radiobuttons	356
10.5.7	Aufgabenstellung	357
10.5.8	Lösungsvorschlag	358
10.5.9	Listboxen	359
10.5.10	Listbox mit Scrollbalken	362
10.6	Fehlerquelltext	363
10.6.1	Was ist die Aufgabe des Programms?	363
10.6.2	Lösung zum Fehlerquelltext	364
10.7	Dialogfenster	366
10.8	Menüs	369
10.8.1	Einfache Menüs	369
10.8.2	Untermenüs	370
10.9	Kurz & knapp	372
11	Debugging	377
11.1	Was ist ein Debugger?	378
11.2	Eine einfache Fehlersuche	378
11.2.1	Haltepunkte setzen und entfernen	379
11.2.2	Das Programm durchlaufen und Werte betrachten	380
11.2.3	Geht es auch ohne Haltepunkte?	383
11.2.4	Interaktive Fehlersuche mit der Python-Konsole	384
11.3	Debuggen von Funktionen und Methoden	386

11.3.1	In Funktionen springen.	387
11.3.2	Abschnitte überspringen und Funktionen verlassen.	388
11.3.3	Clever springen	389
11.4	Watches	390
11.5	Haltepunkte im Detail	392
11.5.1	Ein Beispiel zum Experimentieren.	392
11.5.2	Verwalten von Haltepunkten	393
11.5.3	Unterbrechen oder nicht?	397
11.5.4	Bedingte Haltepunkte	397
11.5.5	Protokollierung.	398
11.5.6	Temporäre Haltepunkte.	399
11.5.7	Verkettete Haltepunkte	400
11.5.8	Haltepunkte für Exceptions.	400
11.5.9	Kombination der Optionen	401
11.6	Einsatz in der Praxis	402
12	Versionsverwaltung mit Git	403
12.1	Der vermeintlich leichte Weg	403
12.2	Wie funktionieren Versionskontrollsysteme?	404
12.2.1	Verallgemeinerte Arbeitsweise.	404
12.2.2	Zentrale und verteilte Versionskontrolle.	405
12.3	Was ist Git?	405
12.4	Interessiert mich nicht, ich arbeite alleine!	407
12.5	Vorbereitungen	407
12.5.1	Git unter Windows installieren.	407
12.5.2	Git unter macOS installieren	408
12.5.3	Git unter Linux installieren	409
12.5.4	GitHub.	409
12.6	Los geht's – Git lokal verwenden	410
12.6.1	Ein bestehendes Projekt unter Versionskontrolle stellen	410
12.6.2	Dateien hinzufügen	412
12.6.3	Commit/Einchecken.	413
12.6.4	Änderungen vornehmen und überprüfen.	414
12.6.5	Änderungen rückgängig machen.	416
12.6.6	Betrachten der Historie	417
12.6.7	Zu älteren Versionen zurückkehren – Möglichkeit 1.	419
12.6.8	Dateien ignorieren	420
12.7	Zusammenarbeit über ein Remote Repository	421
12.7.1	Projekt auf GitHub veröffentlichen	422
12.7.2	Ein Repository klonen.	422
12.7.3	Änderungen pushen	424
12.7.4	Pull.	424
12.7.5	Wer hat's erfunden?.	425

12.7.6	Automatisches Merging.....	425
12.7.7	Mergen und Konflikte beheben	426
12.8	Branching	430
12.8.1	Um was geht es?.....	430
12.8.2	Einen Branch erzeugen und damit arbeiten	431
12.8.3	Zwischen Branches wechseln.....	433
12.8.4	Branches mergen.....	433
12.8.5	Branches pushen	435
12.8.6	Fetch	435
12.8.7	Zu älteren Versionen zurückkehren – Möglichkeit 2.....	436
12.9	Weitere nützliche Features	436
12.9.1	Stashing – Änderungen temporär speichern	437
12.9.2	Commits korrigieren	439
12.9.3	Tagging	439
12.10	Noch ein paar Tipps	440
12.10.1	Zwei einfache Branching-Modelle	440
12.10.2	Atomare Commits	441
12.10.3	Test-Repository griffbereit halten.....	442
12.10.4	Andere Git-Clients	442
12.11	Kurz & knapp	443
13	Minecraft auf dem Raspberry Pi	447
13.1	Um was geht es in diesem Kapitel?	447
13.1.1	Der Raspberry Pi – ein kleines Kraftpaket	447
13.1.2	Minecraft Pi	448
13.1.3	Was wird benötigt?.....	449
13.2	Der Sprung ins kalte Wasser	450
13.3	Die Entwicklungsumgebungen	450
13.4	Den Raspberry Pi einrichten	451
13.4.1	Erstellen der SD-Karte.....	451
13.4.2	Einstellen des Tastaturlayouts	452
13.4.3	Minecraft starten	452
13.4.4	Die Steuerung	452
13.5	Der erste Testlauf	454
13.5.1	Arbeiten mit IDLE	454
13.5.2	Arbeiten mit Thonny	456
13.6	Das Begleitmaterial	457
13.7	Die Minecraft Python API	457
13.7.1	Quellen und weitere Informationen:	458
13.7.2	Eine Übersicht der Minecraft API	458
13.7.3	Die Klasse Minecraft	458
13.7.4	Die Klasse CmdCamera.....	461
13.7.5	Die Klasse CmdPlayer	462

13.7.6	Die Klasse CmdEntity	462
13.7.7	Die Klasse Block	463
13.7.7.1	Eine Übersicht der wichtigsten Blöcke	463
13.7.8	Noch einmal alles zusammen	464
13.8	Beispiele zu Schleifen und Bedingungen	466
13.8.1	Blocktypen ausprobieren	466
13.8.2	Spielfigur automatisch durch die Welt bewegen	468
13.8.3	Eine Treppe bauen	469
13.8.4	Eine Pyramide bauen	470
13.9	Beispiele zu Funktionen	472
13.9.1	Swimmingpools bauen	472
13.9.2	Moderne Kunst?	474
13.10	Beispiele zu Klassen	477
13.10.1	Blöcke regnen lassen	477
13.10.2	Blinklichter	480
13.11	Beispiele zu Containern	483
13.11.1	Lichterkette	483
13.11.2	Mengenoperationen	485
13.12	Ein Beispiel zu Modulen und Paketen	487
13.13	exit() - wie geht es weiter?	490
Index	491

Vorwort

Wer sich in die Welt der Programmierung wagt, steht meist vor der Frage, welche Programmiersprache am besten für den Einstieg geeignet ist. Die Antwort darauf hängt naturgemäß von verschiedenen Aspekten ab: Was möchte man entwickeln, welche Hardware und welche Betriebssysteme spielen eine Rolle und wie leicht ist die gewählte Sprache zu lernen?

Diese Fragen stellen sich jedoch nicht nur dann, wenn man sich zum ersten Mal mit der Programmierung beschäftigt und einen Einstieg sucht. Als Entwickler sieht man sich häufig mit neuen Situationen konfrontiert und muss abwägen, ob nicht eine andere Programmiersprache als die bisher verwendete in der aktuellen Situation besser geeignet ist. Eine Programmiersprache ist wie ein Werkzeug: Man sucht sich für jede Aufgabe das passende heraus. Doch es gibt auch einen weiteren, viel einfacheren Grund, eine neue Programmiersprache zu lernen: die reine Neugierde, die den meisten Entwicklern eigen ist.

Python erfreut sich nicht ohne Grund großer und stets steigender Beliebtheit. Schließlich handelt es sich um eine Sprache, die sich leicht erlernen lässt und mit der gleichzeitig enorm viel möglich ist. Ob Du nun bereits eine Programmiersprache beherrschst, oder ob Du Dich zum Einstieg für Python entschieden hast: Dieses Buch ist so konzipiert, dass Du es in beiden Fällen nutzen kannst.

Beim Schreiben dieses Buches war es mir besonders wichtig, eine lockere und gemütliche Atmosphäre zu erzeugen. Wenn Du beim Lesen den Eindruck hast, dass man bei einer Tasse Kaffee zusammensitzt und gemeinsam neue Themen entdeckt, dann ist mir das hoffentlich auch gelungen.

Danksagung

Der erste und größte Dank gilt an dieser Stelle Naomi, die immer an meiner Seite ist und jederzeit Verständnis hatte, wenn ich mich zum Schreiben ins stille Kämmerlein zurückzog. Ohne Deine Unterstützung wäre dieses Buch niemals zustande gekommen!

Besonderer Dank gilt allen meinen Freunden und meiner Familie für das nicht selbstverständliche Verständnis in hektischen Zeiten.

Bettina Zankl möchte ich für das Durcharbeiten der einzelnen Kapitel und für die vielen konstruktiven und wertvollen Vorschläge danken. Der nächste Behelfsdöner kann kommen!

Sylvia Hasselbach, Irene Weilhart, und Kristin Rothe vom Carl Hanser Verlag möchte ich für die tolle Zusammenarbeit danken. Ich wette, irgendwo im Verlagsgebäude hängt nun ein Schild mit dem Aufdruck „Herr Kalista hat mal wieder eine Frage ...“.

Vielen Dank an Walter Saumweber für das gewissenhafte Korrektorat und das tolle Feedback!

Am Rande Hessens, im Juli 2018

Heiko Kalista

1

Grundlagen

■ 1.1 Hältst Du das richtige Buch in den Händen?

Du möchtest die Programmiersprache Python erlernen oder Dein bereits vorhandenes Wissen darin erweitern? Dann bist Du hier genau richtig! Egal ob Du Dich zum ersten Mal mit dem Thema Programmierung auseinandersetzt oder schon Erfahrungen mit anderen Sprachen hast: Dieses Buch wird Dir dabei helfen, Python effektiv zu nutzen. Bereits im ersten Kapitel wirst Du erste Erfolge erzielen und Ergebnisse sehen, ohne Dich zuvor durch einen Theorie-Dschungel kämpfen zu müssen. Die einzigen Voraussetzungen, die Du mitbringen musst, sind Grundkenntnisse im Umgang mit Computern, Experimentierfreudigkeit, Neugier und natürlich auch Geduld. In diesem Buch wird zwar Wert auf einen praxisnahen Einstieg gelegt, aber das bedeutet nicht, dass Du innerhalb weniger Tage besonders ausgeklügelte Anwendungen entwickeln wirst. Das Lernen einer Programmiersprache ist ein stetiger Prozess, der, wie so vieles andere auch, aus Hochs und Tiefs besteht. Ich möchte Dir daher nicht das Versprechen geben, dass Du in x Tagen das Programmieren erlernst. Dafür versichere ich Dir, dass sämtliche Themen mit der nötigen Ausführlichkeit besprochen werden und es immer Tipps und Hilfestellungen geben wird.

Es spielt keine Rolle, welches Betriebssystem Du bevorzugst. Du kannst mit diesem Buch sowohl unter Windows, macOS als auch Linux arbeiten. Alle nötigen Entwicklungswerkzeuge gibt es frei im Netz und werden gleich näher vorgestellt.

Kenntnisse in anderen Programmiersprachen sind zwar vorteilhaft, aber definitiv keine Grundvoraussetzung. Und falls es Dir doch einmal nicht schnell genug geht, kannst Du auch die Zusammenfassung am Ende jedes Kapitels lesen und dann immer noch entscheiden, ob Du das Kapitel Schritt für Schritt durcharbeiten möchtest.

■ 1.2 Dieses Buch bricht mit einigen Konventionen!

Ich habe mich dazu entschlossen, mit einigen Konventionen zu brechen. So verzichte ich beispielsweise darauf, gleich zu Beginn auf Themen wie Objektorientierung oder Container einzugehen. Vielmehr lege ich Wert darauf, dass möglichst schnell erste Ergebnisse und Erfolgserlebnisse zu sehen sind. Das bedeutet natürlich nicht, dass auf sauberen Programmierstil oder fortgeschrittene Konzepte verzichtet wird. Allerdings sollten zuerst einige Grundlagen geschaffen werden, damit man weiterführende Themen versteht. Ich werde zugunsten der Lesbarkeit und der Lernkurve auch darauf verzichten, ein neues Thema sofort in all seinen tiefsten Details zu durchleuchten. Möchte jemand beispielsweise das Kochen erlernen, dann fängt er in der Regel mit einem einfachen Rezept an. Wenn nach fünfzehn oder zwanzig Minuten das erste unkomplizierte Essen auf dem Tisch steht, ist das eine tolle Motivation. Die physikalischen und chemischen Prozesse, die während des Kochvorgangs ablaufen, interessieren zunächst noch nicht. Später wird man vielleicht ausgefeilte Rezepte ausprobieren und sich dann ganz von selbst für die Details und Hintergründe interessieren. Hierzu ein kleiner Vergleich anhand eines Rezeptes, das auf unterschiedliche Weise präsentiert wird:

- **Variante 1:** Nimm einen schönen großen Topf, fülle ihn mit einem Liter Wasser und streue einen Teelöffel Salz hinein. Stelle den Topf auf den Herd, schalte die Platte auf Stufe 5 und warte, bis das Wasser kocht. Gib nun eine halbe Packung Nudeln hinzu, rühre gelegentlich um und warte, bis diese bissfest sind. Guten Appetit!
- **Variante 2:** Besorgen Sie sich einen temperaturbeständigen, oben offenen Behälter (sogeannter „Topf“), der gute Wärmeleiteigenschaften aufweist. Ideal ist das Element Aluminium oder auch eine Legierung, wie etwa Messing. Messen Sie anschließend mit einem geeigneten und möglichst gut geeichten Messbecher exakt einen Liter Dihydrogenmonoxid (ugs. „Wasser“) ab und füllen Sie dieses in den Behälter. Fügen Sie als Nächstes fünf Gramm Natriumchlorid (besser bekannt als „Salz“) hinzu. Dies dient ausschließlich geschmacklichen Zwecken und nicht etwa dem weit verbreiteten Irrtum, dass sich der Siedepunkt durch die Beigabe in einem relevanten Ausmaß ändert. Beachten Sie dabei jedoch, dass der Siedepunkt vom Luftdruck abhängig ist. Sollten Sie sich also in extremen Höhen oder Tiefen befinden, können Sie den Siedepunkt nach folgender Formel berechnen ...

Auch ohne das Rezept in Variante zwei zu Ende zu führen sollte klar sein, worauf ich hinaus will: Details, Hintergründe und Exaktheit sind durchaus wichtig, aber in dieser Fülle für ein einfaches Beispiel nicht angebracht. Sie können einen erschlagen, die Motivation rauben und vom eigentlichen Geschehen ablenken. Das bedeutet selbstverständlich nicht, dass Themen grundsätzlich oberflächlich behandelt werden. Es geht vielmehr darum, Details auf einen geeigneteren Zeitpunkt zu verschieben.

■ 1.3 Die Arbeit mit diesem Buch

Idealerweise arbeitest Du die Kapitel in der angegebenen Reihenfolge durch, denn sie bauen in der Regel auf vorangegangene Kenntnisse auf. Neue Themen werden zunächst erklärt und deren Anwendungsmöglichkeiten anhand von Beispielen verdeutlicht. Damit sich neu Gelerntes festigt, gibt es immer wieder kleine Übungsaufgaben. Sieh diese nicht als lästige Pflicht, sondern als Herausforderung an. Man lernt viel besser, wenn man sich Lösungen selbst erarbeitet, anstatt nur vorgefertigte Beispiele zu laden und zu starten. Natürlich werde ich Dir zu jeder Aufgabe Tipps und Hinweise geben. Im Anschluss gibt es dann eine Musterlösung, die Du mit Deiner eigenen Lösung vergleichen kannst.

Eine besondere Art von Aufgaben sind die sogenannten Fehlerquelltexte. Dabei handelt es sich um Beispiele, in die absichtlich Fehler eingebaut wurden. Diese Fehler führen dazu, dass sich das Programm entweder nicht ausführen lässt, oder dass es nicht so funktioniert wie erwartet. Selbst der erfahrenste Programmierer macht immer wieder Fehler und muss in der Lage sein, diese zu finden und zu korrigieren. Dennoch gibt es ganz bestimmte Stolpersteine, an denen man gerade anfangs immer wieder hängen bleibt. Die Fehlerquelltexte helfen dabei, ein Gespür für typische Fallen zu entwickeln und diese in Zukunft zu umgehen. Dies ist nützlich, um Fehler nicht nur schneller zu finden, sondern sie nach Möglichkeit auch von Anfang an so gut es geht zu vermeiden. Natürlich gibt es auch an dieser Stelle immer einige Tipps, wie man den Fehler eingrenzen, finden und beheben kann. Zudem wird auch eine korrigierte Fassung des Fehlerquelltextes gezeigt.

Abgesehen von den Vorschlägen wie Du am besten mit diesem Buch arbeitest, möchte ich Dir noch einige allgemeine Tipps mit auf den Weg geben. Wie so oft ist es bei neuen Dingen so, dass man anfangs große Fortschritte macht und dann plötzlich an eine Stelle gerät, an der es scheinbar nicht oder nur langsam vorangeht. Neue Themen wirken dann vielleicht übermäßig komplex oder deren Sinn erschließt sich nicht. Das kann manchmal ziemlich frustrierend sein, ist aber völlig normal. Am besten macht man in einer solchen Situation einfach eine längere Pause und beschäftigt sich mit etwas völlig anderem. Idealerweise mit etwas, das nichts mit Computern zu tun hat. Meistens kehrt man dann später mit einem anderen Blickwinkel zu dem Problem zurück und die Dinge scheinen wieder klarer. Auch wenn das vielleicht nach einem recht offensichtlichen Tipp klingt, solltest Du ihn dennoch beherzigen. Programmierung ist eine sehr fordernde Tätigkeit, die Pausen erfordert. „Eine Nacht darüber schlafen“ hat schon so manchem hartnäckigen Programmierfehler den Garaus gemacht und für frische Ideen gesorgt. Manchmal ist es auch hilfreich, sich einem anderen Thema zu widmen, wenn man an einer Stelle nicht weiterkommt. Blättere einfach einmal in vorherige Kapitel zurück oder schnuppere in neue Kapitel hinein.

Das Erlernen einer Programmiersprache (und das Programmieren an sich) kann man mit dem Erlernen eines Musikinstruments vergleichen: Man ist niemals fertig damit und hat nie ausgelernt. Selbst wenn man alle Features einer Programmiersprache kennt, stößt man immer wieder auf neue Programmiertechniken oder Themengebiete. Daher ist es wichtig, immer am Ball zu bleiben, um nicht den Anschluss zu verlieren.

Du solltest auch nicht vergessen, dass die Zeiten vorbei sind, in denen ein einsamer Programmierer im schwach beleuchteten Keller sitzt und völlig auf sich gestellt ein Programm entwickelt. Natürlich ist das immer noch möglich, aber nicht sonderlich sinnvoll. Program-

mierer sind längst keine winzige Randgruppe mehr, die ihr Wissen völlig neu erarbeitet und dann verschwörerisch hütet. In der heutigen Zeit beschäftigen sich sehr viele Menschen mit diesem Thema und sie tauschen sich über das Internet aus. Anstatt tagelang über einem Problem zu brüten, sucht man im Internet nach einer Lösung. Man kann fast immer davon ausgehen, dass jemand anderes zuvor genau das gleiche Problem hatte und es eine Lösung dafür gibt. Falls nicht, kann man immer noch selbst um Hilfe bitten. Das bedeutet natürlich nicht, dass man nicht selbst nachdenken sollte und sich alles nur noch zusammensuchen muss. Den eigenen Kopf einzuschalten ist die sinnvollste Lösung. Dennoch ist niemand mehr gezwungen, ein Problem auf sich alleine gestellt zu analysieren, ohne dass er Hilfe erhalten würde. Solltest Du auf offene Fragen anderer Programmierer stoßen, dann scheue Dich nicht davor, Deine Hilfe anzubieten. Du wirst staunen, wie schnell Du in der Lage sein wirst, Dein Wissen selbst wieder weiterzuvermitteln. Das ist natürlich auch nicht ganz uneigennützig, denn Du wirst selbst eine Menge lernen, wenn Du anderen hilfst. Wie viel Austausch Du betreiben möchtest, bleibt Dir natürlich selbst überlassen. Falls Du Dich dennoch dazu entschließt, im stillen Kämmerlein zu arbeiten, wirst Du eine Menge verpassen.

■ 1.4 Das Kapitel zu Minecraft

Wie Dir sicher nicht entgangen ist, beschäftigt sich das letzte Kapitel in diesem Buch mit dem bekannten Spiel Minecraft. Wenn Du einen Raspberry Pi besitzt, kannst Du eine abgespeckte Version von Minecraft kostenfrei spielen und diese sogar mit Python programmieren. Das ist zwar kein Muss und nur optional, stellt aber eine tolle Möglichkeit dar, neu Gelerntes besser zu visualisieren. Statt Ergebnisse im Konsolenfenster auszugeben, kannst Du stattdessen eben Blöcke in einer Spielwelt platzieren. Es geht nicht darum, die kreativsten oder aufwendigsten Dinge zu bauen, sondern einfach Beispiele „zum Anfassen“ anzubieten. Du wirst an einigen Stellen im Buch Verweise auf passende Beispiele im Minecraft-Kapitel finden, mit denen Du experimentieren kannst.

■ 1.5 Das Begleitmaterial zum Buch

Welche Entwicklungswerkzeuge Du für das Schreiben von Python-Programmen verwendest, kannst Du natürlich frei entscheiden. In diesem Buch wird die IDE (*Integrated Development Environment*, auf Deutsch *Integrierte Entwicklungsumgebung*) *PyCharm* von JetBrains verwendet. Dabei handelt es sich um ein mächtiges Werkzeug, das es in einer kostenfreien und einer kostenpflichtigen Version gibt. Die kostenfreie Version ist jedoch absolut ausreichend für alles, was in diesem Buch benötigt wird. Mehr zu PyCharm erfährst Du in Abschnitt 1.11.

Sämtliche im Buch abgedruckten Beispiele kannst Du unter www.hanser-fachbuch.de herunterladen. Gib dort oben rechts im Suchfeld **Python 3** ein, klicke auf das Lupen-Icon und anschließend auf den Buchtitel. Im Reiter **Extras** findest Du daraufhin den Link zu den Beispielen.

■ 1.6 Anregungen? Kritik? Forum?

Das Schreiben dieses Buches hat mir viel Freude bereitet und ich hoffe, dass Du ebenso viel Spaß daran haben wirst, es zu lesen und damit zu arbeiten. Falls Du Anregungen, Verbesserungsvorschläge oder Kritik hast, dann schreibe mir eine E-Mail an h.kalista@icloud.com.

Ich habe während des gesamten Entstehungsprozesses dieses Buches darüber nachgedacht, ob ich eine Webseite mit einem Forum bereitstellen soll, in dem generelle Fragen zu Python gestellt werden können. Der Grund, warum ich darauf verzichtet habe, ist recht einfach: Es gibt bereits sehr viele Diskussionsforen und Hilfeseiten, auf denen sich eine große Nutzer-schaft mit unterschiedlicher Erfahrung betätigt. Ein weiteres Forum ins Leben zu rufen, das zunächst mit Inhalt befüllt und eine Community generieren muss, ist da sicher nicht die effektivste Idee. Zudem gibt es in den Weiten des Internets wohl kaum eine Frage zu Python, die noch nicht gestellt und beantwortet wurde. Von daher möchte ich Dich dazu ermutigen, einfach einmal die vielen verschiedenen deutsch- und englischsprachigen Foren zu durchstöbern. Du wirst sehen, dass dort bereits eine große Community existiert.

Sollte es Neuigkeiten, Korrekturen oder Ergänzungen zum Buch geben, findest Du sie auf www.hanser-fachbuch.de. Gib im Suchfeld oben rechts **Python 3** ein, um zur Buchseite zu gelangen.

■ 1.7 Die Geschichte von Python in 120 Wörtern

Ich weiß, dass Du sofort loslegen möchtest, daher mache ich es kurz: Python ist eine recht alte, aber immer noch aktuelle Sprache, die ständig weiterentwickelt wird. Die erste Version wurde zu Beginn der 1990er Jahre von Guido van Rossum entwickelt und erschien im Januar 1994 unter der Versionsnummer 1.0. Nach vielen kleineren Änderungen in den Versionen 1.1 und 1.2 wurde im Oktober 2000 die Version 2.0 veröffentlicht. Die wichtigste Neuerung war die sogenannte Garbage Collection (dabei handelt es sich vereinfacht gesagt um das automatische Aufräumen des Speichers). Ende 2008 wurde Version 3.0 veröffentlicht. Neben neuen Features wurden bestehende Sprachelemente grundlegend verändert und modernisiert, womit allerdings auch die Kompatibilität zu früheren Versionen verloren ging. Bis heute wird auch die Version 2.7 noch mit Updates versorgt und existiert parallel zur aktuellsten Version 3.7.

■ 1.8 Was kann man mit Python machen (und was nicht)?

Häufig stellen sich gerade Programmierneinsteiger die Frage, welche die beste Programmiersprache sei. Das lässt sich genauso wenig beantworten wie die Frage, ob ein Hammer besser ist als ein Schraubenschlüssel. Beides sind Werkzeuge, nur eben für verschiedene Zwecke. Bei Programmiersprachen ist das ähnlich.

Python ist eine sehr vielseitige und flexible Sprache, die sich für fast alle Anwendungsgebiete eignet. Nicht ohne Grund nimmt die Verbreitung der Sprache gegenwärtig immer weiter zu. Zu den großen Stärken von Python zählen die leichte Erlernbarkeit, die gute Strukturierung der Sprache sowie deren Zugänglichkeit. Mit Python erhält man sehr schnell Ergebnisse, denn die Entwicklungszeiten sind gegenüber anderen Programmiersprachen sehr kurz. Nicht umsonst wird Python in der heutigen Zeit sehr oft als Einsteigersprache empfohlen. Das bedeutet jedoch nicht, dass sie nur für kleine Aufgaben geeignet ist. Ganz im Gegenteil: Sogar große und namhafte Unternehmen wie Google, die NASA, Dropbox und Facebook setzen auf die Vorteile von Python.

Möchtest Du ein kleines Tool schreiben, das Dir bei der täglichen Arbeit am Rechner einige immer wiederkehrende Aufgaben abnimmt? Dann verwende Python! Hast Du vor, ein Programm mit grafischer Benutzeroberfläche zu entwickeln, ohne dafür übermäßig viel Aufwand betreiben zu müssen? Das ist mit Python möglich! Wolltest Du schon immer einmal deine eigene Heimautomatisierung realisieren und die Beleuchtung Deiner Zimmer, die Temperatur im Wohnzimmer oder die Rollos am Fenster per Knopfdruck steuern? Mit einem Raspberry Pi und Python kein Problem!

Python ist im wissenschaftlichen Bereich ebenso vertreten wie in der Anwendungsentwicklung und im Webbereich. Die Einsatzzwecke erstrecken sich dabei von kleinen, schnell zu erledigenden Aufgaben für zwischendurch bis hin zu großen, aufwendigen Projekten. Ein echter Allrounder eben.

Ein weiterer Vorteil von Python ist die Plattformunabhängigkeit. Es spielt keine Rolle, ob man für Windows, Linux oder macOS entwickeln möchte. Python ist für alle relevanten Systeme vorhanden und kostenfrei erhältlich. Auf vielen Unix-basierten Systemen ist Python sogar bereits vorinstalliert (Linux, macOS).

Allerdings gibt es auch einige Nachteile, denn es kann natürlich nicht „die“ perfekte Sprache für alles geben. Python eignet sich beispielsweise nicht für tiefgehende Systemprogrammierung. Niemand würde einen Treiber oder ein Betriebssystem in Python schreiben. Das liegt unter anderem daran, dass es sich um eine interpretierte Sprache handelt. Was genau das bedeutet, erfährst Du im nächsten Abschnitt. Zusammenfassend kann man jedoch sagen, dass ein Python-Programm nicht alleine lauffähig ist, sondern immer ein separates Programm zur Ausführung benötigt, den sogenannten Interpreter. Das bedeutet natürlich, dass man mit Python-Programmen in der Regel keine Wettrennen gewinnen wird. Wenn Ausführungsgeschwindigkeit der wichtigste Aspekt ist, ist es also nicht die erste Wahl. Zwar können rechenintensive Programmteile auch in anderen Sprachen entwickelt und dann eingebunden werden, aber „pure“ Python-Programme gehören nicht zu den Geschwindigkeitswundern. Es muss also je nach Anforderung abgewogen werden, was

wichtiger ist: schnelle Entwicklungszeit oder schnelle Ausführung. Eine eigene Grafik-Engine für moderne Computerspiele wird man mit Python also wohl nicht entwickeln.

Möchte man Apps für mobile Systeme wie Smartphones und Tablets entwickeln, so wird man zwangsläufig bei Objective-C oder Swift (iOS), respektive Java (Android) landen. Dies ist also einer der wenigen Bereiche, in denen man mit Python nicht weiterkommt.

Ein weiterer Anwendungsfall, bei dem Python den Kürzeren zieht, ist die Entwicklung im Embedded-Bereich. In diesem Gebiet ist jedes Byte wichtig und es wird an jeder Ecke optimiert. Einen Mikrocontroller wird man auch in den nächsten Jahren noch in C, C++ oder gar Assembler programmieren. Dennoch bleibt einem dieses Gebiet nicht gänzlich verschlossen, denn mit einem Raspberry Pi kann auch eine ganze Menge im Elektronikbereich realisiert werden.

■ 1.9 Interpreter vs. Compiler

Bevor es gleich mit dem ersten Python-Programm losgeht, möchte ich noch auf einige Grundlagen von Programmiersprachen im Allgemeinen eingehen. Grundsätzlich ist es so, dass ein Computer zunächst weder mit einem in C, C++, oder Python verfassten Quelltext etwas anfangen kann. Wie bekannt sein dürfte, arbeitet ein Computer auf seiner elementarsten Ebene nur mit Nullen und Einsen – Strom an, Strom aus. Ein alter Hut, aber wichtig. Denn schließlich hat niemand von uns Lust, Programme in Nullen und Einsen in einen Computer zu hacken. Die nächsthöhere Ebene stellt die sogenannte Maschinen- oder Assemblersprache dar. Vereinfacht kann man sich das als einen kompakten Befehlssatz vorstellen, den ein Prozessor verstehen und verarbeiten kann. Dieser Befehlssatz besteht hauptsächlich aus simplen Rechen-, Vergleichs- und Sprunganweisungen. Im Vergleich zu Nullen und Einsen ist das zwar schon wesentlich bequemer, aber immer noch alles andere als lesbar. Du kannst Dich gerne davon überzeugen, indem Du Dir das folgende Beispiel anschaust:

```
movl  %r9d, %eax
movq  $0x0, -0xd0(%rbp)
movq  $0x0, -0x108(%rbp)
movq  -0x208(%rbp), %rcx
addq  $0x18, %rcx
movq  %rcx, %rdi
movq  %rax, %rdx
movq  %rax, %rcx
callq 0x10046f3f0
leaq  -0x120(%rbp), %rdi
movq  -0x208(%rbp), %rax
movq  0x18(%rax), %rcx
movq  %rcx, -0x220(%rbp)
```

Nicht gerade vielsagend, oder? Dennoch gibt es auch heute noch gewisse Situationen, in denen Maschinensprache weiterhin verwendet wird (direkter Zugriff auf die Hardware, zusätzliche Optimierung und so weiter). Und das wird sich so schnell auch nicht ändern. Zum Glück gibt es aber die sogenannten Hochsprachen, zu denen beispielsweise C, C++, C#, Basic und auch Python zählen. Diese Hochsprachen zeichnen sich durch ihre wesent-

lich bessere Lesbarkeit und ihren größeren Funktionsumfang aus. Wie Du Dir jetzt denken kannst, muss ein in einer solchen Hochsprache geschriebenes Programm zuerst in Maschinensprache umgewandelt werden, damit der Computer etwas damit anfangen kann. Wie dies geschieht, hängt von der verwendeten Programmiersprache ab. Grundsätzlich unterscheidet man zwischen kompilierten und interpretierten Sprachen. Bei kompilierten Sprachen wie etwa C oder C++ übersetzt ein sogenannter Compiler den Quellcode (also das in dieser Sprache geschriebene Programm) in Maschinencode und erzeugt somit ein ausführbares Programm, das von nun an von alleine lauffähig ist. Soll das Programm auf unterschiedlichen Plattformen (Windows, Linux, macOS) laufen, so muss es für die jeweiligen Plattformen separat kompiliert werden.

Bei interpretierten Sprachen wie beispielsweise Python läuft es etwas anders ab. Dreh- und Angelpunkt ist der sogenannte Python-Interpreter. Dieser verwandelt ein Python-Programm im ersten Schritt in sogenannten Bytecode. Dieser Bytecode ist für sich alleine noch nicht lauffähig, dafür ist dann die PVM (*python virtual machine*) zuständig. Wird das Python-Programm gestartet, führt diese virtuelle Maschine nacheinander alle Anweisungen aus, die im Bytecode stecken. Das Ganze ist unter der Haube ein recht komplexer Vorgang, aber glücklicherweise laufen diese Dinge automatisch im Hintergrund ab und wir müssen uns nicht darum kümmern. Dennoch halte ich es für wichtig, dass man mit den grundlegenden Konzepten vertraut ist.

Es sollte klar sein, warum eine interpretierte Sprache in der Regel langsamer ist als eine kompilierte und warum ein Anwender einen Python-Interpreter installiert haben muss, wenn er ein Python-Programm ausführen möchte.

■ 1.10 Python 2.7 oder 3.7?

Wer sich mit Python beschäftigt, wird früher oder später mit der Frage konfrontiert, ob Python 2.7 oder Python 3.7 verwendet werden soll. Dies mag verwundern, da man eigentlich meinen sollte, dass immer die neueste Version vorzuziehen sei. Dass es nicht zwangsläufig so sein muss, zeigen die vielen Diskussionen zu diesem Thema im Internet. Die Verwirrung um diese beiden Versionen hängt mit der Tatsache zusammen, dass mit Version 3.0 einige einschneidende Änderungen vorgenommen wurden. Diese Änderungen waren so fundamental, dass die Abwärtskompatibilität zu älteren Python-Versionen verloren ging. Die neue Version beseitigte viele Unstimmigkeiten und wurde somit konsistenter und einsteigerfreundlicher. Zudem kamen neue Features hinzu und es wurde konsequent auf Unicode gesetzt (dies wurde auch in 2.x-Versionen unterstützt, musste aber separat markiert werden). Es versteht sich von selbst, dass weder Firmen noch Privatpersonen ihren gesamten Quellcode sofort auf Python 3.0 umstellen wollten oder konnten. Von daher wurde beschlossen, zunächst die Versionen 2.7 und 3.0 parallel laufen zu lassen und die ältere Version ebenfalls mit Updates und Fehlerkorrekturen zu versorgen. Wer wollte, konnte neue Projekte mit Version 3.0 erstellen und bei alten Projekten dennoch vom Support für Version 2.7 profitieren. Es wurden sogar einige Features von neueren Python-Versionen in Version 2.7 übernommen, solange diese nicht die Kompatibilität gefährdeten. Python 2.7 sollte ursprünglich bis 2015 gepflegt werden, dieses Datum wurde jedoch bis 2020 verlän-

gert. Doch was bedeutet das nun? Soll man mit der aktuellsten Version von Python beginnen, oder doch lieber erst mit 2.7?

Zunächst einmal sollte man sich bewusst sein, dass es sich bei den beiden Versionen nicht um völlig verschiedene Sprachen handelt. Man muss also nicht von Grund auf neu lernen, sondern sich nur mit einigen Änderungen auseinandersetzen. Der Lernerfolg sollte somit also identisch sein. Dennoch möchte ich Dir empfehlen, mit Python 3.7 einzusteigen und Dich bei Bedarf mit den Unterschieden zu Version 2.7 vertraut zu machen.

Etwas genauer hinschauen musst Du allerdings, wenn Du neben dem reinen Lernen der Sprache auf ein bestimmtes Ziel hinarbeitest. In Python kann man sogenannte Libraries (Bibliotheken) einbinden, die bestimmte Funktionalitäten bieten. Dies können beispielsweise Bibliotheken für Netzwerk-, Grafik-, oder KI-Programmierung sein. Solche Bibliotheken sind ebenfalls in Python geschrieben und stammen meist aus der Open-Source-Szene. Benötigt man unbedingt eine bestimmte Bibliothek, die es allerdings nur in Version 2.7 gibt, hat man nur zwei Möglichkeiten: Man verwendet für das eigene Projekt ebenfalls Python 2.7 oder man versucht, die Bibliothek umzuschreiben.

■ 1.11 Die Entwicklungsumgebung PyCharm

Wir wissen jetzt, was man mit Python so alles anstellen kann, wie es unter der Haube funktioniert und welche Version wir benötigen. Doch womit schreibt man am besten Python-Programme? Wie so oft lässt sich diese Frage nicht pauschal beantworten. Im Grunde genügt ein einfacher Texteditor und ein installierter Python-Interpreter. Allerdings leben wir im Jahre 2017 und es spricht nichts gegen etwas Komfort. Für die meisten Programmiersprachen gibt es sowohl einfache Tools, als auch sehr leistungsfähige Entwicklungsumgebungen, die einem das Leben als Programmierer erleichtern. Bei letzteren spricht man von sogenannten IDEs. Das ist die Abkürzung für den englischen Begriff *Integrated Development Environment*, zu Deutsch *Integrierte Entwicklungsumgebung*. Eine solche IDE besteht aus mehreren Komponenten. Neben einem Texteditor mit vielen auf die jeweilige Programmiersprache zugeschnittenen Bequemlichkeitsfunktionen findet man meistens auch einen Debugger (ein Tool, mit dem man Programmfehler aufspüren kann), eine Quellcodeverwaltung und Werkzeuge für Analyse und Tests. Der Vorteil solcher IDEs besteht darin, dass man alles Nötige zentral in einem Programm gebündelt hat (wie der Name durchaus erahnen lässt).

Einige dieser Tools sind kostenlos erhältlich, andere sind nur für viel Geld zu haben oder stark in ihrer Funktion eingeschränkt. In diesem Buch wird die Community Edition von PyCharm verwendet. Diese ist für Windows, macOS und Linux kostenlos verfügbar, hat alle Features, die wir benötigen, und sie darf sogar für kommerzielle Projekte verwendet werden. PyCharm ist ein sehr mächtiges Werkzeug, das eine entsprechende Einarbeitung erfordert. Allerdings werde ich darauf verzichten, gleich zu Beginn alle Features zu erklären. Vielmehr beschränke ich mich darauf, immer nur die Teile anzusprechen, die für das aktuelle Thema auch wirklich wichtig sind. Es steht Dir natürlich frei, eine andere IDE zu benutzen.

1.11.1 Alternativen zu PyCharm

Neben dem gerade erwähnten PyCharm und der Verwendung eines schlichten Texteditors gibt es selbstverständlich noch andere Entwicklungsumgebungen, von denen ich Dir hier drei vorstellen möchte:

IDLE

Ein bekanntes, schlicht gehaltenes und im Grunde völlig ausreichendes Programm ist *IDLE*, das in der Regel zusammen mit dem Python-Interpreter installiert wird. Wenn Du nur mal eben eine kleine Aufgabe mit Python erledigen willst, dann ist *IDLE* sicherlich nicht verkehrt (es verfügt sogar über einen Debugger). Man muss jedoch fairerweise dazu sagen, dass es sich bei diesem Programm nicht unbedingt um das modernste und benutzerfreundlichste handelt. Starte es einfach einmal und probiere aus, ob es Dir liegt. Wenn Du Ubuntu 18.04 verwendest, musst Du es noch per `sudo apt-get install idle3` installieren. Unter Windows und macOS wird es zusammen mit dem Python-Interpreter ausgeliefert.

Thonny

Mehr Komfort, eine modernere Benutzeroberfläche und zusätzliche Features bietet *Thonny*, das Du unter <http://thonny.org> für Windows und macOS kostenfrei herunterladen kannst. Für die Installation unter Linux findest Du unter dem genannten Link alle nötigen Informationen. In Kapitel 13 (Minecraft) wird *Thonny* für die Python-Entwicklung verwendet, Du wirst es also noch näher kennenlernen.

Visual Studio Code

Sehr viele Features und Möglichkeiten bietet Dir *Visual Studio Code*, das Du unter <https://code.visualstudio.com> ebenfalls kostenfrei für Windows, macOS und Linux herunterladen kannst. Wenn Du Ubuntu 18.04 verwendest, findest Du es auch direkt unter **Ubuntu Software**. *Visual Studio Code* ist ein sehr mächtiges Werkzeug, das für die Entwicklung mit den unterschiedlichsten Programmiersprachen eingesetzt wird. Um es zusammen mit Python zu verwenden, sind jedoch einige vorbereitende Schritte notwendig. Diese sind übersichtlich auf der Seite <https://code.visualstudio.com/docs/python/python-tutorial> beschrieben.

■ 1.12 Python-Interpreter installieren

Python ist eine interpretierte Sprache und benötigt somit einen entsprechenden Interpreter. Diesen gibt es kostenlos für fast alle relevanten Betriebssysteme und bei vielen gehört er bereits zum Lieferumfang (beispielsweise macOS und so gut wie alle Linux-Distributionen). Im Folgenden gehe ich kurz auf die Installation unter Windows, macOS und Linux (Ubuntu 18.04) ein. Falls Du eine andere Linux-Distribution verwendest, kann sich die Vorgehensweise der Installation eventuell unterscheiden.



HINWEIS: Python 3.7 ist nicht abwärtskompatibel zu Python 2.7. Wenn Du also auch mit Python 2.7 arbeiten möchtest, dann musst Du beide Interpreter installieren.

1.12.1 Python-Interpreter unter Windows installieren

Windows-Betriebssysteme enthalten leider keinen vorinstallierten Python-Interpreter, weder den der Version 2.7 noch den der Version 3.7. Um diesen zu installieren, gehe zunächst auf die Seite <https://www.python.org/downloads> und klicke auf den Button **Download Python 3.7.0** um die Installationsdatei herunterzuladen. Zusätzlich hast Du noch die Möglichkeit, weiter unten auf der Seite verschiedene Python-Installationen herunterzuladen. Diese unterscheiden sich anhand der Version, der Architektur (32 oder 64 Bit) und der Art des Installationsprogramms. Das ist an dieser Stelle allerdings nicht von Bedeutung und Du kannst beruhigt die Standardversion verwenden.



HINWEIS: Python ab Version 3.5 benötigt mindestens Windows Vista oder neuer. Solltest Du noch Windows XP verwenden, dann musst Du auf Python 3.4 ausweichen.

Folge nun dem Installationsverlauf und wähle die Option **Add Python 3.7 to PATH**. Das sorgt dafür, dass Du Python aus der Eingabeaufforderung (Kommandozeile) von überall her aufrufen kannst, ohne vorher extra in ein Verzeichnis wechseln zu müssen.

Um zu prüfen, ob Python korrekt installiert wurde, hast Du zwei Möglichkeiten. Öffne zuerst die Eingabeaufforderung und gib wahlweise `python` oder `py` ein. Wenn alles geklappt hat, solltest Du folgende Bildschirmausgabe sehen:

```
Microsoft Windows [Version 10.0.16299.431]
(c) 2017 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Heiko>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Der Python-Interpreter wird gestartet, gibt die aktuelle Version aus und zeigt mit `>>>` an, dass er bereit ist, Python-Befehle zu empfangen. Du kannst das Fenster jetzt entweder einfach schließen, oder mit der Eingabe von `exit()` und der **Enter**-Taste den Interpreter beenden.

Die zweite Möglichkeit besteht darin, über das Startmenü den Eintrag **Python 3.7 -> Python 3.7 (32 Bit)** anzuwählen. In diesem Fall öffnet sich ebenfalls die Eingabeaufforderung. Dieses Mal allerdings direkt mit gestartetem Python-Interpreter. Weiterhin findest Du im Startmenü noch Einträge zur Dokumentation und der mitgelieferten Entwicklungsumgebung *IDLE*. Diese wurde weiter oben bereits als Alternative zu PyCharm angesprochen. Wir

verwenden sie zwar in diesem Buch nicht, aber Du kannst sie natürlich einfach einmal ausprobieren, vielleicht liegt sie Dir ja!

1.12.2 Python-Interpreter unter macOS installieren

Unter macOS ist zwar Python 2.7 installiert, nicht aber 3.7. Das lässt sich jedoch schnell nachholen. Öffne die Seite <https://www.python.org/downloads> und klicke auf die Schaltfläche **Download Python 3.7.0**. Die Installationsdatei findest Du anschließend unter dem Eintrag *macOS 64-bit Installer*. Führe diese aus und folge einfach den gezeigten Schritten. Anpassungen sind nicht nötig. Sobald die Installation abgeschlossen ist, kannst Du zur Sicherheit noch einmal überprüfen, ob alles wie erwartet funktioniert hat. Öffne dazu einfach ein Terminal-Fenster und gib dort `python3` ein. Die Ausgabe sollte wie folgt aussehen:

```
Last login: Thu May 24 15:02:32 on ttys000
Heikos-mac:~ heiko$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Die drei spitzen Pfeile (`>>>`) zeigen an, dass der Python-Interpreter gestartet wurde und bereit ist, Python-Befehle zu empfangen. Bevor Du das Terminal-Fenster schließt, gib noch `exit()` ein, um den Interpreter zu beenden.

1.12.3 Python-Interpreter unter Linux installieren

Ubuntu 18.04 bringt bereits eine Installation von Python 3.6.5 mit, die für die Arbeit mit diesem Buch völlig ausreichend ist. Falls Du die aktuellste Python-Version (Python 3.7) einsetzen möchtest, musst Du diese zuerst noch installieren. Öffne dazu einfach ein Terminal-Fenster und führe die Installation mit dem folgenden Befehl aus:

```
sudo apt-get install python3.7
```

Daraufhin werden die nötigen Pakete heruntergeladen und installiert. Du hast nun sowohl Python 3.6.5, als auch Python 3.7 parallel zur Verfügung. Jetzt solltest Du noch überprüfen, ob die Installation korrekt durchgeführt wurde. Öffne dazu ein Terminal-Fenster und gib dort `python3.7` ein. Wenn Du die folgende Ausgabe siehst, hat alles richtig funktioniert:

```
heiko@ubuntu:~$ python3.7
Python 3.7.0b3 (default, Mar 30 2018, 04:35:22)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits", or "license" for more information.
>>>
```

Du kannst das Terminal-Fenster wieder schließen, solltest aber vorher den Python-Interpreter durch die Eingabe von `exit()` beenden.

1.12.4 PyCharm unter Windows installieren

Idealerweise lädst Du das Installationspaket direkt beim Hersteller unter der Adresse <https://www.jetbrains.com/pycharm/download> herunter. Dort hast Du die Wahl zwischen der kostenlosen *Community Edition* und der kostenpflichtigen *Professional Edition*. Letztere bietet mehr Funktionsumfang, wie etwa Datenbank- und Webentwicklung. Selbstverständlich sind alle Beispiele in diesem Buch auch mit der kostenfreien Version ausführbar.

Folge einfach den Schritten des Installationsprogramms und setze bei *Create associations* den Haken bei *.py*. Das sorgt dafür, dass künftig alle Python-Quelltexte (also Dateien mit der Endung *.py*) direkt mit PyCharm geöffnet werden. Die Wahl der Softwarearchitektur (32 oder 64 Bit) hängt von Deinem System ab. Wenn Du Dir nicht sicher bist, drücke **Windows+Pause** und sieh unter dem Eintrag *Systemtyp* nach. Das Häkchen bei *Download and install JRE x86 by JetBrains* musst Du nur aktivieren, wenn Du ein 32-Bit-System benutzt. Bild 1.1 zeigt das nochmals im Detail.

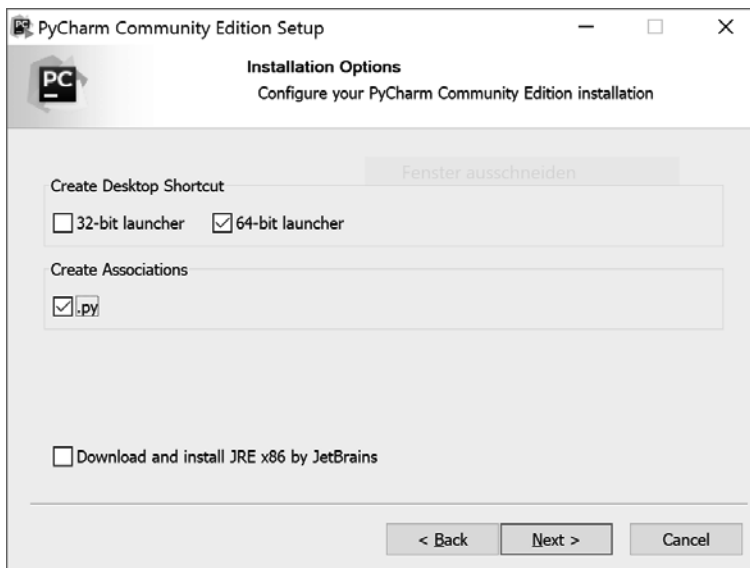


Bild 1.1 Die wichtigsten Optionen bei der Installation von PyCharm unter Windows

1.12.5 PyCharm unter macOS installieren

Zunächst benötigst Du die Installationsdatei für PyCharm, die Du unter der Adresse <https://www.jetbrains.com/pycharm/download> herunterladen kannst. Genau wie für Windows gibt es auch hier sowohl die kostenlose *Community Edition*, als auch die kostenpflichtige *Professional Edition*. Entscheide Dich für die *Community Edition*, diese ist für alle Themen dieses Buchs mehr als ausreichend.

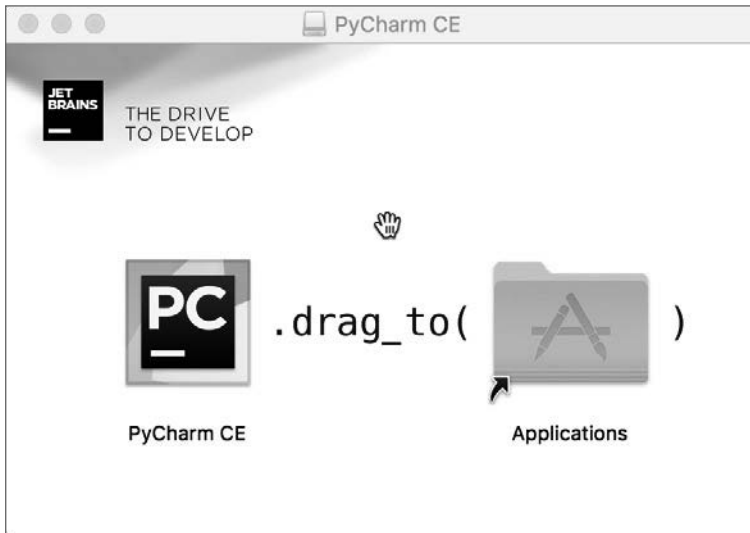


Bild 1.2 Zur Installation unter macOS musst Du PyCharm einfach nur in den Programme-Ordner ziehen.

Die Installation ist nicht weiter kompliziert. Öffne die heruntergeladene *.dmg*-Datei und ziehe in dem sich öffnenden Fenster das Icon *PyCharm CE* in den sich daneben befindlichen Ordner *Applications* (siehe Bild 1.2). Jetzt kannst Du das Image auswerfen und PyCharm öffnen.

1.12.6 PyCharm unter Linux installieren

Die Installation der Entwicklungsumgebung gestaltet sich unter Ubuntu 18.04 sehr einfach, da PyCharm in den offiziellen Paketquellen bereits vorhanden ist. Klicke einfach im Dock auf **Ubuntu-Software** oder alternativ auf **Anwendungen Anzeigen -> Ubuntu-Software**. Klicke auf die Lupe oben rechts und gib im Suchfeld *pycharm* ein. Dir werden verschiedene Versionen angezeigt, von denen Du Dich für die kostenfreie Version *PyCharm CE* entscheiden solltest. Bestätige nun mit einem Klick auf **Installieren** und starte anschließend PyCharm. Bild 1.3 zeigt diesen Schritt.

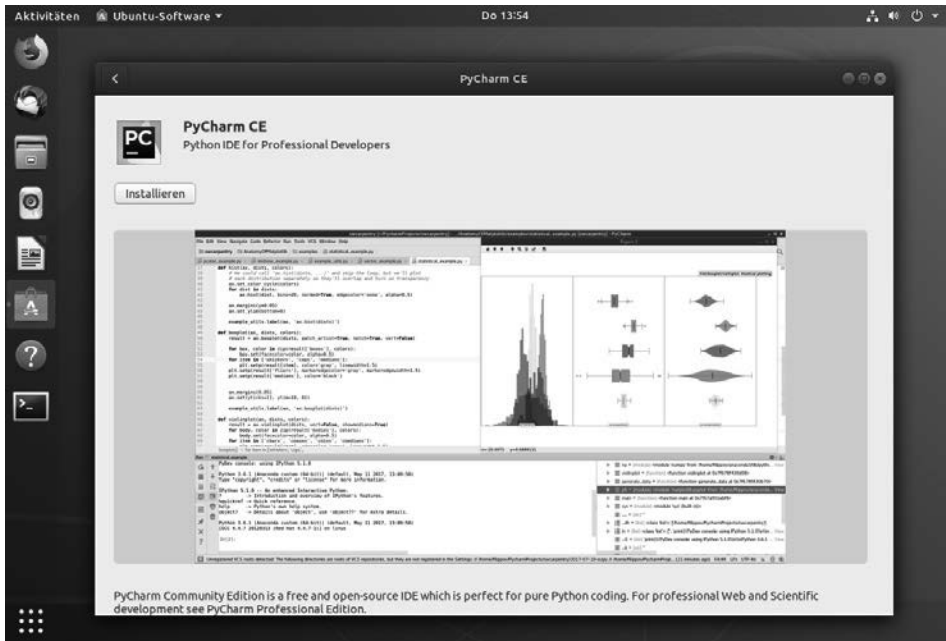


Bild 1.3 Ein Button-Klick genügt für die Installation unter Ubuntu 18.04.



HINWEIS: Falls Du eine andere Linux-Distribution verwendest, weicht die Installation möglicherweise von dem hier beschriebenen Vorgehen ab. Besuche in diesem Fall die Seite <https://www.jetbrains.com/pycharm/download/#section=linux>. Dort findest Du PyCharm als `.tar.gz`-Archiv.

1.12.7 PyCharm einrichten

Beim ersten Start von PyCharm wirst Du gefragt, ob Du bestehende Einstellungen einer alten Installation importieren möchtest. Wähle hier einfach **Do not import settings** und klicke auf **OK**. Als Nächstes kannst Du zwischen verschiedenen Tastaturlayouts (nur macOS) und Farbschemata wählen. Ich habe mich für das voreingestellte Tastaturlayout und die dunkle Variante *Darcula* entschieden, da der Kontrast für mich persönlich angenehmer ist. Du kannst die Einstellungen natürlich jederzeit wieder ändern. Probiere also ruhig beides einmal aus. Klicke zum Abschluss auf **Skip Remaining and Set Defaults**, um die weiteren Einstellungen zu überspringen. Falls Du sehen möchtest, was Du da überspringst, kannst Du auch alternativ auf **Next: Featured plugins** klicken und Dir die weiteren Optionen ansehen. Diese sind für uns allerdings nicht von Bedeutung.

Wähle auf dem Startbildschirm unten rechts die Option **Configure -> Settings** (unter macOS **Configure -> Preferences**), um in das Einstellungsmenü zu gelangen. Über das Hauptmenü gelangst Du ebenfalls zum Einstellungsmenü (Windows/Linux: **File -> Set-**

tings, macOS: PyCharm -> Preferences). Selektiere links in der Auswahl den Eintrag **Project Interpreter** und anschließend in der nun erscheinenden Auswahlbox den vorhin installierten Interpreter. Falls die Liste leer ist, klicke auf **Show All...** und anschließend auf die Schaltfläche mit dem Plus-Symbol. Wähle in der linken Liste den Eintrag **System Interpreter** und anschließend im Drop-down-Feld **python 3.7**. Das sollte dann wie in Bild 1.4 aussehen:

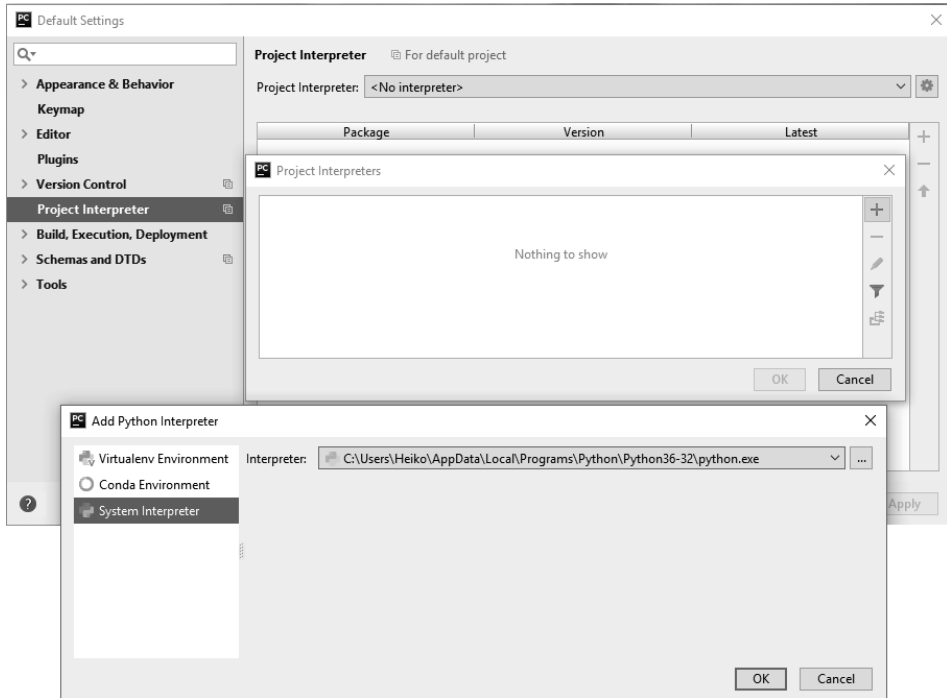


Bild 1.4 Ein wenig verschachtelt, aber schnell erledigt: Festlegen des gewünschten Interpreters.

Diese Einstellung bewirkt, dass bei jedem neuen Projekt automatisch der vorhin installierte Python-Interpreter verwendet wird. Solltest Du mehrere Interpreter installiert haben und in einem Projekt beispielsweise Python 2.7 verwenden, kannst Du das in den Projekteinstellungen ändern.

PyCharm hat standardmäßig eine Rechtschreibprüfung aktiviert. Nach meiner persönlichen Meinung ist das in einem Editor, in dem man vorwiegend Quelltexte schreibt, nicht wirklich hilfreich. Daher habe ich dieses Feature abgeschaltet. Gehe dazu über **Configure** -> **Settings** erneut in die Einstellungen und wähle in der linken Auswahl den Eintrag **Editor** -> **Inspections** und entferne dann das Häkchen rechts bei **Spelling** -> **Typo**. Aber auch hier bleibt es selbstverständlich Dir selbst überlassen, wofür Du Dich entscheidest. Bild 1.5 zeigt, wo sich diese Einstellung versteckt.

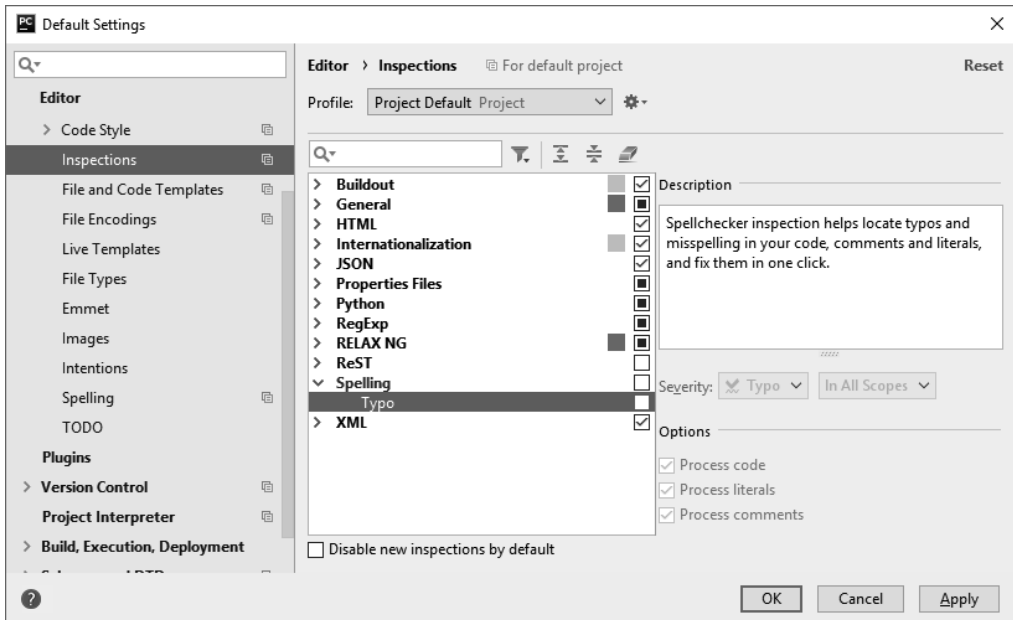


Bild 1.5 Rechtschreibprüfung im Quelltext ist nicht immer vorteilhaft.

Ebenfalls nützlich ist die Darstellung von Zeilennummern. Diese haben zwar keinen Einfluss auf den Programmablauf, helfen Dir aber dabei, den Erklärungen zu den Beispielen in diesem Buch besser zu folgen, da immer wieder auf einzelne Programmzeilen verwiesen wird. Standardmäßig ist diese Option bereits eingeschaltet. Du kannst diese Option unter **Editor -> General -> Appearance -> Show line numbers** ein- und ausschalten.

Mit diesen Einstellungen ist PyCharm soweit vorbereitet, dass wir gleich mit unserem ersten Projekt loslegen können. An späteren Stellen im Buch zeige ich Dir noch einige nützliche Funktionen und Tricks, mit denen Du mehr aus PyCharm herausholen und Dir das Entwicklerleben deutlich erleichtern kannst.

■ 1.13 Genug geredet, los gehts!

Du hast jetzt den obligatorischen, aber hoffentlich kurz genug gehaltenen, theoretischen Teil hinter Dir, also kann es mit dem ersten Beispiel losgehen. Dieses wird zwar nicht wahn-sinnig aufregend ausfallen, ist aber ein vollständiges Python-Programm, mit dem Du experimentieren und das Du erweitern kannst. Im ersten Schritt zeige ich Dir, wie Du mit PyCharm ein neues Projekt anlegst, den Quelltext eingibst und das Programm ausführst. Danach wird der Quelltext erklärt, der fürs Erste natürlich sehr kurz und kompakt sein wird.

1.13.1 Das erste Programm eingeben

Selbstverständlich stehen alle in diesem Buch gezeigten Beispiele als Download verfügbar und müssen nicht einzeln abgetippt werden. Dennoch möchte ich Dir an dieser Stelle empfehlen, zumindest das erste Beispiel selbst einzugeben. Auf diese Weise lernst Du Deine Entwicklungsumgebung von Anfang an kennen und wirst später weniger Schwierigkeiten haben, eigene Projekte anzulegen. Zudem gehst Du auf diese Weise direkt mit dem Quellcode „auf Tuchfühlung“ und bist ein wenig näher am Geschehen.

PyCharm arbeitet mit sogenannten Projekten. Ein Projekt kann aus mehreren Quellcode-Dateien bestehen, die sowohl miteinander in Verbindung stehen dürfen, als auch einzeln ausgeführt werden können, wenn sie selbstständig ausführbaren Quellcode enthalten. Was hier vielleicht kompliziert klingt, ist eigentlich recht einfach. Ein sehr großes Projekt wird in der Regel nicht aus einer einzelnen Datei bestehen, sondern seinen Quellcode thematisch gruppiert in verschiedene Dateien auslagern. Beispielsweise steht der Quellcode für die Benutzeroberfläche Deines Programms in einer Datei, die Logik und Datenverarbeitung in einer anderen. In einem solchen Fall stellt eine dieser Dateien den Startpunkt Deines Programms dar. Die anderen Dateien sind in dann in der Regel nicht alleine lauffähig. Es spricht dennoch nichts dagegen, mehrere Dateien in einem Projekt zu haben, von denen jede für sich ausführbar ist. Die Beispiele in diesem Buch machen von dieser Möglichkeit Gebrauch, damit Du nicht jedes Mal ein neues Projekt öffnen musst. Wie Du die Projektdatei mit allen Beispielen laden und diese einzeln ausführen kannst, sehen wir gleich. Sehen wir uns zunächst an, wie Du eigene Projekte anlegen, mit Leben befüllen und ausführen kannst.

Klicke zuerst auf dem Startbildschirm von PyCharm auf **Create New Project** und wähle im daraufhin erscheinenden Dialog im Feld **Location** einen Ort, an dem Du Deine Projekte speichern möchtest. Ich empfehle Dir, einen Projekte-Ordner anzulegen, in dem zukünftig alle Deine Projekte gespeichert werden. Als Namen für das Projekt kannst Du beispielsweise *Erstes Beispiel* wählen. Achte auch darauf, dass in der Auswahlbox **Interpreter** der vorhin installierte Python-Interpreter selektiert ist. Bild 1.6 zeigt, was gemeint ist.



ACHTUNG: Standardmäßig sind die Einstellungen zu **Project Interpreter** zugeklappt und somit leicht zu übersehen. Stelle sicher, dass die Option **Existing interpreter** aktiviert und **python3.7** ausgewählt ist.

Nach einem Klick auf **Create** legt PyCharm für Dich den gewählten Ordner an und erzeugt darin einen Unterordner namens *.idea*, der unter macOS und Linux unsichtbar ist. In diesem Ordner befinden sich *.xml*-Dateien, die unter anderem die Projekteinstellungen enthalten. In der Regel muss man sich nicht weiter um diese Dateien kümmern, dennoch wollte ich Dir ihre Existenz nicht verschweigen.

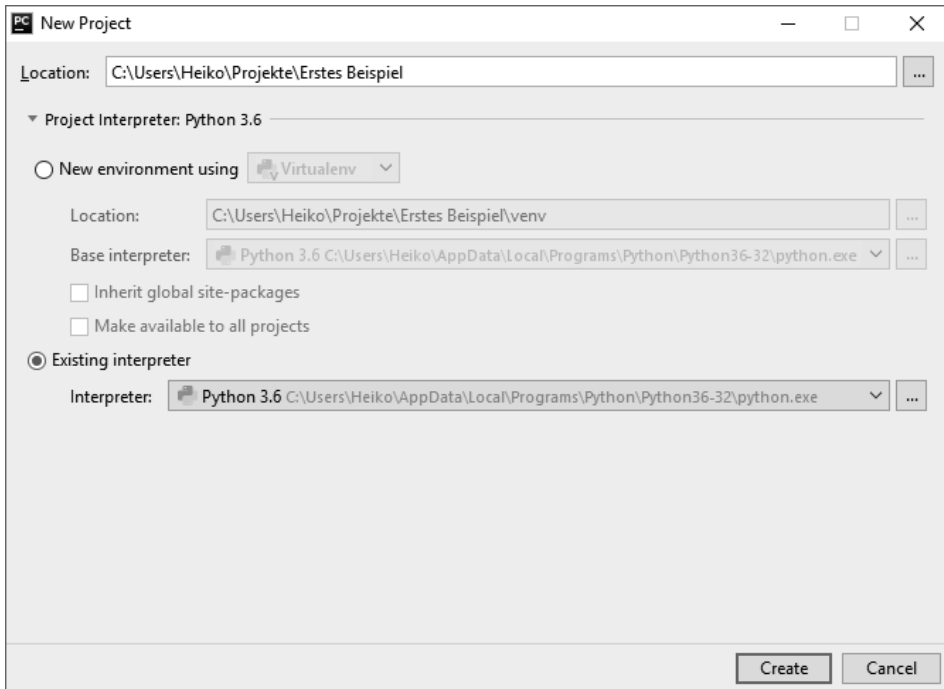


Bild 1.6 Anlegen eines neuen Projekts

So, das Projekt haben wir schon mal. Doch damit alleine kommen wir nicht sonderlich weit, denn wir brauchen noch eine Datei, die den Quelltext des ersten Beispiels enthalten wird. Klicke dazu in der Navigationsansicht auf der linken Seite mit der rechten Maustaste auf den Ordner **Erstes Beispiel** und wähle im daraufhin erscheinenden Menü den Punkt **New** -> **Python File** und suche Dir einen Namen wie beispielsweise *programm* aus. Python-Dateien haben die Endung *.py*, doch darum musst Du Dich nicht kümmern, diese wird automatisch angehängt. Wenn die Datei nicht automatisch selektiert sein sollte, dann hole das einfach mit einem Doppelklick nach. Jetzt ist alles bereit, um endlich den ersten Quelltext einzugeben. Tippe dazu einfach das Beispiel 1.1 ab und achte dabei unbedingt auf die Groß-/Kleinschreibung. Die Zeilennummern samt Doppelpunkt darfst Du nicht mit eingeben, sie gehören nicht zum Programm und dienen nur als Orientierungshilfe in der gleich folgenden Erklärung.

Beispiel 1.1 Ein- und Ausgabe im Konsolenfenster

```
01: # Beispiel 1.1
02: #
03: # Ein- und Ausgabe im Konsolenfenster
04: #
05: print("Hallo Welt!")
06:
07: name = input("Sag mir Deinen Namen: ") # Namen abfragen
08:
09: print("Hallo", name)
```


1.13.2 Ausführen des ersten Beispiels

Sobald Du den Quelltext vollständig eingegeben hast, kannst Du das Programm starten. Wähle dazu im Menü den Punkt **Run** -> **Run...** aus, woraufhin ein kleiner Auswahldialog erscheint. Ein Klick auf Deine Quellcode-Datei (z. B. *programm*) führt nun Dein erstes Programm aus.

Wenn alles geklappt hat, sollte sich wie in Bild 1.7 gezeigt, im unteren Bereich von PyCharm ein Ausgabefenster öffnen:

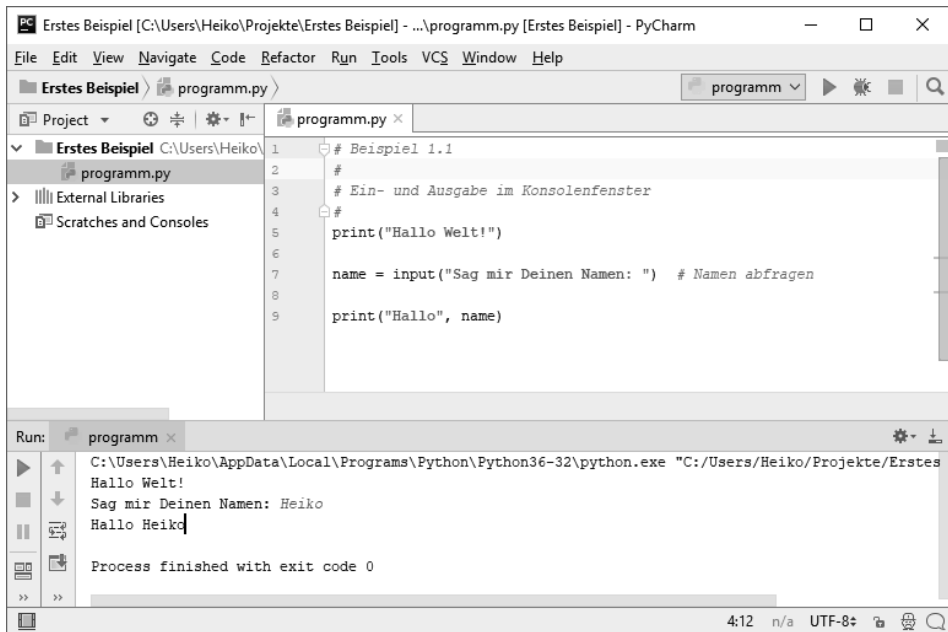


Bild 1.7 Ausführen des ersten Beispiels

Klicke nun in den Ausgabebereich, damit dieser aktiviert wird. Jetzt kannst Du Deinen Namen eingeben, woraufhin eine kleine Begrüßung ausgegeben wird. Damit hast Du Dein erstes Python-Programm geschrieben und gestartet! Wenn Du es erneut ausführen möchtest, genügt von jetzt an auch der Menüpunkt **Run** -> **Run 'main'** oder ein Klick auf den kleinen grünen Pfeil oben rechts. Ein Auswahldialog erscheint nicht mehr, denn PyCharm merkt sich die zuletzt ausgeführte Datei.

1.13.3 Laden der Beispiele

Alle in diesem Buch gezeigten Beispiele sind im Begleitmaterial vorhanden und können direkt geladen und gestartet werden. Den entsprechenden Downloadlink findest Du in Abschnitt 1.5. Lege diesen Ordner am besten in Deinem Dokumentenordner ab.

Klicke auf dem Startbildschirm von PyCharm auf **Open** und wähle den Ordner *Beispiele* aus dem Begleitmaterial. In ihm befinden sich für jedes Kapitel Unterordner, in denen wiederum die entsprechenden Quelltexte liegen. Du musst nicht jede Datei einzeln öffnen, sondern kannst direkt das übergeordnete Verzeichnis *Beispiele* verwenden. Dann hast Du, wie in Bild 1.8 zu sehen, alle Beispiele direkt zur Verfügung. Falls dabei noch einmal der Hinweis erscheint, dass kein Python-Interpreter festgelegt wurde, kannst Du das durch einen Klick auf die Schaltfläche **Configure Python interpreter** nachholen.

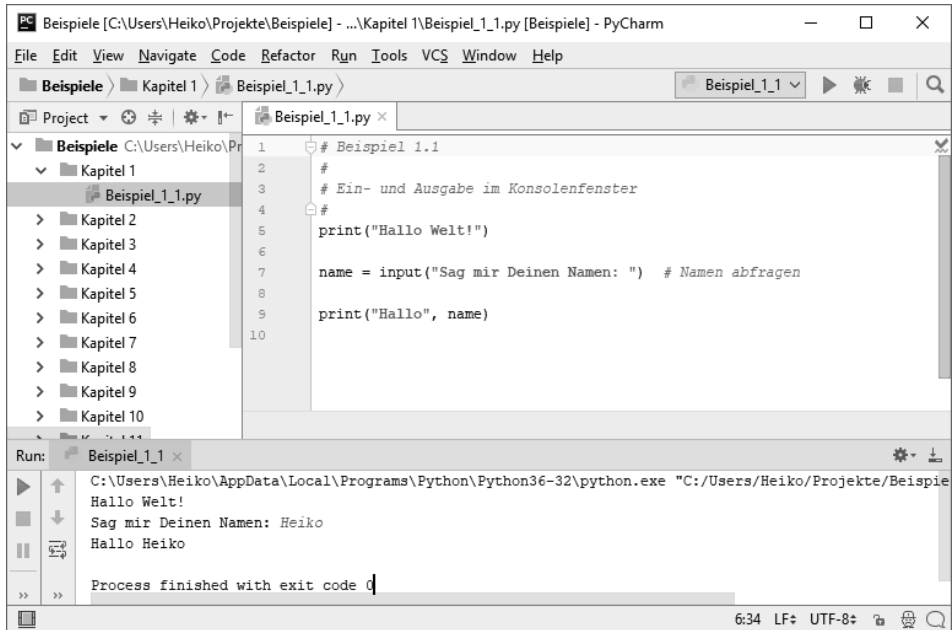


Bild 1.8 Laden der Beispiele

Jetzt kannst Du mittels der Projektübersicht auf der linken Seite jedes beliebige Beispiel durch einen Doppelklick öffnen. Klicke mit der rechten Maustaste auf **Beispiel_1_1.py** und wähle im Kontextmenü **Run 'Beispiel_1_1'** aus, um das Beispiel zu starten. Sobald es einmal gestartet wurde, kannst Du in der rechten oberen Ecke auch den grünen Start-Button klicken. Beachte jedoch, dass dieser immer die zuletzt gestartete Python-Datei ausführt. Wenn Du nun etwa durch einen Doppelklick ein anderes Beispiel öffnest und dann wieder auf den grünen Start-Button klickst, wird trotzdem *Beispiel_1_1.py* ausgeführt. Möchtest Du ein anderes Beispiel starten, dann klicke mit der rechten Maustaste darauf und wähle **Run 'Name des Beispiels'**.



HINWEIS: Sollte die Option **Run** ausgegraut und nicht anklickbar sein, dann liegt das in der Regel daran, dass PyCharm die geladenen Quelltext-Dateien noch indiziert. In diesem Fall steht in der Statusleiste am unteren Rand von PyCharm eine entsprechende Meldung. Warte kurz ab, bis das Indizieren beendet ist, dann sollte die Option im Menü verfügbar sein.

1.13.4 Der Quelltext im Detail

Nachdem das erste Beispiel erfolgreich gestartet wurde, sehen wir uns einmal an, was der Quelltext im Einzelnen bewirkt. Solltest Du bereits Erfahrungen in anderen Programmiersprachen haben, dann werden Dir bestimmte Dinge möglicherweise bekannt vorkommen. Falls Du komplett neu auf dem Gebiet der Programmierung bist, dann solltest Du trotzdem erst einmal versuchen, ohne weitere Erklärung Schlüsse aus dem Quelltext zu ziehen und vielleicht sogar ein wenig damit zu experimentieren. Keine Sorge: Das Schlimmste was passieren kann ist, dass das Programm nicht läuft. Ein wenig Neugier und Experimentierfreude gehört zum Entwicklerdasein genauso dazu wie die Fähigkeit, Unbekanntes zu analysieren. Häufig wird man mit Aufgaben konfrontiert, die man nicht auf Anhieb lösen kann. Dazu gehört auch fremder Quelltext oder gar eine bisher nicht vertraute Programmiersprache. Wer sich davor nicht scheut, sondern die Dinge analytisch angeht, hat von Anfang an einen Vorteil.

Sehen wir uns die ersten vier Zeilen an: Diese beinhalten sogenannte Kommentare und haben keinerlei Einfluss auf den Programmablauf. Kommentare kann man sich wie Notizen im Programmtext vorstellen. Gerade bei großen Programmen oder komplexen Teilabschnitten ist es sinnvoll, zu beschreiben, was im Programm vor sich geht. Weitere Details zu Kommentaren und deren Verwendung gibt es im nächsten Abschnitt. Für den Augenblick genügt es zu wissen, dass alles, was auf eine Raute (#) folgt, vom Python-Interpreter ignoriert wird und ausschließlich der Dokumentation dient.

Zeile 5 ist schon etwas interessanter, denn hier findet zum ersten Mal eine wirkliche Aktion statt, indem der Satz „Hallo Welt!“ in der Konsole ausgegeben wird. Bei `print()` handelt es sich um eine Funktion, die zum Sprachumfang von Python gehört. Auch hier müssen wir leider die Details etwas nach hinten verschieben, da das Thema Funktionen ein gesamtes Kapitel füllt. Fürs Erste sei gesagt, dass eine Funktion einen in sich abgeschlossenen und wiederverwendbaren Programmteil darstellt, dem man Daten (sogenannte Parameter) übergeben kann und der auch Daten zurückliefern kann. In diesem Fall übergeben wir der `print()`-Funktion den Satz „Hallo Welt“, der daraufhin in der Konsole ausgegeben wird. Die Datenübergabe an eine Funktion erfolgt dabei immer in runden Klammern.

Wie Du in den Zeilen 6 und 8 erkennen kannst, ist es auch möglich, leere Programmzeilen zu verwenden. Das dient lediglich dazu, den Quelltext ein wenig übersichtlicher zu gestalten. In Zeile 7 verwenden wir eine weitere Python-interne Funktion namens `input()`. Diese gibt den angegebenen Text auf der Konsole aus und wartet im Anschluss auf eine Texteingabe des Benutzers. Wie bereits oben erwähnt, können Funktionen auch Daten zurückliefern. In diesem Fall liefert die Funktion `input()` den vom Benutzer eingegebenen Text zurück. Damit dieser Text im weiteren Programmverlauf verwendet werden kann, müssen wir ihn in irgendeiner Weise speichern. Das machen wir, indem wir ihn einer sogenannten Variable zuweisen. Was möglicherweise etwas kompliziert klingt, ist im Grunde recht einfach. Eine Variable kann man sich wie eine Schublade vorstellen, in der man etwas verstauen und jederzeit wieder herausholen kann. Da man beliebig viele Variablen erzeugen und verwenden kann, muss man sie eindeutig benennen (sozusagen die Schublade beschriften). Bei `name` in Zeile 7 handelt es sich folglich um eine Variable, die den von der Funktion `input()` zurückgelieferten Text aufnimmt und speichert. Da den Variablen ebenfalls ein gesamtes Kapitel gewidmet ist, verzichten wir an dieser Stelle auf allzu viele Details. Vorerst

genügt es, zu wissen, dass in der Variablen `name` jetzt der vom Benutzer eingegebene Text steht und weiterverwendet werden kann.

Den Abschluss unseres ersten Beispiels bildet Zeile 9, die dafür sorgt, dass der Benutzer in angemessener Weise begrüßt wird. Dazu verwenden wir erneut die Funktion `print()`, der dieses Mal ein Begrüßungstext sowie die Variable `name` übergeben wird. Beide sind durch ein Komma getrennt. Das deutet darauf hin, dass einer Funktion auch mehrere Parameter übergeben werden können, die sie auswerten und verwenden kann.

Wenngleich nicht sonderlich spektakulär, so hast Du dennoch gerade Dein erstes funktionsfähiges Python-Programm geschrieben und ausgeführt. Doch bevor Du Dich mit dem nächsten Beispiel beschäftigst, solltest Du ein wenig mit dem eben gezeigten experimentieren. Versuche, den Quelltext etwas zu verändern und zu erweitern. Wie bereits gesagt, kann dabei nichts schiefgehen. Solltest Du es tatsächlich schaffen, aus Versehen ein Programm zu schreiben, dass die gesamte Festplatte formatiert, bist Du definitiv ein Kandidat für einen Sechser im Lotto.



HINWEIS: Python ist eine *case sensitive*-Sprache, was auf Deutsch so viel heißt wie *Beachtung der Groß- und Kleinschreibung*. Das bedeutet, dass einmal vergebene Funktions- und Variablennamen sowie Schlüsselwörter immer gleich geschrieben werden müssen. Die Aufrufe `Print("Hallo")` oder `PRINT("Hallo")` werden somit zu einer Fehlermeldung führen. Genauso handelt es sich bei `name` und `Name` um zwei verschiedene Variablen. Achte also immer auf die korrekte Schreibweise.

1.13.5 Kommentare im Detail

Kommentare im Quelltext sind eine ziemlich nützliche und wichtige Sache, auch wenn das in dem oben gezeigten Beispiel nicht sofort ersichtlich wird. Stell Dir einfach einmal vor, dass Du schon eine Weile an einem großen und aufwendigen Programm arbeitest, dass bereits aus hunderten oder gar tausenden Zeilen Quellcode besteht. Der Quellcode kann noch so sauber und lesbar gestaltet sein – ohne vernünftige Dokumentation werden sich früher oder später mit ziemlicher Sicherheit Probleme ergeben. Programmteile, die beim Schreiben noch logisch und eindeutig waren, müssen nach einigen Monaten nicht zwangsläufig immer noch selbsterklärend erscheinen. Es kann durchaus passieren, dass man sich nicht mehr genau erinnert, warum man sich für eine bestimmte Vorgehensweise entschieden hat oder wie ein bestimmter Algorithmus funktioniert. Zumindest geht mir das häufig so. In einer solchen Situation ist man dann froh, wenn der Quelltext sauber kommentiert ist. Man muss sich nicht mehr in allen Einzelheiten hineindenken, sondern kann den Sinn schnell anhand der Kommentare erfassen. Und selbst wenn Du zu denen gehörst, die sich auch nach Jahren noch an alle Einzelheiten erinnern (es gibt solche Leute, ich finde das immer wieder faszinierend), dann ergibt es dennoch Sinn, Kommentare zu schreiben, schließlich arbeitet man nicht immer alleine an einem Projekt. Jeder, der an Deinem Projekt mitarbeitet, wird Dir dankbar sein, wenn er den Sinn von einzelnen Programmteilen schnell und effektiv erfassen kann.

Natürlich bedeutet das nicht, dass man jede Zeile kommentieren oder ganze Romane schreiben soll. In einem richtigen Projekt würde man beispielsweise einen Kommentar, wie er in Zeile 7 des oben gezeigten Quelltextes zu sehen ist, nicht verwenden, da die Bedeutung dieser Zeile ohnehin sehr schnell ersichtlich ist. Man sollte es also vermeiden, Offensichtliches zu kommentieren.

Erwischt man sich dabei, dass ein Quellcode aus mehr Kommentaren als aus wirklichem Code besteht, dann sollte man sich überlegen, ob dieser an sich nicht zu komplex ist. Es macht keinen Sinn, zu versuchen, völlig chaotischen Quellcode durch Kommentare aufzuwerten. Nicht nur Kommentare sorgen für Verständlichkeit, sondern in erster Linie die Umsetzung an sich. Somit liegt die höchste Priorität beim Entwickeln darin, auf saubere und logische Strukturen zu achten.

Was ist schlimmer als ein Quellcode ohne Kommentare? Richtig: Veraltete oder gar falsche Kommentare! Jeder, der schon an größeren Projekten gearbeitet hat, bei denen mehrere Leute mitwirkten, hat es sicherlich schon erlebt: Ein Programmteil wurde umgeschrieben, aber die Kommentare wurden aufgrund von Zeitdruck (oder Bequemlichkeit) nicht angepasst. In diesem Fall sind diese nicht nur wertlos geworden, sondern sogar gefährlich. Achte also darauf, Kommentare immer aktuell zu halten, denn sonst ist die investierte Arbeitszeit leider verschwendet.



PRAXISTIPP: Selbsterklärender Quellcode ist hilfreicher als jeder Kommentar. Kommentiere nichts, was ohnehin offensichtlich ist. Beschreibe komplexe oder nicht weiter zu vereinfachende Quelltext-Abschnitte ausreichend. Pflege Deine Kommentare und halte sie aktuell.

Kommentare können noch auf andere Weise nützlich sein: Man kann durch sogenanntes *Auskommentieren* sehr schnell Teile des Quellcodes „ausschalten“ ohne gleich zeilenweise Code löschen zu müssen. Stell Dir beispielsweise vor, dass Du gerade ein Tool geschrieben hast, das alle Dateien in einem Ordner nach einem bestimmten Muster umbenennt. Nun möchtest Du dieses Muster ändern, bist Dir aber nicht sicher, ob Deine Anpassungen im Quelltext wirklich richtig sind. In diesem Fall wäre es nützlich, die neuen Dateinamen zuerst einmal in der Konsole auszugeben, um zu sehen, ob alles klappt. Der Teil des Quelltextes, der für das eigentliche Umbenennen zuständig ist, soll dabei natürlich nicht ausgeführt werden. Doch dazu müssen keine Zeilen gelöscht werden. Es genügt völlig, einfach eine Raute (#) vor die betreffenden Zeilen zu schreiben. Schon werden diese vom Interpreter als Kommentar gesehen und nicht mehr ausgeführt. Sobald man sich sicher ist, dass alles korrekt umprogrammiert wurde, entfernt man die Kommentare wieder (einkommentieren).



PRAXISTIPP: Wenn Du häufig Programmteile auskommentierst, dann achte darauf, keinen toten Code zu produzieren. Zeilen, die sich später als unnötig erweisen, sollten endgültig gelöscht werden. Auskommentieren ist eine temporäre Hilfe und nichts für die Ewigkeit. Ein Quellcode, der zu großen Teilen aus totem Code besteht, ist schlecht zu lesen. Wenn es einen wirklich guten Grund gibt, etwas länger auskommentiert zu lassen, dann dokumentiere es – mit einem Kommentar!

Möchtest Du einen größeren Programmteil auskommentieren, dann bietet Dir Python eine einfachere Möglichkeit, als jede Zeile mit einer Raute beginnen zu lassen. Drei einfache oder auch drei doppelte Anführungszeichen leiten einen langen Kommentar ein und schließen ihn auch wieder ab. Das sieht folgendermaßen aus:

```
"""
Alles innerhalb dieses Blocks wird nicht ausgeführt.
Das ist bequemer und übersichtlicher.
"""

'''
So geht es auch.
Leider ist das nicht in jeder Situation ideal.
'''
```



HINWEIS: Wie immer gibt es auch hier etwas zu beachten: Diese Art von Kommentar hat in Python eine bestimmte Bedeutung und dient dazu, Dokumentationen zu generieren. Ein mehrzeiliger Kommentar an einer beliebigen Stelle im Quellcode sollte somit also mit vorangestellten Rauten realisiert werden. Es spricht jedoch nichts dagegen, auf diese Weise kurzzeitig einen größeren Teil Programmcode auszukommentieren.

■ 1.14 PEP8 – um sie ewig zu binden

In Abschnitt 1.13.4 hast Du erfahren, dass Python eine *case sensitive*-Sprache ist und somit zwischen Groß- und Kleinschreibung unterscheidet. Daher ist man gewissen Regeln unterworfen, wenn man Python-Programme schreibt. Das bedeutet jedoch nicht, dass man keine Freiheiten hat. Wie wir später sehen werden, kann man beispielsweise Funktions- und Variablennamen fast beliebig wählen, es gibt nur sehr wenige Ausnahmen, auf die man achten muss. Auch ist es möglich, den Quelltext beispielsweise durch Einfügen von Leerzeilen ein wenig zu gliedern, wie es auch in Beispiel 1.1 gemacht wurde. Somit wird deutlich, dass jeder die Möglichkeit hat, seinem eigenen Quellcode den eigenen Stempel aufzudrücken. So schön demokratisch diese Freiheit auch klingen mag: Es ist nicht immer sinnvoll, sich seinen eigenen Stil auszudenken und anzugewöhnen.

Als Programmierer ist man in den wenigsten Fällen komplett auf sich alleine gestellt und selbst wenn doch, so gibt es immer wieder Situationen, in denen man sich mit anderen austauscht. Sei es aktiv durch Zusammenarbeit an einem Projekt oder rein passiv, indem man im Internet nach Problemlösungen oder freiem Quellcode sucht. Mit zunehmender Erfahrung wirst Du feststellen, dass es durchaus hinderlich sein kann, wenn jeder sein eigenes Süppchen kocht. Immer dann, wenn eine Zusammenarbeit ansteht, müssen sich alle Beteiligten in die Eigenheiten des Quellcodes der jeweils anderen hineindenken. Um diesem Problem entgegenzutreten, gibt es sogenannte *Style Guides*, zu Deutsch etwa *Gestal-*

tungsrichtlinien. Diese sind entgegen der Überschrift dieses Abschnitts zwar nicht bindend, aber dennoch durchaus sinnvoll.

PEP ist die Abkürzung für *Python Enhancement Proposal*, was auf Deutsch etwa so viel bedeutet wie Verbesserungsvorschläge oder Anregungen für Python. Während der gesamten Entwicklungszeit von Python wurden und werden PEP-Dokumente erstellt und weiterentwickelt. So gibt es neben dem eben angesprochenen Style Guide (*PEP8*) beispielsweise auch Dokumente zur geplanten weiteren Entwicklung von Python (*PEP537 - Python 3.7 Release Schedule*). Eine Liste aller PEPs findest Du unter folgendem Link: <https://www.python.org/dev/peps>

Zusammengefasst und etwas vereinfacht kann man sagen, dass PEP8 (sehr detaillierte) Vorschläge zur Formatierung, Namensgebung und Strukturierung in Python-Programmen enthält. Dazu zählen auch Dinge wie etwa die maximale Länge einer Zeile, maximale Anzahl aufeinanderfolgender Leerzeilen oder Abstand zwischen Programmzeile und Kommentaren. Auch wenn diese Vorschläge nicht bindend sind, richtet sich dieses Buch nach diesen Vorgaben und ich kann Dir empfehlen, dies ebenfalls zu tun. Du wirst sehen, dass es wirklich einfacher ist, fremden Quelltext zu lesen, der auch auf PEP8 aufbaut. Man muss weniger umdenken, da alles konsistenter und schlüssiger ist.

Im Augenblick macht es natürlich noch keinen Sinn, alle Aspekte von PEP8 vorzustellen. Vielmehr werde ich immer dann auf entsprechende Style Guides eingehen, wenn neue Themen angesprochen werden. Natürlich musst Du nicht jede Regel auswendig lernen, denn das meiste kommt mit der Zeit von selbst. Außerdem unterstützt Dich PyCharm beim Schreiben des Quellcodes, denn es kann Deinen Quellcode automatisch auf PEP8-Konformität überprüfen. Das kannst Du beispielsweise sehen, wenn Du einmal mehrere leere Zeilen einfügst oder ein Leerzeichen vor dem Kommentar in Zeile 7 von Beispiel 1.1 entfernst. In diesen Fällen fügt PyCharm eine kleine, gewellte Linie ein. Fahre einfach mit der Maus darüber, um zu erfahren, was nicht stimmt. Daraufhin sollte ein Hinweis wie in Bild 1.9 gezeigt erscheinen.

```

1 # Beispiel 1.1
2 #
3 # Ein- und Ausgabe im Konsolenfenster
4 #
5 print("Hallo Welt!")
6
7 name = input("Sag mir Deinen Namen: ") # Namen abfragen
8
9 print("Hallo", name # PEP 8: at least two spaces before inline comment
10

```

Bild 1.9 PEP8-Konformität: Auf den ersten Blick pedantisch, auf den zweiten sehr nützlich.

■ 1.15 Python-Programme ohne Entwicklungsumgebung starten

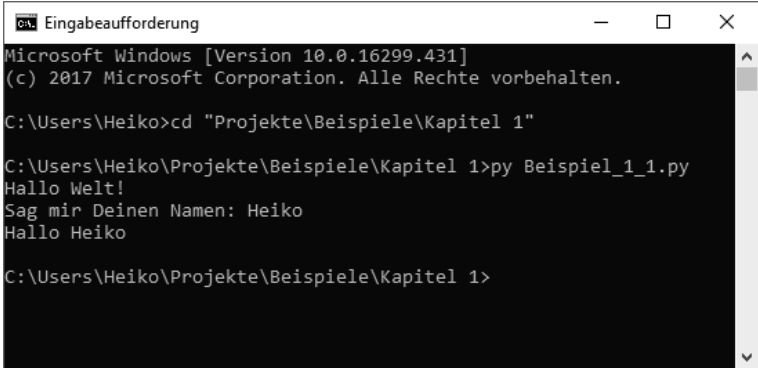
Unser erstes Programm hat zwar nur Beispielcharakter und erfüllt keinen wirklichen Zweck, aber tun wir einfach mal so, als ob es eine Anwendung wäre, die wir an jemanden weitergeben, damit dieser sie benutzen kann. In diesem Fall erwartet man selbstverständlich nicht, dass dieser Jemand sich extra eine Entwicklungsumgebung installiert. Glücklicherweise kann man Python-Programme einfach aus einem Konsolen-, respektive Terminal-Fenster heraus starten. Einzige Voraussetzung ist natürlich immer ein installierter Python-Interpreter. Schauen wir uns einmal an, wie das funktioniert. Solltest Du noch keine Erfahrung in der Arbeit mit der Konsole oder dem Terminal haben, dann schau Dir die Vorgehensweise einfach in den zugehörigen Bildern ab und ersetze die Pfade entsprechend. Zudem gibt es in den Hinweiskästen noch weitere Hilfestellungen.

1.15.1 Python-Programme unter Windows starten

Öffne ein neues Konsolenfenster, indem Du entweder die **Windows**-Taste drückst und anschließend **cmd**, gefolgt von der **Return**-Taste eingibst, oder im Startmenü den Eintrag **Windows-System -> Eingabeaufforderung** auswählst. Wechsle anschließend mit **cd Pfadname** in das Verzeichnis, in dem sich das gewünschte Python-Programm befindet. Unter Windows wird mit Python 3.7 auch der Python-Launcher namens **py** installiert. Dieser hilft Dir dabei, auf einfache Weise sowohl Python 3.7 als auch eventuell ältere installierte Versionen zu starten. Die Benutzung des Launchers ist recht einfach, wie Du in Bild 1.10 sehen kannst:

```
py [Version] Dateiname.py
```

Beachte, dass die Angabe der Python-Version optional ist und auch nur eine Rolle spielt, wenn Du verschiedene Python-Interpreter installiert hast. Schreibst Du beispielsweise **py meinprogramm.py**, dann wird automatisch der aktuellste Interpreter gestartet. Möchtest Du hingegen Python 2.7 verwenden, dann schreibe **py -2 meinprogramm.py**.



```
Eingabeaufforderung
Microsoft Windows [Version 10.0.16299.431]
(c) 2017 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Heiko>cd "Projekte\Beispiele\Kapitel 1"

C:\Users\Heiko\Projekte\Beispiele\Kapitel 1>py Beispiel_1_1.py
Hallo Welt!
Sag mir Deinen Namen: Heiko
Hallo Heiko

C:\Users\Heiko\Projekte\Beispiele\Kapitel 1>
```

Bild 1.10 Ein Python-Programm unter Windows aus der Konsole heraus starten



PRAXISTIPP: Mittels `cd` *Verzeichnisname* (`cd` steht für *change directory*) kannst Du in das angegebene Verzeichnis wechseln. Zurück zum übergeordneten Verzeichnis kommst Du mit `cd..`. Wenn Du Dir alle Ordner und Dateien im aktuellen Pfad anzeigen lassen willst, dann verwende `dir`.

1.15.2 Python-Programme unter Linux oder macOS starten

Öffne zunächst ein Terminal und wechsle mit `cd` *Pfadname* in das Verzeichnis, in dem sich Dein Python-Programm befindet. Sobald Du Dich im richtigen Verzeichnis befindest, genügt es, `python3` gefolgt vom Namen des Programms einzugeben. Denke daran, dass Python 2 und Python 3 nicht miteinander kompatibel sind. Wenn Du ein Programm starten möchtest, das auf Python 2 basiert, dann verwende einfach `python` *programmname.py* statt `python3` *programmname.py*. Wie das aussehen soll, siehst Du in Bild 1.11 und Bild 1.12.

```

Kapitel 1 -- bash -- 60x16
Heikos-iMac:~ heiko$ cd Projekte/Beispiele/Kapitel\ 1
Heikos-iMac:Kapitel 1 heiko$ python3 Beispiel_1_1.py
Hallo Welt!
Sag mir Deinen Namen: Der steht doch oben!
Hallo Der steht doch oben!
Heikos-iMac:Kapitel 1 heiko$

```

Bild 1.11 Ein Python-Programm unter macOS aus dem Terminal heraus starten

```

heiko@heiko-VirtualBox: ~/Projekte/Beispiele/Kapitel 1
Datei Bearbeiten Ansicht Suchen Terminal Hilfe
heiko@heiko-VirtualBox:~$ cd Projekte/Beispiele/Kapitel\ 1
heiko@heiko-VirtualBox:~/Projekte/Beispiele/Kapitel 1$ python3 Beispiel_1_1.py
Hallo Welt!
Sag mir Deinen Namen: Heiko
Hallo Heiko
heiko@heiko-VirtualBox:~/Projekte/Beispiele/Kapitel 1$

```

Bild 1.12 Ein Python-Programm unter Ubuntu 18.04 aus dem Terminal heraus starten



PRAXISTIPP: Mittels `cd Verzeichnisname` kannst Du das Verzeichnis wechseln. Hierbei ist die Groß- und Kleinschreibung wichtig. Ein Verzeichnis zurück kommst Du mit `cd ..`. Achte dabei auf das Leerzeichen zwischen `cd` und `..`. Mit `ls` kannst Du Dir alle Ordner und Dateien im aktuellen Pfad anzeigen lassen. `pwd` zeigt Dir an, in welchem Verzeichnis Du Dich gerade befindest.

■ 1.16 Python interaktiv

Der Python-Interpreter kann nicht nur fertige Python-Programme ausführen, sondern auch im sogenannten interaktiven Modus gestartet werden. In diesem Modus läuft der Interpreter in einer Konsole und führt jede vom Benutzer eingegebene Zeile in der Regel sofort aus. Bevor ich im Detail auf die Vor- und Nachteile dieses Modus eingehe, sehen wir uns das erst einmal in der Praxis an. Öffne dazu ein Konsolenfenster und tippe einfach nur `py` (unter macOS oder Linux `python3`) ein. Nach einem Druck auf die **Return**-Taste startet Python im interaktiven Modus, was an den drei spitzen Klammern (`>>>`) zu erkennen ist. Gib nun einfach einmal die Zeile `print("Das ist interaktiv")` ein und schau Dir das Ergebnis an. Der gewählte Text wird direkt in der Konsole ausgegeben und der Interpreter wartet auf die nächste Eingabe.

Wird eine Eingabe vom Benutzer erwartet, wie es beispielsweise bei der Funktion `input()` der Fall ist, so wird einfach eine neue Zeile erzeugt und auf die Eingabe gewartet. Auch die Variablenzuweisung funktioniert im interaktiven Modus. Der Interpreter merkt sich so lange den Inhalt einer Variablen, bis der interaktive Modus wieder verlassen wird. Bild 1.13 zeigt, wie so etwas aussieht:

```
(c) 2017 Microsoft Corporation. Alle Rechte vorbehalten.
C:\Users\Heiko>py
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32
bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.

>>> name = input("Sag mir Deinen Namen: ")
Sag mir Deinen Namen: Heiko
>>> print(name)
Heiko
>>> 4*8+15*16
272
>>> exit()

C:\Users\Heiko>
```

Bild 1.13 Python im interaktiven Modus

Doch welche Vor- und Nachteile bietet dieser interaktive Modus und wann ist es sinnvoll, ihn zu benutzen? Der erste Vorteil liegt auf der Hand: Wenn man „mal eben“ etwas ausprobieren möchte, muss man nicht extra einen Editor oder gar eine Entwicklungsumgebung starten, sondern kann direkt loslegen. Ein weiterer Vorteil ist, dass mathematische Ausdrücke direkt ausgewertet und das Ergebnis angezeigt wird. Kurz: Man kann den interaktiven Modus jederzeit auch als Taschenrechner verwenden. Probiere das einfach mal aus, indem Du etwa $4*8+15*16$ eingibst.

Gerade wenn man die ersten Schritte in Python macht, muss man häufig nachschlagen, wie bestimmte Befehle oder Funktionen richtig verwendet werden. Befindet man sich im interaktiven Modus, kann man sich häufig den Umweg über den Browser und die Suchmaschine sparen. Bedenke jedoch, dass die Hilfe nur auf Englisch verfügbar ist. Möchtest Du beispielsweise mehr Details zu `print()` oder `input()` wissen, dann gib einfach einmal `help(print)`, respektive `help(input)` ein. Du wirst folgende Bildschirmausgabe sehen:

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

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

(END)
```

Der interaktive Modus bietet sich auch dann besonders an, wenn man auf die Schnelle eine Aufgabe erledigen möchte, die wenig Programmieraufwand benötigt und die nur ein einziges Mal benötigt wird. Das könnten beispielsweise Dateioperationen, wie Löschen oder Umbenennen mehrerer Dateien anhand bestimmter Muster sein. Ebenfalls möglich wäre es etwa, alle in einem Ordner befindlichen *.jpg*-Bilder in das *.png*-Format zu konvertieren. Die Einsatzmöglichkeiten sind vielfältig. Doch bei so vielen tollen Vorteilen gibt es natürlich auch einige Nachteile. So ist es zwar möglich, auch mehrere Zeilen hintereinander einzugeben, bevor die eigentliche Ausführung stattfindet. Editieren lassen sich diese Zeilen nach der Eingabe allerdings nicht mehr. Schreibt man also ein mehrzeiliges Programm im interaktiven Modus, so darf einem dabei kein Fehler passieren. Somit sollte auch klar sein, dass man sich grundsätzlich auf kurze und überschaubare Programme beschränkt. Für aufwendigere Aufgaben sollte man lieber wieder einen Editor oder eine Entwicklungsumgebung verwenden. Wie man mehrzeilige Programme im interaktiven Modus schreibt, sehen wir in Kapitel 3.

Wenn Du den interaktiven Modus verlassen möchtest, dann verwende entweder die Funktion `exit()` oder die Tastenkombination `ctrl+D` (macOS), `Strg+Z+Return` (Windows) oder `Strg+D` (Linux).



PRAXISTIPP: Mit den Tasten **Pfeil nach oben**/**Pfeil nach unten** kannst Du die zuletzt eingegebenen Befehle wiederholen, ohne diese nochmals eingetippen zu müssen.

■ 1.17 Aufgabenstellung

In diesem Kapitel gab es zwar bereits eine geballte Ladung an neuem Wissen und interessanten Dingen, allerdings wurden noch nicht wirklich genügend Programmierkenntnisse vermittelt, um eine entsprechende Aufgabe zu stellen. Dennoch gibt es wichtige Dinge, die Dir in Fleisch und Blut übergehen sollten. Es ist das Eine, ein neues Python-Projekt mit einer Anleitung zu erstellen, das Andere jedoch, die nötigen Schritte wirklich selbst zu erledigen. Das Gleiche gilt für den Inhalt des ersten Beispiels. Wenn Du es Dir anschaust, dann scheint alles überschaubar und sehr einfach. Doch gerade wenn Du bisher noch nie mit der Programmierung in Berührung gekommen bist, wird die Sache vermutlich etwas anders aussehen, wenn Du das bisher Gelernte ohne Erklärungen umsetzen sollst. Von daher besteht die erste Aufgabe nur darin, dieses Buch zur Seite zu legen und zu versuchen, das Beispiel mit der Namensabfrage einmal eigenhändig umzusetzen. Versteh mich an dieser Stelle jedoch bitte nicht falsch: Es geht nicht darum, Dinge einfach nur auswendig zu lernen. So etwas hilft in der Programmierung nicht wirklich weiter. Vielmehr liegt der Fokus darauf, dass Du Dich mit Deiner Entwicklungsumgebung und den für die Erstellung eines einfachen Programms notwendigen Schritten vertraut machst.

Sobald es Dir gelungen ist, das Beispiel auf eigene Faust zum Laufen zu bekommen, solltest Du ein wenig mit dem Quellcode experimentieren und versuchen, ihn ein wenig zu erweitern. Das Einzige, was schiefgehen kann, ist, dass Dir PyCharm die eine oder andere Fehlermeldung um die Ohren haut. Entscheide an dieser Stelle selbst, ob Du versuchen möchtest, die Fehlermeldung zu verstehen, vielleicht sogar im Internet nachzuschlagen und das Problem zu lösen. Investiere dabei jedoch nicht zu viel Zeit, denn wir werden noch früh genug zu den entsprechenden Themen kommen. Ein bisschen Experimentierfreude und Neugier gehören jedoch einfach zum Programmierenlernen dazu.

■ 1.18 Kurz & knapp

In diesem Kapitel hast Du erfahren, dass Python eine sehr universell einsetzbare Programmiersprache ist, die bis auf wenige Ausnahmen für fast alle erdenklichen Einsatzzwecke verwendet werden kann. In sehr vielen aktuellen Projekten wird immer noch Python 2.7 verwendet, wohingegen die aktuellste Version 3.7 ist. Bei Python handelt es sich um eine interpretierte Programmiersprache, die ohne entsprechenden Interpreter in der Regel nicht ausgeführt werden kann. Bei kompilierten Programmiersprachen wird der Quellcode hingegen direkt in ausführbaren Maschinencode übersetzt.

Um Programme mit Python zu entwickeln, genügt bereits ein Texteditor und ein installierter Interpreter, mit dem sich die Python-Programme ausführen lassen. Weiterhin gibt es vollständige Entwicklungsumgebungen (*IDEs*), die weit mehr leisten, als ein einfacher Texteditor. Beispielsweise ist es möglich, darin ein Python-Programm Zeile für Zeile zu durchlaufen und mittels spezieller Werkzeuge Fehler aufzuspüren. In diesem Buch wird die kostenfreie *PyCharm Community Edition* von *JetBrains* verwendet. Auf die grundlegende

Funktionsweise dieser IDE wurde eingegangen, indem ein erstes, funktionsfähiges Projekt angelegt, der Quelltext eingegeben und ausgeführt wurde. Alternativen zu PyCharm sind unter anderem IDLE, Thonny und Visual Studio Code.

Das erste Beispiel zeigte, wie man Kommentare im Quelltext einfügt und wann man diese verwenden sollte. Des Weiteren wurde auf die Python-internen Funktionen `print()` und `input()` eingegangen. Mit `print()` können Texte und der Inhalt von Variablen in der Konsole ausgegeben und mit `input()` Texte vom Benutzer abgefragt und in Variablen gespeichert werden.

Python-Programme können nicht nur über eine IDE gestartet werden, sondern auch direkt über eine Konsole respektive ein Terminal. Eine weitere Möglichkeit stellt der interaktive Modus dar. Mit diesem ist es möglich, Python-Anweisungen direkt auszuführen. Das ist etwa dann nützlich, wenn man auf die Schnelle etwas ausprobieren will oder eine kurze, nicht häufig benötigte Aufgabe ausführen möchte.

Index

Symbole

- _ 81
- \ 286
- # 23
- __enter__() 305
- __exit__() 305
- __getitem__(self, item) 345
- .idea 18
- .idea-Ordner 412
- __init__ 134
- __init__.py 296
- __main__ 112, 295
- % (Modulo-Operator) 49
- __name__ 112, 294
- __setitem__(self, key, value) 345
- @staticmethod 150
- __str__() 182

A

- Ableitung 165
- Absolute Pfade 317
- Abwärtskompatibilität 8
- add() 247
- and 69
- Anonyme Funktionen 203
- Anwendungsgebiete 6
- append() 192
- App-Entwicklung 7
- Arbeitsverzeichnis 314
- args (Exceptions) 264
- Argumente von Funktionen 100
- as 271
 - as (with) 305
 - bei Import von Modulen 288

- Assembler 7
- Attribute (einer Klasse) 133
- AttributeError 138
- Aufrufbares Objekt 203
- Aufrufen einer Funktion 98, 101
- ausführen eines Python-Programms 20
- Ausführungsgeschwindigkeit 6
- Ausgabefenster 20
- Auskommentieren von Quelltext 24
- Ausnahme, unbehandelte 264
- Ausnahmebehandlung 82, 261
- Ausnahmefehler 83

B

- BaseException 264, 269
 - Ableitung 270
- basename() 316
- Basisklasse 164
- Bedingungen 59
- Begleitmaterial zum Buch 4, 20
- Beispiele im Buch 20
- Benennung von Variablen 38
- Betriebsblindheit 54
- Binärdatei 311
- Binden von Methoden 174
- bool 34
- break 72
- Breakpoints 379
- Buffer Overflow 123
- Bugs 377
- Bytecode 8

C

Callable 203
 Camel Case 39
 case sensitive 23, 227
 cget() 345
 chdir() 314
 class 133
 clear() 241
 close() 304
 Code-Blöcke 61
 – im interaktiven Modus 67
 Community 5
 Compiler 7
 const 50
 Container 189
 – iterieren 191
 continue 74
 copy() 326
 Copy-and-paste 117
 copytree() 326
 count() 230

D

Dateien
 – hinzufügen 19
 – versteckte 314
 Datei-Encoding 304
 Dateiojekt 304
 Datenkapselung 140
 Datenstrom 307
 Datentyp 34, 42
 Debugger 378
 – Activation policy 401
 – Ausführen bis Cursor 388
 – Breakpoints 379
 – Compare Value with Clipboard 391
 – Copy Value 391
 – Einzelschrittmodus 382
 – Force Run to Cursor 389
 – Force Step Over 388
 – Frames 381, 387
 – in Funktionen springen 387
 – Haltepunkte 379
 – Haltepunkte benennen 396
 – Haltepunkte bei Exceptions 383
 – Haltepunkte löschen 395
 – Ignore library files 401
 – interaktive Konsole 385

– Jump to Cursor 388
 – Jump to Source 392
 – On raise 401
 – On termination 401
 – Remove once hit 399
 – Run to Cursor 388
 – Smart Step Into 389
 – Stacktrace 398
 – starten 380
 – Step Into 387
 – Step Into My Code 388
 – Step Over 382, 388
 – Suspend 398
 – View Breakpoints 393
 – Watches mit Ausdrücken 391

deepcopy() 209

def 98

Definieren einer Funktion 98

del 195, 241

Dezimaltrennzeichen 35

Dictionaries 235

– dict_items 239

– dict_keys 240

– dict_values 240

– Reihenfolge 236

difference() 251

difference_update() 252

Differenz zweier Mengen 251

dirname() 316

discard() 248

Division durch null 83

Docstrings 286

– Konventionen 287

Dokumentation des Quelltexts 23

Duck Typing 35

Dynamische Anzahl von Instanzen
 189

dynamische Attribute 142

Dynamisch typisiert 34

E

Echte Teilmengen 252

Eigene Sortierfunktion 201

Einkommentieren von Quelltext 24

Einrücken

– Einrücktiefe von Quellcode 61

– im interaktiven Modus 67

– von Quelltext 61

- Einsatzzwecke von Python 6
- elif 64
- else 62
- bei while-Schleifen 75
- in Exceptions 272
- Embedded-Entwicklung 7
- Encoding 304
- end 217
- Endlosschleife 73
- Ententest 35
- Entwicklungsumgebung 9
- Etcher 451
- except 82, 263, 266
- Exceptions 82, 261
 - Ableitung 270
 - Ableitungshierarchie 278
 - alle abfangen 268
 - args 264
 - durchreichen 268
 - ignorieren 276
 - mehrere abfangen 266
 - testen 274
- Exception-Klasse 264
- Exception-Typen 263
- exists() 315
- exit() 30
- extend() 207

F

- False 34
- Fehlermeldungen 43
- Fehlerquelltexte zum Üben 3
- Fehlersuche (Strategien) 377
- FileExistsError 319
- FileNotFoundError 306
- finally 272
- find() 230
- Flaches Kopieren 207
- Fließkommazahl 34
 - Genauigkeit 47
- float 34
- flush() 310
- format() 46
- Formatierte Ausgabe mit print() 45
- Formatierung (Quellcode) 26
- for-Schleifen 76
- Forum zum Buch 5
- Fragen zum Buch 5

- from 91, 289
 - x import * 289
 - x import y 289
- frozenset() 246
- Frozensets 245
- Führende Nullen 116
- function decorator 150
- Funktionen 97
 - anonyme 203
 - Argumente 100
 - Aufruf 98, 101
 - Einsatzweise 116
 - Namen 99
- Funktionsrumpf 98

G

- Ganzzahl 34
- Ganzzahliger Rest einer Division 49
- Ganzzahliger Teil einer Division 45
- Genauigkeit von Fließkommazahlen 47
- Geschichte von Python 5
- Geschwindigkeit von Python-Programmen 8
- Gestaltungsrichtlinien für Quellcode 26
- get() 237
- getcwd() 314
- Getter 136
- Git *siehe* Versionsverwaltung
- global 106
- Globaler Namensraum 105, 110
- Globale Variablen, Nachteile 103, 109, 111
- Grafik-Engine 7
- Grafische Benutzeroberfläche 331
- Groß- und Kleinschreibung 23
- Grundrechenarten 41
- GUI 331
- Guido van Rossum 5

H

- Hallo Welt! 22
- help() 30, 286
- Hilfe bei Problemen 5
- Hochsprache 7

I

- id() 218
- IDE 9

idea-Ordner 18
 IDLE 10, 450
 if-Bedingung 60
 Implementierungsdetails 138
 import 91, 285
 – as 288
 – automatisches Ausführen 294
 – from x import * 289
 – from x import y 289
 – Konflikte 290
 – Mehrdeutigkeiten auflösen 297
 – Suchpfade 291
 Import in den globalen Namensraum 289
 ImportError 298
 in 193, 230, 237
 IndexError 125
 Index-Operator 191
 input() 22, 43
 insert() 193
 Instanzbezogene Bindung 178
 Instanzen 134
 Instanziierung 134
 int 34
 Integrierte Entwicklungsumgebung 9
 Interaktiver Modus 29, 67
 Interpreter 7
 intersection() 250
 IOError 306
 is 162
 IsADirectoryError 306
 isdir() 315
 isfile() 315
 issubset() 251
 issuperset() 252
 items() 239
 iterierbare Objekte 191
 iterieren 76

J

join() 223
 join() (Dateipfade) 317

K

Kapseln von Aufgaben 102
 KeyboardInterrupt 269
 key (Sortierung) 199
 key/value pair 235

keys() 240
 Klasse
 – Ableitung 165
 – Attribute 133
 – Aufbau 133
 – Binden von Methoden 174
 – definieren 132
 – dynamische Attribute 142
 – expliziter Methodenaufruf 170
 – Getter 136
 – Instanz 134
 – instanzbezogene Bindung 178
 – Konstruktor 135
 – Mehrfachvererbung 171
 – Methoden 133
 – Methoden überschreiben 166
 – Properties 136
 – Setter 139
 – statische Methoden 149, 180
 – Vererbung 163
 Klassenattribute 145
 Kommentare im Quelltext 22f.
 Konsole 27
 Konstanten 35, 50
 Konstruktor 135
 Kontext-Manager 305
 Konventionen für Variablennamen 51
 Konvertierung von Datentypen 43
 Korrekturvorschläge von PyCharm 70

L

lambda 203
 Laufwerksbuchstaben 314
 Laufzeitfehler 261
 Launcher 27
 Leere Code-Blöcke 66
 Leerzeichen entfernen (Strings) 225
 len() 193, 239
 listdir() 314
 Listen 190
 Logische Operatoren 67
 Lokale Variablen 103
 lower() 228
 lstrip() 226

M

- main()-Funktion 112
- makedirs() 321
- Maschinsprache 7
- maxsplit 225
- Mehrere Rückgabewerte 212
- Mehrfachvererbung 171
 - Mehrdeutigkeit 172
 - Nachteile 172
- mehrzeilige Kommentare 25
- Menge 245
- Mengenoperationen 248
- Methoden (einer Klasse) 133
- Minecraft 4
 - Drained Data 467
 - Kommunikation mit Python 458
 - Python API 458
- Minecraft Pi 448
- mkdir() 318, 321
- mode (beim Öffnen einer Datei) 310 ff.
- mode (Erzeugen von Verzeichnissen) 321
- Modul 283
 - dokumentieren 286
 - einzelne Funktionen importieren 289
 - importieren 285
 - importieren in den globalen Namensraum 289
 - Konflikte beim Importieren 290
 - Mehrdeutigkeiten auflösen 297
 - mehrere Module importieren 285
 - Namenskonventionen 288
 - Suchpfade 291
- ModuleNotFoundError 291
- Modulo-Operator 49
- Modus, interaktiver 29
- Monkey Patch 181
- move() 322, 325

N

- Nachkommaanteil 55
- namedtuple 214
- NameError 85
- Namensbindung 37
- Namensraum 104
 - globaler 105
 - umbenennen 288
- None 161

- not 69
- NotADirectoryError 319

O

- Objekt, aufrufbares 203
- Objektorientierung 131
 - Sinn und Zweck 151
- open() 304
- or 69
- os, Modul 314
- os.path 315

P

- packing 213
- Pakete 295
 - Abwärtskompatibilität 298
 - Initialisierungsdatei 298
- Parameter 22
- Parameterliste einer Funktion 98
 - Schlüsselwortparameter 114
 - Standardwerte 113
- Pascal Case 39
- pass 66
- PEP8 25
- PermissionError 306, 319
- Plattformunabhängigkeit 6
- pop() 195, 241, 248
- popitem() 241, 244
- Positionsbezogene Parameterübergabe 101
- print() 22, 45
 - formatierte Ausgabe 45
 - Formatierung von Fließkommazahlen 48
 - Platzhalter 46
- Programm starten 20
- Programmierung, prozedurale 131
- Projekt anlegen 18
- Properties 136
- Prozedurale Programmierung 131
- Punktoperator 135
 - mehrfache Verwendung 154
- Punkt-vor-Strich-Regel 41
- PVM 8
- py 27
- PyCharm 9
- Python 2.7 8
- Python 3.7 8
- Python Enhancement Proposal 26

Python-Interpreter 8
 Python-Launcher 27
 python virtual machine 8

Q

Quellcode, sinnvoll aufteilen 283
 Quelltext eingeben 18

R

raise 264, 268
 randint() 92
 range() 78
 Raspberry Pi 4, 447
 Raspberry Pi Zero 449
 Raspian Stretch 448
 read() 304, 307
 readline() 307f.
 Rechenoperationen 40
 Rechtschreibprüfung 16
 Reihenfolge in Dictionaries 236
 Rekursion 122
 Rekursionstiefe 123
 Relative Pfade 317
 remove() 194, 248, 319
 removedirs() 322
 rename() 322
 replace() 234, 322
 return 100
 reverse (Sortierung) 199
 rfind() 230
 Richtlinien für Variablennamen 38
 rmdir() 319
 rmtree() 320
 rsplit() 225
 rstrip() 226
 Rückgabewert von Funktionen
 100
 RuntimeError 243

S

Schleifen 59
 – verschachtelte 78, 92
 Schleifen vorzeitig beenden 72
 Schleifenvariable 78
 Schlüssel-Wert-Paar 235
 Schlüsselwortparameter 114

Schnittmenge 250
 seek() 307
 Selbsterklärender Quellcode 24
 self 134
 Separator in Strings 224
 Sequenzielle Container 190
 set() 246
 Sets 245
 Setter 139
 shadowing 110
 shutil 320
 site-packages 292, 298
 Slicing 219
 Snake Case 39
 sort() 199
 sorted() 202
 Speicherreservierung 33
 split() 223, 316
 splitext() 316
 Stack- oder Stapelspeicher 123
 Stack Overflow 123
 Standardbibliothek 290
 Standardwerte für Parameter 113
 Stapelüberlauf 123
 Statische Methoden 149, 178
 Statisch typisiert 34
 Steuerung des Programmablaufs 59
 Strategien zur Fehlersuche 377
 Strings 35, 216
 – addieren 222
 – multiplizieren 223
 strip() 226
 Strukturierung von Quelltext 116
 Style Guides 25
 super() 170
 swapcase() 228
 symmetric_difference() 251
 Symmetrische Differenz zweier Mengen 251
 Syntax Error 261
 Syntax-Highlighting 38
 sys.path 291
 SystemExit 269

T

Taschenrechner 30
 Teilmenge 251
 tell() 307
 Terminal 27

- Textdatei auslesen 304
- Thonny 10, 450
- Tiefes Kopieren 207
- tk 334
- tkinter 331
 - absolute Positionierung 332
 - add_cascade() 370
 - add_command() 370
 - add_separator() 370
 - anchor 334
 - automatischer Zeilenumbruch 352
 - BaseWidget 364
 - BooleanVar 349
 - CENTER 334
 - Checkboxes/Checkbuttons 354
 - columnspan 340
 - command 347
 - configure() 345
 - curselection() 361
 - delete() 353, 361
 - Dialogfenster 366
 - DoubleVar 349
 - Eingabefelder 349
 - Entry 349
 - font 352
 - Frame 337
 - grid() 339
 - grid_columnconfigure() 343
 - grid-Manager 338
 - grid_rowconfigure() 343
 - insert() 353
 - IntVar 349
 - Kontrollvariablen 349
 - Label 334
 - Layouts 332
 - Listboxen 359
 - mainloop() 334
 - Master Widget 334
 - maxsize() 344
 - Mehrfachauswahl 361
 - Menüs 369
 - messagebox 367
 - minsize() 344
 - modale Dialoge 367
 - offvalue 355
 - onvalue 355
 - orient 362
 - pack() 334
 - pack-Manager 333, 338
 - Radiobuttons 356
 - resizable() 339
 - root 334
 - rowspan 340
 - Schriftart 352
 - Scrollbalken 362
 - search() 353
 - side 335
 - Steuerelement 334
 - sticky 343
 - StringVar 349
 - tearoff 370
 - Textboxen 350
 - textvariable 349
 - Toplevel widget 334
 - Untermenüs 370
 - update() 344
 - Vaterfenster 334
 - Widget 334
 - widget-Optionen 344
 - winfo_height() 344
 - winfo_width() 344
 - word wrapping 352
 - xscrollcommand 363
 - xview() 363
 - yscrollcommand 363
 - yview() 363
 - Zellen verbinden 340
- True 34
- try 82, 263, 266
- Tupel 209
- type() 42, 214
- TypeError 53

U

- Überschreiben von Methoden 166
- Umwandlung von Datentypen 43
- Unbehandelte Ausnahme 264
- UnboundLocalError 106
- unhandled exception 264
- UnicodeDecodeError 306
- union() 250
- Unpacking 214
- Untermenge 251
- Unveränderliche Listen 209
- update() 250
- upper() 228

V

ValueError 82, 264
 values() 240
 Variable 22, 33
 Variablenname 38, 51
 Vererbung 163
 Vergleichen von Werten 60
 Vergleichsoperatoren 61
 Verschachtelte Schleifen 78, 92
 Verschachtelungstiefe 123
 Verschachtelung von Code-Blöcken 64
 Versionsverwaltung
 - Amend commit 439
 - Änderungen zuordnen 425
 - Änderungsverlauf 417
 - Annotate 425
 - Apply Stash 438
 - Benennung von Branches 431
 - Binärdateien 429
 - Branching 430
 - Branching-Modelle 440
 - Clone 406, 423
 - Commit 404, 413
 - detached HEAD 419
 - Drop (Stash) 438
 - Einchecken 404, 413
 - entferntes Repository 406
 - Entwicklungsbranch 441
 - Feature-Branch 441
 - Fetch 435
 - Git 405
 - GitHub 409
 - .gitignore 421
 - Hauptentwicklungslinie 430
 - Historie 417
 - Klonen 406, 422
 - Konflikte auflösen 429
 - lokale Sicherung 403

- master-Branch 430
 - Merge 425
 - origin 422
 - Pop Stash 438
 - push 406, 424
 - Remote Repository 406
 - Repository 404
 - Repository anlegen 410
 - Revert 416
 - Revision 417
 - SHA-Hash 418
 - Sperren von Dateien 404
 - Stashing 437
 - Tags 439
 - Unstash 438
 - Workflows 440
 - Zusammenarbeit im Team 406
 - Zweig 430
 Versteckte Dateien 314
 Verzeichnis 235
 Virtuelle Maschine 8
 Visual Studio Code 10

W

Wegwerfvariable 81
 while-Schleifen 71
 Whitespaces 227
 with 305
 Wörterbuch 235
 write() 310
 writelines() 311

Z

Zeichenkette 35, 216
 Zeilennummern 17, 19
 ZeroDivisionError 83
 Zufallszahlen 92
 Zuweisungsoperator 42