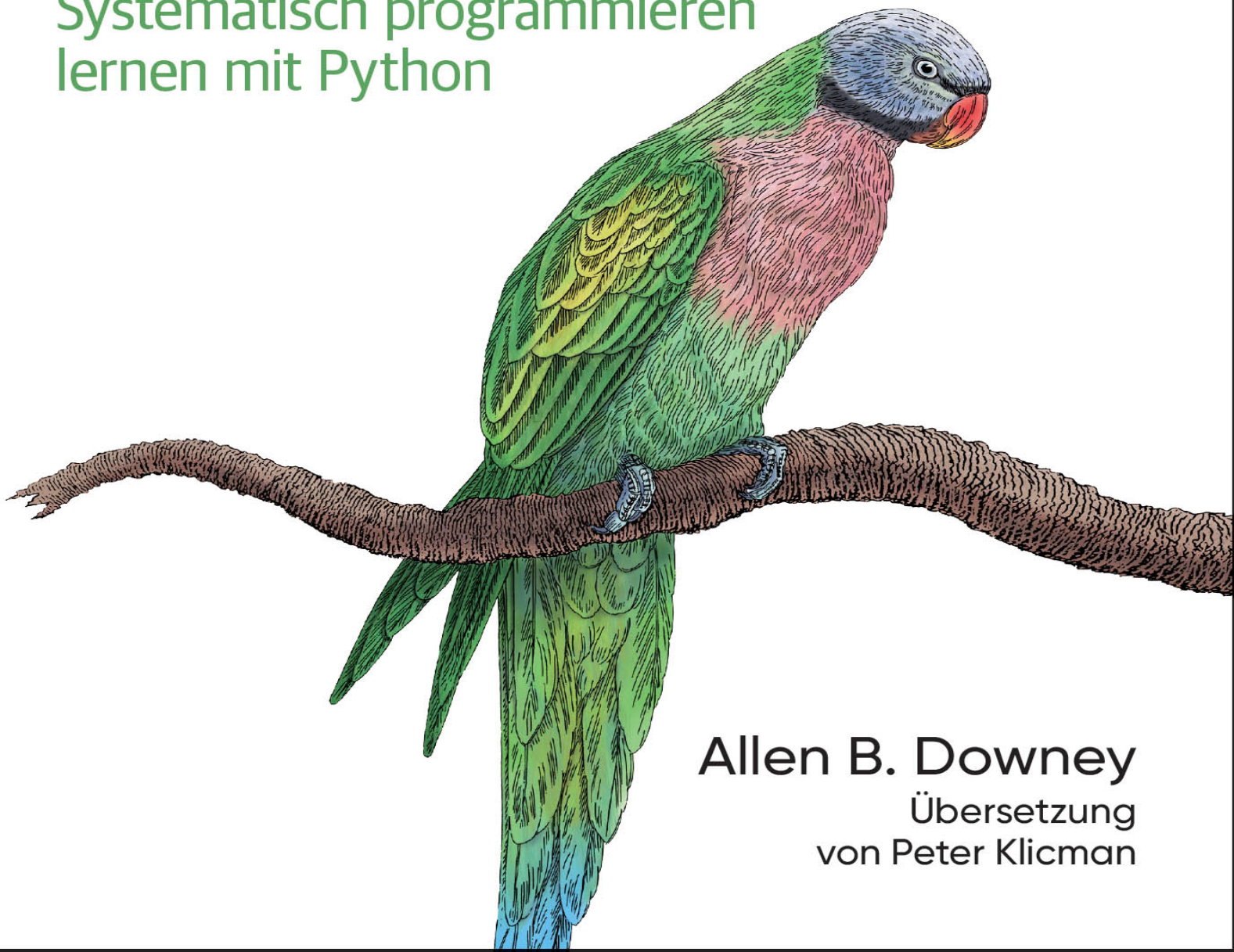
O'REILLY®

Aktuell zu Python 3

Think Python

Systematisch programmieren lernen mit Python



Allen B. Downey

Übersetzung von Peter Klicman

Think Python

Systematisch programmieren lernen mit Python

Allen B. Downey

Deutsche Übersetzung von Peter Klicman

O'REILLY®

Allen B. Downey

Lektorat: Alexandra Follenius
Übersetzung: Peter Klicman
Korrektorat: Claudia Lötschert, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, Michael Oréal, *www.oreal.de*

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

ISBN:
Print 978-3-96009-169-1
PDF 978-3-96010-507-7
ePub 978-3-96010-508-4
mobi 978-3-96010-509-1

1. Auflage

Wieblinger Weg 17
69123 Heidelberg

Authorized German translation of the English edition of *Think Python, 2nd Edition*, ISBN 9781491939369 © 2016 Allen Downey. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«. O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: *kommentar@oreilly.de*.

5 4 3 2 1 0

Inhalt

Vorwort

Die seltsame Geschichte dieses Buchs

Im Januar 1999 bereitete ich mich als Dozent auf einen Einführungskurs in die Java-Programmierung vor. Ich hatte den Kurs bereits dreimal gehalten, und so langsam frustrierte er mich. Die Durchfallquote in den Kursen war zu hoch, und selbst bei den erfolgreichen Studenten waren die Leistungen immer noch schwach.

Eines der Probleme bestand meiner Meinung nach in den Lehrbüchern: Sie waren zu dick, enthielten zu viele unnötige Einzelheiten über Java und zu wenige Informationen darüber, wie man programmiert. Und sie litten alle unter dem Falltüreffekt: Die Bücher fingen einfach an, steigerten sich allmählich, und irgendwo um Kapitel 5 herum kam dann der Einbruch. Die Studenten erhielten zu schnell zu viel neues Material und verbrachten den Rest des Semesters damit, die Einzelteile zusammenzusetzen.

Zwei Wochen vor dem ersten Kurstag entschied ich mich, ein eigenes Buch zu schreiben. Meine Ziele waren:

- So kurz wie möglich: Es ist einfacher, 10 statt 50 Seiten zu lesen.
- Bewusste Wortwahl: Ich habe versucht, den Fachjargon zu minimieren und jeden Begriff bei der

erstmaligen Verwendung zu definieren.

- Langsame Steigerung: Um Falltüren zu vermeiden, habe ich die schwierigen Themen in eine Reihe kleinerer Schritte aufgeteilt.
- Fokus auf der Programmierung, nicht auf der Programmiersprache: Ich habe den kleinstmöglichen nützlichen Ausschnitt aus Java erklärt und den Rest weggelassen.

Aus einer Laune heraus wählte ich als Titel *How to Think Like a Computer Scientist* (Wie Sie wie ein Informatiker denken).

Meine erste Fassung war holprig, aber sie funktionierte. Beim Lesen verstanden die Studenten genug, damit ich mich in der Unterrichtszeit auf die schwierigen und interessanten Themen konzentrieren konnte – und die Studenten Zeit zum Üben hatten. Schließlich veröffentlichte ich das Buch unter der *GNU Free Documentation License*, nach der die Nutzer das Buch kopieren, ändern und verteilen dürfen.

Und dann kam der spannende Teil: Jeff Elkner, ein Highschool-Lehrer in Virginia, nahm mein Buch und übersetzte es für Python. Er schickte mir eine Ausgabe seiner Übertragung, und ich machte die ungewöhnliche Erfahrung, Python zu lernen, indem ich mein eigenes Buch las. Unter dem Verlagsnamen *Green Tea Press* veröffentlichte ich die erste Python-Version im Jahr 2001.

2003 begann ich dann, am Olin College zu unterrichten, und gab auch zum ersten Mal Kurse in Python. Der Unterschied zu den Java-Kursen war offensichtlich: Die Studenten hatten weniger zu kämpfen, lernten mehr, arbeiteten an interessanteren Projekten und hatten insgesamt eine Menge mehr Spaß.

In den darauffolgenden Jahren habe ich das Buch weiterentwickelt, Fehler beseitigt, die Beispiele verbessert und zusätzliches Material eingefügt, vor allem neue Übungen.

Das Ergebnis war das 2012 bei O'Reilly Media veröffentlichte Buch mit dem etwas weniger bombastischen Titel *Think Python*. Unter anderem hat sich Folgendes gegenüber der Green-Tea-Press-Version geändert:

- Am Ende jedes Kapitels habe ich einen Abschnitt zum Thema Debugging eingefügt. Diese Abschnitte enthalten allgemeine Techniken zum Aufspüren und Vermeiden von Bugs sowie Warnungen vor entsprechenden Stolpersteinen in Python.
- Ich habe zusätzliche Übungen eingefügt – von kurzen Verständnistests bis hin zu grundlegenden Projekten. Und für die meisten habe ich Lösungen geschrieben.
- Außerdem gibt es Fallstudien – längere Beispiele mit Übungen, Lösungen und Erläuterungen.
- Die Darstellung von Entwicklungsplänen und grundlegenden Entwurfsmustern habe ich erweitert.
- Ich habe Kapitel zum Thema Debugging und der Analyse von Algorithmen eingefügt.

Die zweite Auflage von *Think Python* umfasst nun außerdem die folgenden Neuerungen:

- Das Buch und der gesamte Code wurden an Python 3 angepasst.
- Ich habe einige Abschnitte eingefügt (und zusätzliche Hinweise im Web), die Einsteigern dabei helfen, Python im Browser auszuführen. Sie müssen sich also nicht mit der Installation von Python herumschlagen, wenn Sie das nicht wollen.

- Für den Abschnitt »Das turtle-Modul« auf Seite 51 bin ich von meinem eigenen Turtle-Paket auf ein Standard-Python-Modul namens `turtle` umgestiegen, das sich einfacher installieren lässt und auch leistungsfähiger ist.
- Ich habe das Kapitel 19, »Weitere nützliche Python-Features«, ergänzt, in dem Funktionalitäten vorgestellt werden, die nicht unbedingt notwendig, aber manchmal recht praktisch sind.

Ich hoffe, dass Ihnen die Arbeit mit diesem Buch Spaß macht und es Ihnen dabei hilft, zu lernen, wie Sie wie ein Informatiker programmieren und vielleicht auch ein bisschen so denken.

Allen B. Downey

Typografische Konventionen

In diesem Buch werden die folgenden typografischen Konventionen verwendet:

Kursivschrift
: Wird für URLs, E-Mail-Adressen, Dateinamen, Dateiendungen, Pfadnamen und Verzeichnisse verwendet.

Fettschrift
: Wird zum Hervorheben genutzt und um die erste Verwendung eines Begriffs zu kennzeichnen.

`Nichtproportionalschrift`
: Wird für Befehle oder anderen Text, den Sie wortwörtlich eingeben müssen, sowie für Befehlsausgaben verwendet.

Nutzung der Codebeispiele

Die Beispiele und die Lösungen zu den Übungen in diesem Buch stehen zum Download zur Verfügung. Sie finden sie auf unserer Verlagswebsite: *https://oreilly.de/9783960091691*

Dieses Buch soll Ihnen bei der Arbeit helfen. Es ist grundsätzlich erlaubt, den Code dieses Buchs in Ihren Programmen und der Dokumentation zu verwenden. Hierfür ist es nicht notwendig, uns um Erlaubnis zu fragen, es sei denn, es handelt sich um eine größere Menge Code. So ist es beim Schreiben eines Programms, das einige Codeschnipsel dieses Buchs verwendet, nicht nötig, sich mit uns in Verbindung zu setzen; beim Verkauf oder Vertrieb einer CD-ROM mit Beispielen aus O'Reilly-Büchern dagegen schon. Das Beantworten einer Frage durch Zitieren von Beispielcode erfordert keine Erlaubnis. Verwenden Sie einen erheblichen Teil des Beispielcodes aus diesem Buch in Ihrer Dokumentation, ist jedoch unsere Erlaubnis nötig.

Eine Quellenangabe ist zwar erwünscht, aber nicht unbedingt notwendig. Hierzu gehört in der Regel die Erwähnung von Titel, Autor, Verlag und ISBN. Zum Beispiel: »*Think Python* von Allen B. Downey (O'Reilly). Copyright 2021 dpunkt.verlag, ISBN 978-3-96009-169-1«.

Falls Sie nicht sicher sind, ob Ihre Nutzung der Codebeispiele über die hier erteilte Genehmigung hinausgeht, nehmen Sie bitte unter der Adresse *permissions@oreilly.com* Kontakt mit uns auf.

Danksagungen

Herzlichen Dank an Jeff Elkner, der mein Java-Buch auf Python übertrug, dieses Projekt auf den Weg gebracht und mich mit dem vertraut gemacht hat, was sich als meine Lieblingssprache entpuppen sollte.

Vielen Dank auch an Chris Meyers für mehrere Abschnitte in *How to Think Like a Computer Scientist*.

Ich danke der Free Software Foundation für die Entwicklung der GNU Free Documentation License, die mir die Zusammenarbeit mit Jeff und Chris erleichtert hat, sowie Creative Commons für die Lizenz, die ich jetzt nutze.

Vielen Dank außerdem an die Lektorinnen und Lektoren bei Lulu, die an *How to Think Like a Computer Scientist* gearbeitet haben.

Vielen Dank auch an die Lektorinnen und Lektoren von O'Reilly Media, die an *Think Python* gearbeitet haben.

Mein herzlicher Dank gilt außerdem allen Studenten, die mit früheren Versionen dieses Buchs gearbeitet haben, sowie allen Beitragenden für ihre Korrekturen und Vorschläge.

KAPITEL 1

Programme entwickeln

Das Ziel dieses Buchs ist es, Ihnen beizubringen, wie ein Informatiker zu denken. Diese Denkweise kombiniert einige der besten Eigenschaften aus Mathematik, Ingenieurswesen und Naturwissenschaft. Wie Mathematiker verwenden Informatiker formale Sprachen, um Ideen symbolisch darzustellen (genauer gesagt, Berechnungen). Ähnlich wie Ingenieure entwerfen Informatiker Dinge, setzen Komponenten zu Systemen zusammen und suchen einen Kompromiss aus mehreren Alternativen. Und wie Wissenschaftler beobachten sie komplexe Systeme, entwickeln Hypothesen und testen Prognosen.

Die allerwichtigste Fähigkeit eines Informatikers besteht darin, **Probleme zu lösen**. Mit Problemlösung ist die Fähigkeit gemeint, Probleme zu formulieren, kreativ über Lösungen nachzudenken und eine Lösung klar und präzise auszudrücken. Dabei zeigt sich, dass Programmieren zu lernen eine ausgezeichnete Gelegenheit ist, Ihre Problemlösungsfähigkeiten zu trainieren. Deshalb heißt dieses Kapitel auch »Programme entwickeln«.

Auf einer Ebene werden Sie das Programmieren lernen – was an sich schon eine nützliche Fähigkeit ist. Auf einer anderen Ebene werden Sie die Programmierung als Mittel

zum Zweck kennenlernen. Und im weiteren Verlauf dieses Buchs wird dieser Zweck immer klarer werden.

Was ist ein Programm?

Ein **Programm** ist eine Folge von Anweisungen, die bestimmen, wie eine Berechnung durchgeführt wird. Eine solche Berechnung kann etwas Mathematisches sein wie etwa die Lösung eines Gleichungssystems oder die Bestimmung der Wurzeln eines Polynoms. Es kann sich aber auch um eine symbolische Berechnung handeln, wenn Sie beispielsweise Text in einem Dokument suchen und ersetzen oder ein Programm kompilieren (seltsam, oder?).

Die Details sehen natürlich in jeder Programmiersprache anders aus, aber einige grundlegende Anweisungen gibt es in so ziemlich jeder Sprache:

Eingabe
: Daten von der Tastatur, einer Datei oder einem Gerät abrufen

Ausgabe
: Daten auf dem Bildschirm anzeigen oder an eine Datei bzw. ein Gerät senden

Mathematische Anweisungen
: Grundlegende mathematische Berechnungen (wie etwa Addition und Multiplikation) ausführen

Bedingte Ausführung
: Bestimmte Bedingungen prüfen und den entsprechenden Code ausführen

Wiederholung
: Aktionen wiederholt ausführen, meistens in einer bestimmten Variation

Ob Sie es glauben oder nicht: Das ist auch schon so ziemlich alles. Jedes Programm, das Sie jemals benutzt haben – unabhängig davon, wie kompliziert es ist –, besteht aus solchen Anweisungen. Insofern können Sie sich die Programmierung als den Vorgang vorstellen, komplizierte Aufgaben in immer kleinere Teilaufgaben zu zerlegen, bis diese einfach genug sind, um sie durch eine dieser grundlegenden Anweisungen zu erledigen.

Python ausführen

Eine Herausforderung beim Einstieg in Python besteht darin, dass Sie möglicherweise Python und die dazugehörige Software auf Ihrem Rechner installieren müssen. Wenn Sie mit Ihrem Betriebssystem und insbesondere mit der Kommandozeile vertraut sind, werden Sie mit der Installation von Python keine Schwierigkeiten haben. Doch für den Einsteiger kann es mühsam sein, Systemadministration und Programmierung gleichzeitig lernen zu müssen.

Um dieses Problem zu vermeiden, empfehle ich Ihnen, Python im Browser auszuführen. Später, wenn Sie mit Python vertraut sind, gebe ich einige Hinweise zur Installation von Python auf Ihrem Computer.

Es gibt eine Reihe von Webseiten, über die Sie Python ausführen können. Falls Sie bereits einen Favoriten haben, nur zu. Anderenfalls empfehle ich PythonAnywhere. Detaillierte englischsprachige Anweisungen für den Einstieg erhalten Sie unter *http://tinyurl.com/thinkpython2e*.

Dieses Buch ist für die Version Python 3 geschrieben, Python 2 wird seit April 2020 nicht mehr unterstützt. Wir haben einige wenige Hinweise zu Unterschieden zwischen

den Versionen im Buch beibehalten, damit Sie informiert sind, falls Sie einmal auf Python-2-Code treffen.

Der Python-**Interpreter** ist ein Programm, das Python-Code liest und ausführt. Abhängig von Ihrer Umgebung starten Sie den Interpreter durch einen Klick auf ein Icon oder indem Sie `python` in der Kommandozeile eingeben. Sobald Sie ihn starten, sehen Sie in etwa Folgendes:

```
Python 3.8.0 (default, Jun 19 2020, 14:20:21)

[GCC 10.1] on linux

Type "help", "copyright", "credits" or "license" for more
information.

>>>
```

Die ersten drei Zeilen enthalten Informationen über den Interpreter und das Betriebssystem, unter dem er läuft. Die Ausgabe kann bei Ihnen daher anders aussehen, doch Sie sollten darauf achten, dass die Versionsnummer (hier `3.8.0`) mit einer 3 beginnt, was bedeutet, dass Python 3 ausgeführt wird. Beginnt sie stattdessen mit einer 2, läuft (Sie ahnen es) Python 2.

Die letzte Zeile ist die sogenannte **Eingabeaufforderung** (auch Prompt genannt, `>>>`), die anzeigt, dass der Interpreter bereit ist, Ihren Code zu verarbeiten. Geben Sie eine Zeile Code ein, gibt der Interpreter das Ergebnis aus:

```
>>> 1 + 1

2
```

Sie sind nun bereit, zu beginnen. Von hier an gehe ich davon aus, dass Sie wissen, wie man den Python-Interpreter startet und Code ausführt.

Das erste Programm

Traditionell heißt das erste Programm, das Sie in einer neuen Sprache schreiben, »Hallo, Welt!« – weil es einfach nur die Worte »Hallo, Welt!« ausgibt. In Python sieht das folgendermaßen aus:

```
>>> print('Hallo, Welt!')
```

Das ist ein Beispiel für eine `print`**-Anweisung**, die in Wahrheit natürlich nichts »druckt«. Sie zeigt den Wert einfach auf dem Bildschirm an. In diesem Fall lautet das Ergebnis

```
Hallo, Welt!
```

Die Apostrophe in der Programmanweisung kennzeichnen den Anfang und das Ende des anzuzeigenden Texts und erscheinen nicht im Ergebnis.

Die Klammern zeigen an, dass `print` eine Funktion ist. Wir gehen auf Funktionen in Kapitel 3 ein.

In Python 2 sieht die print-Anweisung etwas anders aus. Sie ist keine Funktion und verwendet daher keine Klammern.

```
>>> print 'Hello, World!'
```

Arithmetische Operatoren

Nach »Hallo, Welt« ist die Arithmetik der nächste Schritt. Python stellt **Operatoren** bereit, d. h. spezielle Symbole, die Berechnungen wie Addition und Multiplikation repräsentieren.

Die Operatoren +, - und * führen die Addition, Subtraktion und Multiplikation aus. Hier ein Beispiel:

```
>>> 40 + 2

42

>>> 43 - 1

42

>>> 6 * 7

42
```

Der Operator / führt die Division aus:

```
>>> 84 / 2

42.0
```

Sie fragen sich vielleicht, warum das Ergebnis `42.0` lautet und nicht einfach `42`. Das erkläre ich im nächsten Abschnitt.

Zu guter Letzt führt der Operator ** die Potenzierung durch:

```
>>> 6**2 + 6

42
```

In einigen anderen Sprachen wird ^ für die Potenzierung genutzt, doch bei Python ist das der bitweise Operator XOR. Wenn Sie mit bitweisen Operatoren nicht vertraut sind, wird Sie das Ergebnis überraschen:

```
>>> 6 ^ 2
```

Ich gehe in diesem Buch nicht weiter auf bitweise Operatoren ein, doch Sie können unter *http://wiki.python.org/moin/BitwiseOperators* mehr über sie erfahren.

Werte und Typen

Ein **Wert** ist eines der grundlegenden Dinge, mit denen ein Programm arbeitet, etwa ein Buchstabe oder eine Zahl. Einige Werte, die wir bisher gesehen haben, sind `2`, `42.0` und `'Hallo, Welt!'`.

Diese Werte gehören zu verschiedenen **Typen**: `2` ist eine **ganze Zahl (Integer)**, `42.0` ist eine **Fließkommazahl**, und `'Hallo, Welt!'` ist eine **Zeichenkette (String)**, die man so nennt, weil die Buchstaben »aneinandergekettet« sind.

Wenn Sie nicht sicher sind, welchen Typ ein Wert hat, kann Ihnen der Interpreter Auskunft geben:

```
>>> type(2)

<class 'int'>

>>> type(42.0)

<class 'float'>

>>> type('Hello, World!')

<class 'str'>
```

Bei diesen Ergebnissen wird das Wort »class« (also »Klasse«) im Sinne einer Kategorie verwendet. Ein Typ ist eine Wertekategorie.

Ganze Zahlen, also Integerwerte, gehören (wenig überraschend) zum Typ `int`, Strings zu `str` und Fließkommazahlen zu `float`.

Was ist mit Werten wie `'2'` und `'42.0'`? Sie sehen wie Zahlen aus, stehen aber wie Strings zwischen Anführungszeichen:

```
>>> type('2')

<class 'str'>

>>> type('42.0')

<class 'str'>
```

Sie sind Strings.

Die (englische) Schreibweise mit Kommata bei einem großen Integerwert, z. B. `1,000,000`, ist in Python kein gültiger *Integerwert*, aber dennoch legal:

```
>>> 1,000,000

(1, 0, 0)
```

Das haben Sie sicher nicht erwartet! Python interpretiert `1,000,000` als eine durch Kommata getrennte Sequenz (Folge) von Integerwerten. Sie erfahren später mehr über diese Art von Sequenzen.

Formale und natürliche Sprachen

Natürliche Sprachen sind jene Sprachen, die Menschen sprechen, wie etwa Deutsch, Englisch, Spanisch und Französisch. Diese Sprachen wurden nicht von Menschen entworfen (obwohl wir Menschen versuchen, eine gewisse

Ordnung hineinzubringen), sondern haben sich natürlich entwickelt.

Formale Sprachen sind dagegen Sprachen, die von Menschen für bestimmte Anwendungen entworfen wurden. So ist beispielsweise die Notation der Mathematiker eine formale Sprache, die besonders gut dafür geeignet ist, die Beziehungen zwischen Zahlen und Symbolen darzustellen. Chemiker verwenden eine formale Sprache, um die chemische Struktur von Molekülen abzubilden. Und natürlich das Wichtigste:

> **Programmiersprachen sind formale Sprachen, die entwickelt wurden, um Berechnungen auszudrücken.**

Formale Sprachen haben eher strenge Syntaxregeln. Beispielsweise ist 3 + 3 = 6 ein syntaktisch korrekter mathematischer Ausdruck, 3 + = 3$6 dagegen nicht. H_2O ist eine syntaktisch korrekte chemische Formel, ${}_2Zz$ dagegen nicht.

Es gibt zweierlei Syntaxregeln: Die einen regeln **Tokens** und die anderen die Struktur. Tokens sind die grundlegenden Elemente einer Sprache, wie etwa Wörter, Zahlen oder chemische Elemente. Eines der Probleme an 3 + = 3$6 besteht darin, dass $ kein zulässiges Token in der Mathematik ist (zumindest nach meinem Kenntnisstand nicht). Auf ähnliche Weise ist ${}_2Zz$ als chemische Formel nicht zulässig, weil es kein Element mit der Abkürzung Zz gibt.

Die zweite Art von Syntaxfehlern bezieht sich auf die **Struktur** einer Anweisung, also auf die Art und Weise, in der Tokens arrangiert sind. Die Anweisung 3 + = 3 ist nicht zulässig, weil $+$ und $=$ zwar legale Tokens sind, aber nicht unmittelbar hintereinanderstehen dürfen. Auf ähnliche Weise kommt in einer chemischen Formel der Index nach dem Elementnamen, nicht davor.

Dies ist ein s@uber strukturierter deut$scher Satz mit ungültigen T*kens. Dieser Satz nur gültige Tokens hat, aber Struktur ungültig ist.

Wenn Sie einen Satz im Deutschen lesen, oder eine Anweisung in einer formalen Sprache, müssen Sie dessen Struktur verstehen (auch wenn Sie das bei einer natürlichen Sprache unterbewusst machen). Diesen Prozess nennt man **parsing**.

Obwohl formale und natürliche Sprachen viele Merkmale gemeinsam haben – Tokens, Struktur und Semantik –, gibt es jedoch auch einige Unterschiede:

Mehrdeutigkeit
: Natürliche Sprachen sind voller Mehrdeutigkeiten, mit denen wir Menschen anhand von Kontext und anderen Informationen gut umgehen können. In formalen Sprachen gibt es fast keine oder überhaupt keine Mehrdeutigkeiten. Insofern hat jede Anweisung unabhängig vom Kontext genau eine Bedeutung.

Redundanz
: Um die Mehrdeutigkeiten wieder wettzumachen und die Gefahr von Missverständnissen zu minimieren, gibt es eine Menge Redundanzen in natürlichen Sprachen. Dadurch sind sie oft sehr wortreich. Formale Sprachen dagegen sind weniger redundant und prägnanter.

Sprichwörtlichkeit
: Natürliche Sprachen sind voller Idiome (Redewendungen) und Metaphern. Bei dem Ausspruch »Der Groschen ist gefallen!« gibt es wahrscheinlich weder einen Groschen, noch fällt etwas herunter (dieses Idiom bedeutet einfach, dass jemand nach längerer Verwirrung endlich etwas verstanden hat).

Formale Sprachen dagegen bedeuten exakt das, was sie ausdrücken.

Menschen, die mit einer natürlichen Sprache aufwachsen (also jeder), haben oft Schwierigkeiten, sich an formale Sprachen zu gewöhnen. In gewisser Weise ist der Unterschied zwischen einer formalen und einer natürlichen Sprache wie der Unterschied zwischen Poesie und Prosa:

Poesie
: Wörter werden sowohl aufgrund ihres Klangs als auch ihrer Bedeutung eingesetzt, und das Gedicht insgesamt zielt auf einen Effekt oder eine emotionale Reaktion ab. Mehrdeutigkeiten sind nicht nur häufig, sondern oftmals beabsichtigt.

Prosa
: Die wörtliche Bedeutung der Wörter ist wichtiger, die Struktur trägt zusätzlich zur Bedeutung bei. Prosa ist für eine Analyse zugänglicher als Poesie, aber trotzdem oft mehrdeutig.

Programme
: Die Bedeutung eines Computerprogramms ist eindeutig und wortwörtlich. Sie kann durch Analyse der Tokens und der Struktur vollständig erfasst werden.

Hier einige Vorschläge für das Lesen von Programmen (und anderen formalen Sprachen): Erstens sollten Sie nicht vergessen, dass formale Sprachen wesentlich dichter als natürliche Sprachen sind und dass es daher länger dauert, sie zu lesen. Außerdem spielt die Struktur eine entscheidende Rolle. Deshalb ist es üblicherweise keine sonderlich gute Idee, von oben nach unten und von links nach rechts zu lesen. Stattdessen sollten Sie lernen, das Programm in Ihrem Kopf zu »parsen«, wobei Sie die Tokens erkennen und die Struktur interpretieren. Und letztendlich