mitp

Brett
Slatkin

2. Auflage

Effektiv Python programmieren

90 Wege für bessere Python-Programme

Inhaltsverzeichnis

Einleitung

Die Programmiersprache Python besitzt einzigartige Stärken und Vorteile, die nicht immer ganz einfach zu verstehen sind. Mit anderen Sprachen vertraute Programmierer nähern sich Python oftmals nur zögerlich, anstatt die enorme Ausdrucksfähigkeit willkommen zu heißen. Andere übertreiben in entgegengesetzter Richtung und überstrapazieren Python-Funktionalitäten, was später große Probleme verursachen kann.

Dieses Buch möchte Ihnen einen Einblick geben in das, was im Englischen als *pythonic* (»Python-artig«) bezeichnet wird: die beste Art und Weise, Python zu verwenden. Dieser Programmierstil setzt grundlegende Kenntnisse der Sprache voraus, über die Sie, wie ich annehme, bereits verfügen. Programmieranfänger lernen die bewährten Vorgehensweisen zur Nutzung von Pythons Fähigkeiten kennen. Erfahrene Programmierer hingegen erfahren, wie sie souverän mit den Eigenheiten dieses neuen Werkzeugs zurechtkommen.

Ich möchte Sie darauf vorbereiten, mit Python große Wirkung zu erzielen.

Inhalt des Buchs

In den einzelnen Kapiteln des Buchs sind verschiedene Punkte aufgeführt, die jeweils weit gefasste, aber doch verwandte Themen betreffen. Es steht Ihnen frei, ganz nach Ihrem Interesse im Buch herumzublättern. Jeder Punkt enthält kompakte und präzise Anleitungen, die Ihnen dabei helfen sollen, effektivere Python-Programme zu schreiben. Sie finden dort Ratschläge, was zu tun und was besser zu lassen ist, Hinweise zum Eingehen von Kompromissen und Erklärungen, warum das eine oder das andere die bessere Wahl ist. Es gibt Querverweise, die Ihnen beim Lesen das Verständnis erleichtern sollen.

Die zweite Ausgabe dieses Buchs konzentriert sich vollständig auf Python 3 bis einschließlich Version 3.8 (siehe Punkt 1: *Kenntnis der Python-Version*). Die meisten Punkte der ersten Ausgabe wurden überarbeitet und sind nach wie vor vorhanden, bei vielen hat es jedoch beträchtliche Aktualisierungen gegeben. Bei einigen Punkten hat sich mein Rat im Vergleich zur ersten Ausgabe aufgrund der Best Practices, die sich bei der Weiterentwicklung von Python ergeben haben, sogar völlig verändert. Sollten Sie trotz der endgültigen Abkündigung zum 1. Januar 2020 noch immer vornehmlich Python 2 verwenden, dürfte die erste Ausgabe dieses Buchs für Sie nützlicher sein.

Python verfolgt hinsichtlich der Standardbibliothek die Philosophie, dass sozusagen die Batterien im Lieferumfang enthalten sein sollten. Viele andere Sprachen werden mit nur einigen wenigen gängigen Paketen ausgeliefert und Sie müssen sich selbst auf die Suche begeben, wenn Sie zusätzliche wichtige Funktionalitäten benötigen. Allerdings sind viele der Standardmodule so eng mit den typischen Spracheigenheiten Pythons verflochten, dass sie sehr wohl auch Bestandteil der Sprachspezifikation sein könnten. Viele dieser Pakete sind so eng mit typischem Python-Code verwoben, dass sie eigentlich auch Teil des Sprachumfangs sein könnten. Die Anzahl der Standardmodule ist zu groß, um sie in diesem Buch behandeln zu können. Dennoch habe ich diejenigen aufgenommen, deren Kenntnis und Nutzung ich für unverzichtbar halte.

Kapitel 1: Pythons Sicht der Dinge

In der Python-Community dient das englische Adjektiv *pythonic* zur Beschreibung von Code, der einem bestimmten Programmierstil folgt. Dieser Stil hat sich im Laufe der Zeit durch Erfahrung im Umgang mit der Sprache und die Zusammenarbeit der Community allmählich entwickelt. Dieses Kapitel beschreibt die beste Art und Weise, die gängigsten Aufgaben in Python zu erledigen.

Kapitel 2: Listen und Dictionaries

In Python ist die häufigste Methode zur Organisation von Informationen das Speichern einer Reihe von Werten in einer Liste. Die naheliegende Ergänzung dazu ist das Dictionary, das den Werten zugeordnete Schlüssel speichert. Dieses Kapitel beschreibt, wie sich Programme mit diesen vielfältigen Bausteinen erstellen lassen.

Kapitel 3: Funktionen

In Python besitzen Funktionen eine Vielzahl von Merkmalen, die einem Programmierer das Leben erleichtern. Einige sind den Fähigkeiten anderer Programmiersprachen ähnlich, viele gibt es jedoch nur in Python. Dieses Kapitel hat zum Thema, wie man Funktionen dazu verwendet, ihren Zweck zu verdeutlichen, ihre Wiederverwendung zu ermöglichen und Bugs zu vermeiden.

Kapitel 4: Listen-Abstraktionen und Generatoren

In Python gibt es eine spezielle Syntax, um Listen, Dictionaries und Sets schnell durchlaufen zu können und davon abgeleitete Datenstrukturen zu erzeugen. Ebenso ist es möglich, dass eine Funktion die durchlaufenen Werte der Reihe nach zurückgibt. Dieses Kapitel erläutert, wie diese Features eine bessere Leistung erbringen können, weniger Speicher benötigen und besser verständlich eingesetzt werden.

Kapitel 5: Klassen und Schnittstellen

Python ist eine objektorientierte Programmiersprache. Um Aufgaben in Python zu erledigen, ist es oft erforderlich, neue Klassen anzulegen und zu definieren, wie sie mittels ihrer Schnittstellen und der Klassenhierarchie interagieren. Dieses Kapitel erläutert, wie man Klassen zur Modellierung des beabsichtigten Verhaltens durch Objekte nutzt.

Kapitel 6: Metaklassen und Attribute

Metaklassen und dynamische Attribute sind leistungsfähige Python-Merkmale, die es allerdings auch ermöglichen, äußerst merkwürdiges und unerwartetes Verhalten zu implementieren. Dieses Kapitel stellt die üblichen Verwendungsweisen dieser Mechanismen vor, damit Sie dem *Prinzip der geringsten Überraschung* folgen können.

Kapitel 7: Nebenläufigkeit und parallele Ausführung

In Python können schnell und einfach Programme geschrieben werden, die scheinbar mehrere Aufgaben gleichzeitig erledigen. Python kann außerdem durch Systemaufrufe, Subprozesse oder C-Erweiterungen Code parallel ausführen. Dieses Kapitel führt vor, wie sich diese leicht unterschiedlichen Verfahren am besten einsetzen lassen.

Kapitel 8: Robustheit und Performance

Python bringt eine Reihe integrierter Features und Module mit, die dabei helfen, stabile und verlässliche Programme zu erstellen. Darüber hinaus bietet Python Tools, die es ermöglichen, mit minimalem Aufwand eine höhere Leistung zu erzielen. Dieses Kapitel beschreibt, wie Sie Python nutzen können, um die Stabilität und Effizienz Ihrer Programme zu maximieren.

Kapitel 9: Testen und Debuggen

In welcher Sprache Sie programmieren, spielt keine Rolle; Sie sollten Ihren Code stets testen. Pythons dynamische Features können allerdings das Risiko erhöhen, dass es zu Laufzeitfehlern kommt. Erfreulicherweise vereinfachen sie es jedoch auch, Tests zu schreiben und nicht richtig funktionierende Programme zu untersuchen. Dieses Kapitel befasst sich mit Pythons integrierten Werkzeugen zum Testen und Debuggen.

Kapitel 10: Zusammenarbeit

Bei der gemeinsamen Entwicklung von Python-Programmen muss dem Programmierstil Beachtung geschenkt werden. Selbst wenn Sie der einzige Programmierer sind, müssen Sie doch wissen, wie die von anderen Entwicklern erstellten Module

verwendet werden. Dieses Kapitel behandelt die Standardwerkzeuge und bewährte Vorgehensweisen, die es ermöglichen, gemeinsam an Python-Programmen zu arbeiten.

Konventionen dieses Buchs

Die Codebeispiele in diesem Buch sind in einer `nicht-proportionalen` Schrift gedruckt. Wenn lange Zeilen umbrochen werden müssen, verwende ich zur Markierung das Zeichen »\«. Weggelassene Codeabschnitte sind durch Auslassungszeichen (`...`) gekennzeichnet. Sie weisen darauf hin, dass weiterer Code vorhanden ist, der im gegebenen Zusammenhang jedoch nicht von Bedeutung ist. Sie müssen sich den vollständigen Beispielcode herunterladen (siehe unten), um die gekürzten Codeschnipsel ausführen zu können.

Ich habe mir hinsichtlich der üblichen Python-Stilregeln eine gewisse »künstlerische Freiheit« genommen, um die Darstellung an die Gegebenheiten eines Buchs anzupassen oder die wichtigsten Teile hervorzuheben. Außerdem habe ich aus Platzgründen die Docstrings entfernt. Das sollten Sie in Ihren eigenen Projekten allerdings nicht tun – folgen Sie besser den üblichen Stilregeln (siehe Punkt 2: *Stilregeln gemäß PEP 8*) und dokumentieren Sie Ihren Code (siehe Punkt 84: *Docstrings für sämtliche Funktionen, Klassen und Module*).

Viele Codeabschnitte enthalten auch die bei der Ausführung des Codes erfolgenden Ausgaben. Damit sind die Ausgaben auf der Konsole oder im Terminalfenster gemeint, die erscheinen, wenn das Programm im interaktiven Python-Interpreter ausgeführt wird. Vor den in `nicht-proportionaler` Schrift gedruckten Ausgaben steht eine Zeile mit den Zeichen >>> (die interaktive Python-Eingabeaufforderung). Dem liegt der Gedanke zugrunde, dass Sie die Beispiele in einer Python-Shell eingeben und die Ausgaben reproduzieren können.

Schließlich gibt es noch weitere Abschnitte in `nicht-proportionaler` Schrift ohne führende >>>-Zeile. Sie stellen Ausgaben des laufenden Programms dar (nicht des Interpreters). Am Anfang der Beispiele steht oftmals ein Dollarzeichen ($), das darauf hinweist, dass ich Programme von einer Shell wie Bash aus starte. Falls Sie die Befehle unter Windows oder einem anderen System ausführen, müssen Sie die Programmnamen und Argumente womöglich entsprechend anpassen.

Beispielcode herunterladen

Manche Beispiele in diesem Buch sollten besser als komplette Programme ohne die eingestreuten Texte betrachtet werden. Sie können sich den Quellcode auf der englischen Website zum Buch (`http://www.effectivepython.com`) herunterladen. Sie haben dann auch die Möglichkeit, mit dem Code herumzuexperimentieren und die Funktionsweise besser zu verstehen.

Über den Autor

Brett Slatkin ist bei Google als Führungskraft in der Softwareentwicklung tätig. Er ist leitender Ingenieur und Mitbegründer des Projekts Google Surveys, ist Miterfinder des PubSubHubbub-Protokolls und war an der Entwicklung von Googles erstem Cloud-Computing-Produkt (App Engine) beteiligt. Vor 14 Jahren sammelte er bei der Verwaltung von Googles riesigem Serverbestand erste Erfahrungen mit Python.

Neben der alltäglichen Arbeit spielt er gern Klavier und surft (beides mehr schlecht als recht). Auf seiner privaten Website (`http://www.onebigfluke.com`) schreibt er über Programmierung und damit verwandte Themen. Er erwarb seinen Bachelor of Science in technischer Informatik an der Columbia-Universität in New York und lebt in San Francisco.

Danksagungen

Ohne die Hilfe, Unterstützung und den Zuspruch vieler Menschen würde es dieses Buch nicht geben.

Dank an Scott Meyers für die Buchreihe *Effective Software Development*. Als ich 15 Jahre alt war, habe ich erstmals *Effective C++* gelesen und es war um mich geschehen. Es besteht kein Zweifel daran, dass Scotts Bücher zu meiner akademischen Laufbahn und meiner ersten Anstellung geführt haben. Ich bin begeistert, dass ich die Möglichkeit hatte, dieses Buch zu schreiben.

Dank an die technischen Korrekturleser für ihre ausführlichen und gründlichen Rückmeldungen zur zweiten Ausgabe dieses Buchs: Andy Chu, Nick Cohron, Andrew Dolan, Asher Mancinelli und Alex Martelli. Dank an meine Kollegen bei Google für die Durchsicht des Buchs. Ohne eure Hilfe wäre dieses Buch nicht vorstellbar.

Dank an alle Beteiligten beim amerikanischen Originalverlag Pearsons, die die zweite Ausgabe dieses Buchs Wirklichkeit werden ließen. Dank an meine Lektorin Debra Williams für unaufhörliche Unterstützung. Dank an das hilfreiche Team: Herausgeber Chris Zahn, Marketingleiter Stephane Nakib, Redakteurin Catherine Wilson, Projektleiterin Lori Lyons und Cover-Designer Chuti Prasertsith.

Dank an alle, die mich bei der ersten Ausgabe dieses Buchs unterstützt haben: Trina MacDonald, Brett Cannon, Tavis Rudd, Mike Taylor, Leah Culver, Adrian Holovaty, Michael Levine, Marzia Niccolai, Ade Oshineye, Katrina Sostek, Tom Cirtin, Chris Zahn, Olivia Basegio, Stephane Nakib, Stephanie Geels, Julie Nahil und Toshiaki Kurokawa. Dank an alle Leser, die Fehler und Raum für Verbesserungen aufzeigten. Dank an alle Übersetzer, die dieses Buch in anderen Sprachen verfügbar gemacht haben.

Dank an die wunderbaren Python-Programmierer, mit denen ich zusammenarbeiten durfte: Anthony Baxter, Brett Cannon, Wesley Chun, Jeremy Hylton, Alex Martelli, Neal Norwitz, Guido van Rossum, Andy Smith, Greg Stein und Ka-Ping Yee. Ich weiß eure Anleitungen und Hinweise wirklich zu schätzen. Die Python-Community ist großartig und ich bin dankbar, zu ihr zu gehören.

Dank an meine Teamkollegen, die seit all den Jahren erdulden müssen, dass ich das schwächste Glied in der Kette bin. Dank an Kevin Gibbs, der mich ermutigte, Risiken einzugehen. Dank an Ken Ashcraft, Ryan Barret und Jon McAlister, die mir zeigten, wie man's macht. Dank an Brad Fitzpatrick, der alles auf Vordermann gebracht hat. Dank an Paul McDonald, der unser verrücktes Projekt mitbegründet hat. Dank an Jeremy Ginsberg, Jack Hebert, John Skidgel, Evan Martin, Tony Chang, Troy Trimble, Tessa Pupius und Dylan Lorimer, die mir halfen, zu lernen. Dank an Sagnik Nandy und Waleed Ojeil für euer Mentoring.

Dank an meine inspirierenden Programmierlehrer: Ben Chelf, Glenn Cowan, Vince Hugo, Russ Lewin, Jon Stemmle, Derek Thomson und Daniel Wang. Ohne eure Anleitung wäre es mir nie gelungen, unsere Tätigkeit auszuüben, und schon gar nicht, sie anderen zu lehren.

Dank an meine Mutter, die mir das Gefühl gab, etwas Sinnvolles zu tun, und mich ermutigte, Programmierer zu werden. Dank an meinen Bruder, meine Großeltern und den Rest meiner Familie sowie Jugendfreunde, die mir Vorbilder waren, als ich aufwuchs und meine Leidenschaft für die Programmierung entwickelte.

Und zum Schluss Dank an meine Frau Colleen, für ihre Zuneigung, ihre Unterstützung und ihr Lachen auf dem gemeinsamen Lebensweg.

Kapitel 1

Pythons Sicht der Dinge

Die Eigenheiten einer Programmiersprache werden durch die Benutzer festgelegt. Im Laufe der Jahre hat sich in der Python-Community das englische Adjektiv *pythonic* eingebürgert, das Code beschreibt, der einem bestimmten Programmierstil folgt. Dieser Stil ist nicht reglementiert und spielt für den Compiler keine Rolle. Er hat sich im Laufe der Zeit allmählich durch die im Umgang mit der Sprache gewonnene Erfahrung und die Zusammenarbeit der Community entwickelt. Python-Programmierer befürworten eine klare Ausdrucksweise, ziehen Einfaches Komplexem vor und maximieren die Lesbarkeit. (Geben Sie doch mal `import this` im Python-Interpreter ein, um *The Zen of Python* zu lesen.)

Mit anderen Sprachen vertraute Programmierer versuchen oft, Python-Code im Stil von C++ oder Java zu schreiben (oder im Stil derjenigen Sprache, die sie eben am besten kennen). Programmieranfänger gewöhnen sich aber meist schnell an die große Vielfalt von Konzepten, die sich in Python formulieren lassen. Es ist für alle Programmierer gleichermaßen wichtig, die beste Art und Weise zu kennen, mit der die gebräuchlichsten Aufgaben in Python erledigt werden. Davon sind alle Programme betroffen, die Sie jemals schreiben werden.

Punkt 1 Kenntnis der Python-Version

Der größte Teil der Beispiele in diesem Buch verwendet die Syntax der Python-Version 3.7, die im Juni 2018 veröffentlicht wurde. Es kommen jedoch auch einige Beispiele vor, in denen die Syntax der Version 3.8 verwendet wird (veröffentlicht im Oktober 2019), um neue Features aufzuzeigen, die schon bald weiter verbreitet sein werden. Dieses Buch befasst sich nicht mit Python 2.

Auf vielen Systemen sind mehrere Versionen der Standard-Laufzeitumgebung CPython vorinstalliert. Dabei ist die standardmäßige Bedeutung des Befehls `python` auf der Kommandozeile nicht immer ganz klar. Meist ist `python` ein Alias für `python2.7`, gelegentlich verweist es aber auch auf ältere Versionen wie `python2.6` oder `python2.5`. Verwenden Sie den Parameter `--version`, um die genaue Versionsnummer herauszufinden.

```
$ python --version
Python 2.7.10
```

Python 3 ist für gewöhnlich unter der Bezeichnung `python3` verfügbar.

```
$ python3 --version
Python 3.8.0
```

Sie können die Versionsnummer auch zur Laufzeit feststellen, indem Sie die Werte des integrierten `sys`-Moduls überprüfen.

```
import sys
print(sys.version_info)
print(sys.version)

>>>
sys.version_info(major=3, minor=8, micro=0,
releaselevel='final', serial=0)
3.8.0 (default, Oct 21 2019, 12:51:17)
[Clang 6.0 (clang-600.0.57)]
```

Python 3 wird von den Python-Kernentwicklern und der Community gepflegt und kontinuierlich verbessert. Python 3 enthält eine Vielzahl leistungsfähiger neuer Features, die in diesem Buch erläutert werden. Die meisten der gebräuchlichen Open-Source-Bibliotheken sind mit Python 3.1 kompatibel und nutzen die neuen Features. Ich empfehle dringend, für alle weiteren Python-Projekte die Version 3 zu verwenden.

Python2 ist zum 1. Januar 2020 abgekündigt (End-of-Life). Nach diesem Termin wird es keine Fehlerbehebungen, Sicherheitspatches und Rückportierungen von Features mehr geben. Die Verwendung von Python 2 nach diesem Datum geschieht auf eigene Gefahr, weil es keine offizielle Unterstützung mehr gibt. Wenn Sie noch immer auf eine Codebasis in Python 2 angewiesen sind, sollten Sie in Betracht ziehen, hilfreiche Werkzeuge wie `2to3` (wird zusammen mit Python installiert) und `six` (steht als Paket zur Verfügung; siehe Punkt 82: *Module der Python-Community*) zu verwenden, um zu Python 3 zu wechseln.

Kompakt

- Python 3 ist die aktuellste und am besten unterstützte Version und sollte für neue Projekte verwendet werden.
- Vergewissern Sie sich, dass auf der Kommandozeile Ihres Systems auch wirklich die erwartete Python-Version ausgeführt wird.
- Verzichten Sie auf Python 2, da es nach dem 1. Januar 2020 nicht mehr unterstützt wird.

Punkt 2 Stilregeln gemäß PEP 8

Das Dokument *Python Enhancement Proposal #8* (Vorschläge zur Verbesserung von Python Nr. 8) oder kurz PEP 8 gibt die Stilregeln zur Formatierung von Python-Code vor. Es steht Ihnen natürlich frei, Ihren Python-Code so zu formatieren, wie Sie es für richtig halten, sofern Sie die Syntax beachten. Allerdings macht ein einheitlicher Stil den Code verständlicher und besser lesbar. Außerdem erleichtert ein einheitlicher Stil die Zusammenarbeit mit anderen Python-Programmierern der Community an größeren Projekten. Und auch wenn nur Sie selbst den Code jemals lesen werden, vereinfacht ein einheitlicher Stil nachträgliche Änderungen und hilft dabei, viele typische Fehler zu vermeiden.

PEP 8 legt eine Vielzahl von Details fest, die beim Schreiben von klarem Python-Code zu beachten sind. Die Regeln werden während der Weiterentwicklung von Python ständig überarbeitet. Es lohnt sich, das gesamte Dokument zu lesen (`https://www.python.org/dev/peps/pep-0008/`). Hier sind einige der Regeln, die Sie unbedingt einhalten sollten:

- **Leerraum (Whitespace):** In Python ist Leerraum syntaktisch von Bedeutung, daher schenken Python-Programmierer dem Einfluss von Leerraum auf die Klarheit des Codes besondere Beachtung.
  - Verwenden Sie Leerzeichen zum Einrücken von Text, keine Tabulatoren.
  - Verwenden Sie für jede Ebene syntaktisch relevanter Einrückungen vier Leerzeichen.
  - Zeilen sollten nicht mehr als 79 Zeichen enthalten.
  - Die Fortsetzung eines langen Ausdrucks in der nächsten Zeile sollte um zusätzliche vier Leerzeichen eingerückt werden.
  - In Dateien sollten Funktionen und Klassen durch zwei Leerzeilen voneinander getrennt sein.
  - Innerhalb einer Klasse sollten Methoden durch eine Leerzeile voneinander getrennt sein.
  - Fügen Sie in einem Dictionary kein Leerzeichen zwischen Schlüssel und Doppelpunkt ein. Fügen Sie genau ein Leerzeichen vor dem dazugehörigen Wert ein, sofern er in dieselbe Zeile passt.
  - Fügen Sie bei Variablenzuweisungen genau ein Leerzeichen vor und nach dem Zuweisungsoperator ein.
  - Vergewissern Sie sich, dass bei Methodensignaturzuweisungen zwischen Variablennamen und Doppelpunkt kein Leerraum steht. Verwenden Sie ein Leerzeichen vor der Typenbezeichnung.

- **Bezeichner:** Gemäß PEP 8 sollten zur Benennung der verschiedenen Teile der Sprache unterschiedliche Stile verwendet werden. Dadurch wird es beim Betrachten des Codes vereinfacht, den Typ eines Bezeichners zu erkennen.
  - Funktionen, Variablen und Attribute sollten als `kleinbuchstaben_mit_unterstrich` geschrieben werden.
  - Geschützte Instanzattribute sollten mit `_führendem_unterstrich` geschrieben werden.
  - Private Instanzattribute sollten mit `__doppelten_führenden_unterstrichen` geschrieben werden.
  - Klassen und Exceptions sollten als `GroßgeschriebeneWörter` geschrieben werden.
  - Modulweite Konstanten sollten `IN_GROSSBUCHSTABEN` geschrieben werden.
  - Instanzmethoden einer Klasse sollten als Bezeichnung des ersten Parameters (der auf das Objekt verweist) `self` verwenden.
  - Klassenmethoden sollten als Bezeichnung des ersten Parameters (der auf die Klasse verweist) `cls` verwenden.
- **Ausdrücke und Anweisungen:** In *The Zen of Python* heißt es: »Es sollte einen – und vorzugsweise *nur* einen – offensichtlichen Weg geben, ein Problem zu lösen.« PEP 8 schreibt daher für Ausdrücke und Anweisungen Folgendes fest:
  - Verwenden Sie Negierungen möglichst inmitten eines Ausdrucks (`if a is not b`), statt positive Ausdrücke zu negieren (`if not a is b`).
  - Testen Sie leere Werte wie `[]` oder `''` nicht, indem Sie deren Länge überprüfen (`if len(eineliste) == 0`). Verwenden Sie stattdessen `if not eineliste`. Leere Werte werden implizit als `False` bewertet.
  - Gleiches gilt für nicht leere Werte wie `[1]` oder `'hallo'`. Die Anweisung `if eineliste` wird für nicht leere Werte implizit als `True` bewertet.
  - Verzichten Sie auf einzeilige `if`-Anweisungen, `for`- bzw. `while`-Schleifen und zusammengesetzte `except`-Anweisungen. Verteilen Sie sie der Übersichtlichkeit halber auf mehrere Zeilen.
  - Wenn ein Ausdruck nicht in eine Zeile passt, sollten Sie ihn einklammern und Zeilenumbrüche und Einrückungen verwenden, um die Lesbarkeit zu verbessern.
- **Importe:** Gemäß PEP gelten für Importe die folgenden Regeln:
  - Platzieren Sie `import`-Anweisungen (auch in der Form `from x import y`) immer am Anfang der Datei.
  - Verwenden Sie beim Import von Modulen stets absolute Pfadbezeichnungen, nicht die zum aktuellen Modul relativen. Wenn Sie beispielsweise das Modul `foo` des Pakets `bar` importieren, sollten Sie nicht `import foo`, sondern `from bar import foo` verwenden.

- Verwenden Sie die ausdrückliche Syntax `from . import foo`, falls Sie auf relative Pfade angewiesen sind.
- Importe sollten in folgender Reihenfolge stattfinden: Zuerst Module der Standardbibliothek, dann Module Dritter, dann die eigenen Module. Die jeweilige Gruppe sollte alphabetisch sortiert sein.

Hinweis

Pylint (`http://www.pylint.org`) ist ein beliebtes Werkzeug zur statischen Analyse von Python-Quelltexten. Pylint erzwingt automatisch die Einhaltung der Stilregeln gemäß PEP 8 und kann viele weitere typische Fehler in Python-Programmen entdecken. Viele IDEs und Editoren bringen Linting-Tools mit oder unterstützen Plug-ins mit vergleichbarer Funktionalität.

Kompakt

- Halten Sie sich beim Schreiben von Python-Code an die Stilregeln gemäß PEP 8.
- Ein einheitlicher Stil erleichtert die Zusammenarbeit mit anderen Programmierern der Python-Community.
- Außerdem erleichtert ein einheitlicher Stil nachträgliche Änderungen des eigenen Codes.

Punkt 3 Unterschiede zwischen bytes und str

In Python 3 gibt es zwei Datentypen, die Zeichenfolgen repräsentieren: `bytes` und `str`. `bytes`-Instanzen enthalten reine, vorzeichenlose 8-Bit-Werte (die häufig als ASCII-Encoding dargestellt werden):

```
a = b'h\x65llo'
print(list(a))
print(a)
>>>
[104, 101, 108, 108, 111]
b'hello'
```

In `str`-Instanzen hingegen sind Unicode-Codepoints gespeichert, die Textzeichen repräsentieren.

```
a = 'a\u0300 propos'
print(list(a))
print(a)
```

```
>>>
['a', '`', ' ', 'p', 'r', 'o', 'p', 'o', 's']
à propos
```

Den `str`-Instanzen ist jedoch keine Binärcodierung und den `bytes`-Instanzen ist keine Textcodierung zugeordnet. Um Unicode-Zeichen in Binärdaten umzuwandeln, müssen Sie die `encode`-Methode von `str` verwenden. Zur Umwandlung der Binärdaten in Unicode-Zeichen dient die Methode `decode` von `bytes`. Sie können das gewünschte Encoding für diese Methoden ausdrücklich angeben oder die Voreinstellung des Systems verwenden. Dabei handelt es sich typischerweise um UTF-8 (aber nicht immer – mehr dazu in Kürze).

Beim Schreiben Ihres Python-Programms sollten Sie darauf achten, die Codierung und Decodierung von Unicode an den äußersten Schnittstellen vorzunehmen. Ihr eigentliches Programm sollte den Unicode-Datentyp `str` verwenden und keinerlei Annahmen über die Zeichencodierung machen. Auf diese Weise können Sie auch alternative Textcodierungen (wie Latin-1, Shift JIS oder Big5) einlesen, gleichzeitig aber bei der Ausgabe strikt auf eine bestimmte Textcodierung setzen (idealerweise UTF-8).

Das Vorhandensein der beiden unterschiedlichen Zeichendatentypen führt zu zwei typischen Situationen:

- Sie möchten reine 8-Bit-Werte verarbeiten, die als UTF-8-Zeichen codiert sind (oder eine andere Textcodierung aufweisen).
- Sie möchten Unicode-Zeichen verarbeiten, denen keine Textcodierung zugeordnet ist.

Sie benötigen dann zwei Hilfsfunktionen zur Konvertierung der verschiedenen Zeichentypen, die gewährleisten, dass die Eingabewerte den Erwartungen Ihres Codes entsprechen.

Die erste Funktion nimmt eine `str`- oder eine `bytes`-Instanz entgegen und liefert stets eine `str`-Instanz zurück:

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value # str-Instanz
print(repr(to_str(b'foo')))
print(repr(to_str('bar')))
>>>
```

```
'foo'
'bar'
```

Die zweite Funktion nimmt eine `str`- oder eine `bytes`-Instanz entgegen und gibt stets eine `bytes`-Instanz zurück

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value # bytes-Instanz
print(repr(to_bytes(b'foo')))
print(repr(to_bytes('bar')))
```

Es gibt bei der Verwendung von reinen 8-Bit-Werten und Unicode-Zeichen in Python zwei große Vorbehalte.

Zum einen scheinen `bytes` und `str` auf dieselbe Art und Weise zu funktionieren, aber die Instanzen sind nicht miteinander kompatibel. Deshalb müssen Sie stets darauf achtgeben, welche Art Zeichenfolge Sie übergeben.

Mit dem +-Operator können Sie `bytes` mit `bytes` bzw. `str` mit `str` verknüpfen:

```
print(b'one' + b'two')
print('one' + 'two')
>>>
b'onetwo'
onetwo
```

`str`-Instanzen können jedoch nicht mit `bytes`-Instanzen verknüpft werden:

```
b'one' + 'two'
>>>
Traceback ...
TypeError: can't concat str to bytes
```

Und `bytes`-Instanzen nicht mit `str`-Instanzen:

```
'one' + b'two'
>>>
Traceback ...
```

```
TypeError: can only concatenate str \
(not "bytes") to str
```

Sie können `bytes` mit `bytes` bzw. `str` mit `str` vergleichen:

```
assert b'red' > b'blue'
assert 'red' > 'blue'
```

Nicht aber eine `str`-Instanz mit einer `bytes`-Instanz:

```
assert 'red' > b'blue'
>>>
Traceback ...
TypeError: '>' not supported between instances \
of 'str' and 'bytes'
```

Und auch eine `bytes`-Instanz nicht mit einer `str`-Instanz:

```
assert b'blue' < 'red'
>>>
Traceback ...
TypeError: '<' not supported between instances \
of 'bytes' and 'str'
```

Der Test auf Gleichheit von `bytes`- und `str`-Instanzen ergibt stets `False`, selbst wenn beide genau dieselben Zeichen enthalten (in diesem Fall »foo« in ASCII-Codierung):

```
print(b'foo' == 'foo')
>>>
False
```

Dem %-Operator können Sie einen Formatierungsstring beider Typen übergeben:

```
print(b'red %s' % b'blue')
print('red %s' % 'blue')
>>>
b'red blue'
red blue
```

Sie können mit einem Formatierungsstring im `bytes`-Format jedoch keine `str`-Instanz verwenden, weil Python nicht weiß, welche binäre Textcodierung benutzt werden soll:

```
print(b'red %s' % 'blue')
>>>
Traceback ...
TypeError: %b requires a bytes-like object, or \
an object that implements __bytes__, not 'str'
```

Sie können mit dem %-Operator einem Formatierungsstring im `str`-Format zwar eine `bytes`-Instanz übergeben, aber dann geschieht nicht das, was man erwartet:

```
print('red %s' % b'blue')
>>>
red b'blue'
```

Tatsächlich ruft dieser Code die `__repr__`-Methode der `bytes`-Instanz auf (siehe Punkt 75: *Debuggen mit repr-Strings*) und ersetzt das `%s` durch diese – deshalb bleibt das `b'blue'` in der Ausgabe »escaped«.

Der zweite Vorbehalt ergibt sich dadurch, dass Operationen mit Datei-Handles (die von der integrierten `open`-Funktion zurückgegeben werden) standardmäßig von einer Unicode-Codierung ausgehen, nicht von einer Binärcodierung. Das kann zu überraschenden Fehlschlägen führen, insbesondere dann, wenn die Programmierer an Python 2 gewohnt sind.

Wenn ich beispielsweise einige Binärdaten in eine Datei schreiben möchte, schlägt der nachstehende, scheinbar einfache Code fehl:

```
with open('data.bin', 'w') as f:
    f.write(b'\xf1\xf2\xf3\xf4\xf5')
>>>
Traceback ...
TypeError: write() argument must be str, not bytes
```

Der Grund für diesen Fehler ist, dass die Datei im binären Schreibmodus (write binary, `'wb'`) statt im zeichenweisen Schreibmodus (`'w'`) geöffnet werden muss. Bei einer Datei im zeichenweisen Schreibmodus erwarteten `write`-Operationen `str`-Instanzen, die Unicode-Daten enthalten, keine `bytes`-Instanzen mit Binärdaten. Hier ändere ich den `open`-Modus auf `'wb'`, damit es korrekt funktioniert:

```
with open('data.bin', 'wb') as f:
    f.write(b'\xf1\xf2\xf3\xf4\xf5')
```

Dieses Problem tritt beim Lesen von Dateien ebenfalls auf. Hier versuche ich beispielsweise, die soeben geschriebene Binärdatei zu lesen:

```
with open('data.bin', 'r') as f:
    data = f.read()
>>>
Traceback ...
UnicodeDecodeError: 'utf-8' codec can't decode \
byte 0xf1 in position 0: invalid continuation byte
```

Das schlägt fehl, weil die Datei statt im binären Lesemodus (`'rb'`) im Textmodus (`'r'`) geöffnet wurde. In diesem Modus wird das Standardencoding des Systems für die Methoden `bytes.encode` (beim Schreiben) und `str.decode` (beim Lesen) zur Interpretation von Binärdaten verwendet. Bei den meisten Systemen ist das Standardencoding UTF-8, das mit den Binärdaten `b'\xf1\xf2\xf3\xf4\xf5'` nicht zurechtkommt und deshalb den obigen Fehler verursacht. Hier gebe ich beim Öffnen der Datei mittels `'rb'` an, dass die Datei im binären Modus geöffnet werden soll:

```
with open('data.bin', 'rb') as f:
    data = f.read()
assert data == b'\xf1\xf2\xf3\xf4\xf5'
```

Alternativ könnte ich der `open`-Funktion ausdrücklich den Parameter `encoding` übergeben, um zu gewährleisten, dass es zu keinem plattformabhängigen überraschenden Verhalten kommt. Hier gehe ich beispielsweise davon aus, dass die Daten in der Datei als `'cp1252'` (ein älteres Windows-Encoding) gespeichert sind:

```
with open('data.bin', 'r', encoding='cp1252') as f:
    data = f.read()
assert data == 'ñòóôõ'
```

Der Fehler ist behoben und die Interpretation des Dateiinhalts unterscheidet sich sehr von der Ausgabe beim Lesen als reine Bytes. Man sollte also immer das Standardencoding des Systems überprüfen (durch `python3 -c 'import locale; print(locale.getpreferredencoding())'`), um in Erfahrung zu bringen, inwieweit es den Erwartungen entspricht. Im Zweifelsfall sollten Sie beim Öffnen den `encoding`-Parameter übergeben.

Kompakt

- `bytes`-Instanzen enthalten reine 8-Bit-Werte und `str`-Instanzen Unicode-Zeichenfolgen.
- `bytes`- und `str`-Instanzen können *nicht* durch Operatoren wie >, ==, + oder % miteinander verknüpft werden.
- Verwenden Sie Hilfsfunktionen, um zu gewährleisten, dass die zu verarbeitenden Eingaben im erwarteten Format (8-Bit-Werte, UTF-8-codierte Zeichen, Unicode-Zeichen usw.) vorliegen.
- Beim Lesen oder Schreiben von Binärdaten müssen Sie die Datei im binären Modus (wie z.B. `'rb'` oder `'wb'`) öffnen.
- Beachten Sie beim Lesen oder Schreiben von Unicode-Daten das Standardtextencoding Ihres Systems. Übergeben Sie der `open`-Methode den Parameter `encoding`, um Überraschungen zu vermeiden.

Punkt 4 Interpolierte F-Strings statt C-Formatierungsstrings und str.format

Strings sind in der Codebasis von Python überall vorhanden. Sie werden zur Ausgabe von Meldungen der Benutzeroberfläche oder von Kommandozeilenprogrammen verwendet. Sie werden auch genutzt, um Daten in Dateien zu schreiben oder über Sockets auszugeben. Sie beschreiben im Falle von Exceptions, was genau schiefgegangen ist (siehe Punkt 27: *Listen-Abstraktionen statt map und filter*). Und sie werden beim Debugging verwendet (siehe Punkt 80: *Interaktives Debuggen mit pdb* und Punkt 75: *Debuggen mit repr-Strings*).

Bei der *Formatierung* werden vordefinierte Texte mit Datenwerten zu einer für Menschen verständlichen Mitteilung kombiniert, die als String gespeichert wird. In Python gibt es vier verschiedene Methoden zur Formatierung von Strings, die Teile der Sprache selbst sind oder zur Standardbibliothek gehören. Bis auf eine, die in diesem Abschnitt zuletzt erörtert wird, weisen sie jedoch Unzulänglichkeiten auf, die Sie kennen und vermeiden sollten.

Am gebräuchlichsten ist es, Strings in Python mit dem %-Operator zu formatieren. Die vordefinierte Textvorlage wird auf der linken Seite des Operators als *Formatierungsstring* bereitgestellt, die in die Vorlage einzufügenden Daten stehen als einzelne Werte oder als Tupel auf der rechten Seite des Formatierungsoperators. Hier verwende ich den %-Operator beispielsweise, um schwer verständliche binäre und hexadezimale Werte in Dezimalzahlen zu konvertieren:

```
a = 0b10111011
b = 0xc5f
print('Binär %d, Hex %d' % (a, b))
```

```
>>>
Binär 187, Hex 3167
```

Im Formatierungsstring werden Platzhalter (wie `%d`) verwendet, die durch die Werte auf der rechten Seite des Ausdrucks ersetzt werden. Die Syntax für diese Platzhalter hat Python von der C-Funktion `printf` übernommen (wie viele andere Programmiersprachen auch). Python unterstützt alle üblichen Optionen, die man von `printf` erwarten würde, wie etwa `%s`, `%x` oder `%f` sowie die Angabe der Dezimalstellen, das Auffüllen mit Leerzeichen und die bündige Ausrichtung. Viele Programmierer, die Python lernen, nutzen Formatierungsstring im C-Stil, weil sie ihnen vertraut sind und sich einfach nutzen lassen.

Bei der Verwendung von Formatierungsstrings im C-Stil treten in Python vier Probleme auf.

Das erste Problem: Wenn man den Typ oder die Reihenfolge der Datenwerte im Tupel auf der rechten Seite des Ausdrucks ändert, kann es aufgrund von Inkompatibilitäten bei der Konvertierung zu Fehlern kommen. Dieser einfache Formatierungsausdruck beispielsweise funktioniert:

```
key = 'my_var'
value = 1.234
formatted = '%-10s = %.2f' % (key, value)
print(formatted)
>>>
my_var     = 1.23
```

Aber wenn `key` und `value` vertauscht werden, gibt es einen Laufzeitfehler:

```
reordered_tuple = '%-10s = %.2f' % (value, key)
>>>
Traceback ...
TypeError: must be real number, not str
```

Belässt man die Parameter auf der rechten Seite in der ursprünglichen Reihenfolge, ändert aber den Formatierungsstring, kommt es zum gleichen Fehler:

```
reordered_string = '%.2f = %-10s' % (key, value)
>>>
Traceback ...
TypeError: must be real number, not str
```

Um derartige Fehler zu vermeiden, muss man stets darauf achten, dass die beiden Seiten des %-Operators aufeinander abgestimmt sind. Dabei kommt es leicht zu Fehlern, weil das bei jeder Änderung manuell erfolgen muss.

Das zweite Problem ist, dass Formatierungsstrings schwer verständlich sind, wenn man kleinere Modifikationen an den Werten vornehmen muss, bevor sie in einen String umgewandelt werden – und das kommt äußerst häufig vor. Hier gebe ich den Inhalt meines Küchenschranks aus, ohne inline Änderungen vorzunehmen:

```
pantry = [
    ('Avocados', 1.25),
    ('Bananen', 2.5),
    ('Kirschen', 15),
]
for i, (item, count) in enumerate(pantry):
    print('#%d: %-10s = %.2f' % (i, item, count))
>>>
#0: Avocados   = 1.25
#1: Bananen    = 2.50
#2: Kirschen   = 15.00
```

Nun nehme ich einige Änderungen an den zu formatierenden Werten vor, damit die Ausgabe nützlicher ist. Dadurch wird das Tupel im Formatierungsausdruck so lang, dass es auf mehrere Zeilen aufgeteilt werden muss, was die Verständlichkeit beeinträchtigt:

```
for i, (item, count) in enumerate(pantry):
    print('#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count)))
>>>
#1: Avocados   = 1
#2: Bananen    = 2
#3: Kirschen   = 15
```

Das dritte Problem mit Formatierungsausdrücken betrifft die mehrmalige Verwendung eines Werts, denn er muss im Tupel auf der rechten Seite wiederholt angegeben werden:

```
template = '%s liebt Essen. Sieh %s beim Kochen zu.'
name = 'Max'
formatted = template % (name, name)
print(formatted)
>>>
Max liebt Essen. Sieh Max beim Kochen zu.
```

Das ist besonders lästig und fehleranfällig, wenn an den zu formatierenden Werten mehrfach kleine Modifikationen vorgenommen werden müssen. Hier habe ich daran gedacht, die `title()`-Methode mehrmals aufzurufen, aber es hätte mir leicht passieren können, dass ich den Aufruf bei einem der Verweise auf `name` hinzufüge und beim anderen nicht, was eine fehlerhafte Ausgabe erzeugen würde:

```
name = 'Brad'
formatted = template % (name.title(), name.title())
print(formatted)
>>>
Brad liebt Essen. Sieh Brad beim Kochen zu.
```

Um diese Probleme zumindest teilweise zu lösen, bietet der %-Operator in Python die Möglichkeit, statt eines Tupels ein Dictionary für Formatierungen zu verwenden. Die Schlüssel des Dictionarys werden den Platzhaltern entsprechend benannt, wie beispielsweise `%(Schlüsselname)s`. Hier nutze ich diese Funktionalität und ändere die Reihenfolge der Werte auf der rechten Seite, ohne dass es Auswirkungen auf die Ausgabe hat, was das erstgenannte obige Problem löst:

```
key = 'my_var'
value = 1.234
old_way = '%-10s = %.2f' % (key, value)
new_way = '%(key)-10s = %(value).2f' % {
    'key': key, 'value': value}  # Ursprünglich
reordered = '%(key)-10s = %(value).2f' % {
    'value': value, 'key': key}  # Vertauscht
assert old_way == new_way == reordered
```

Die Verwendung von Dictionaries in Formatierungsausdrücken löst auch das dritte oben genannte Problem, weil mehrere Platzhalter auf denselben Wert verweisen können, was es überflüssig macht, ihn mehrfach anzugeben:

```
name = 'Max'
template = '%s liebt Essen. Sieh %s beim Kochen zu.'
```

```
before = template % (name, name)   # Tupel
template = \
       '%(name)s liebt Essen. Sieh %(name)s beim Kochen zu.'
after = template % {'name': name}  # Dictionary
assert before == after
```

Dictionaries als Formatierungsstrings bringen allerdings ihre eigenen Probleme mit sich. Beim zweiten oben genannten Problem werden die Formatierungsausdrücke durch die Verwendung von Dictionaries länger und optisch auffälliger, weil die rechte Seite den Schlüssel und den Doppelpunkt enthält. Hier gebe ich den gleichen String mit und ohne Dictionaries aus, um das Problem zu demonstrieren:

```
for i, (item, count) in enumerate(pantry):
    before = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))
    after = '#%(loop)d: %(item)-10s = %(count)d' % {
        'loop': i + 1,
        'item': item.title(),
        'count': round(count),
    }
    assert before == after
```

Die Verwendung von Dictionaries in Formatierungsstrings vergrößert zudem den Umfang des Codes, was das vierte Problem mit Formatierungsstrings im C-Stil darstellt. Jeder Schlüssel muss mindestens zweimal angegeben werden. Einmal als Platzhalter, einmal als Schlüssel im Dictionary und potenziell nochmals im Variablennamen, der den Dictionary-Wert enthält:

```
soup = 'Linsensuppe'
formatted = 'Heute gibt es %(soup)s.' % {'soup': soup}
print(formatted)
>>>
Heute gibt es Linsensuppe.
```

Neben den doppelt vorhandenen Zeichen sorgt diese Redundanz dafür, dass Formatierungsausdrücke, die Dictionaries verwenden, sehr lang sind. Die Ausdrücke sind oft mehrere Zeilen lang. Die Formatierungsstrings werden über mehrere Zeilen hinweg verkettet, und die Zuweisungen zum Dictionary belegen pro Wert jeweils eine Zeile:

```
menu = {
    'soup': 'Linsensuppe',
    'oyster': 'Kumamoto',
    'special': 'Schnitzel',
}
template = ('Heute gibt es %(soup)s, '
            'als Vorspeise zwei %(oyster)s-Austern, '
            'und das Tagesgericht ist %(special)s.')
formatted = template % menu
print(formatted)
>>>
Heute gibt es Linsensuppe, als Vorspeise zwei \
Kumamoto-Austern und das Tagesgericht ist Schnitzel.
```

Um erkennen zu können, welche Ausgabe dieser Ausdruck erzeugt, muss man ständig zwischen den Zeilen des Formatierungsstrings und denen des Dictionarys hin und her springen. Durch diese Trennung ist es schwierig, Bugs zu erkennen, und die Verständlichkeit leidet weiter, wenn vor der Formatierung kleine Modifikationen an den Werten vorgenommen werden müssen.

Hierfür muss es eine bessere Möglichkeit geben.

Die integrierte format-Funktion und str.format

Python 3 unterstützt erweiterte Stringformatierungen, die aussagekräftiger sind als die älteren Formatierungsstrings im C-Stil, die den %-Operator verwenden. Auf die einzelnen Werte kann man mit der integrierten `format`-Funktion zugreifen. Hier verwende ich einige der neuen Optionen (»,« als Trennzeichen für Tausend und »^« zum Zentrieren), um Werte zu formatieren:

```
a = 1234.5678
formatted = format(a, ',.2f')
print(formatted)
b = 'my string'
formatted = format(b, '^20s')
print('*', formatted, '*')
>>>
1,234.57
*      my string       *
```

Sie können diese Funktionalität nutzen, um mehrere Werte gleichzeitig zu formatieren, indem Sie die neue `format`-Methode des `str`-Typs aufrufen. Statt Platz-

halter im C-Stil wie `%d` zu verwenden, können Sie diese mit `{ }` angeben. Die Platzhalter im Formatierungsstring werden standardmäßig durch die Positionsargumente ersetzt, die der `format`-Methode übergeben werden, und zwar in der Reihenfolge, in der sie erscheinen:

```
key = 'my_var'
value = 1.234
formatted = '{} = {}'.format(key, value)
print(formatted)
>>>
my_var = 1.234
```

Bei jedem Platzhalter können Sie optional einen Doppelpunkt nebst Formatbezeichner angeben, um festzulegen, wie Werte in Strings konvertiert werden. (Geben Sie `help('FORMATTING')` ein, um alle Optionen anzuzeigen.)

```
formatted = '{:<10} = {:.2f}'.format(key, value)
print(formatted)
>>>
my_var     = 1.23
```

Die Funktionsweise kann man sich so vorstellen: Der Formatbezeichner wird zusammen mit dem Wert der integrierten `format`-Funktion übergeben (`format(value, '.2f')` im obigen Beispiel). Das Ergebnis dieses Funktionsaufrufs ersetzt den Platzhalter im formatierten String. Das Verhalten der Formatierung kann für eine Klasse durch die spezielle Methode `__format__` angepasst werden.

Bei Formatierungsstrings im C-Stil muss man das %-Zeichen escapen, indem es doppelt eingegeben wird, damit es nicht als Platzhalter interpretiert wird. Bei der `str.format`-Methode müssen Sie die geschweiften Klammern auf ähnliche Weise schützen:

```
print('%.2f%%' % 12.5)
print('{} ersetzt {{}}'.format(1.23))
>>>
12.50%
1.23 ersetzt {}
```

In den geschweiften Klammern können Sie einen Positionsindex angeben, der an die `format`-Methode übergeben wird und den Platzhalter ersetzt. Das ermöglicht es, den Formatierungsstring zwecks Umordnung zu aktualisieren, ohne die rechte Seite des Formatierungsausdrucks ändern zu müssen. Dadurch wird das erste oben genannte Problem gelöst:

```
formatted = '{1} = {0}'.format(key, value)
print(formatted)
>>>
1.234 = my_var
```

Auf diesen Positionsindex kann im Formatierungsstring mehrfach verwiesen werden, ohne ihn der `format`-Methode mehr als einmal übergeben zu müssen. Das löst das dritte oben genannte Problem:

```
formatted = '{0} liebt Essen. Sieh {0} beim \
Kochen zu.'.format(name)
print(formatted)
>>>
Max liebt Essen. Sieh Max beim Kochen zu.
```

Leider bietet die neue `format`-Methode keine Lösung für das zweite oben genannte Problem. Der Code bleibt schwer verständlich, wenn man vor der Formatierung kleine Modifikationen an den Werten vornehmen muss. Bei der Verständlichkeit gibt es kaum einen Unterschied zwischen der alten und der neuen Version:

```
for i, (item, count) in enumerate(pantry):
    old_style = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))
    new_style = '#{}: {:<10s} = {}'.format(
        i + 1,
        item.title(),
        round(count))
    assert old_style == new_style
```

Es gibt noch ausgeklügeltere Optionen für die Bezeichner der `str.format`-Methode, beispielsweise die Verwendung einer Kombination aus Dictionary-Schlüsseln und Listenindizes in Platzhaltern oder die erzwungene Umwandlung von Werten in Unicode und `repr`-Strings:

```
formatted = 'Erster Buchstabe {menu[oyster][0]!r}'.format(
    menu=menu)
print(formatted)
>>>
Erster Buchstabe 'k'
```

Allerdings helfen auch diese Features nicht, die Redundanz durch mehrfach vorkommende Schlüssel zu verringern (das vierte der oben genannten Probleme). Hier ist beispielsweise ein Vergleich der Ausführlichkeit bei der Verwendung von Dictionaries mit Formatierungsausdrücken im C-Stil und der Übergabe von Schlüsseln als Argumente an die `format`-Methode:

```
old_template = (
    'Heute gibt es %(soup)s, '
    'als Vorspeise zwei %(oyster)s-Austern, '
    'und das Tagesgericht ist %(special)s.')
old_formatted = template % {
    'soup': 'Linsensuppe',
    'oyster': 'Kumamoto',
    'special': 'Schnitzel',
}
new_template = (
    'Heute git es {soup}, '
    'als Vorspeise zwei {oyster}-Austern, '
    'und das Tagesgericht ist {special}.')
new_formatted = new_template.format(
    soup='Linsensuppe',
    oyster='Kumamoto',
    special='Schnitzel',
)
assert old_formatted == new_formatted
```

Dieser Stil ist optisch zwar etwas weniger auffällig, weil einige der Anführungsstriche im Dictionary und ein paar Buchstaben in den Formatbezeichnern entfallen, aber er ist wohl kaum als elegant zu bezeichnen. Die erweiterten Features (Verwendung von Dictionary-Schlüsseln und Indizes in Platzhaltern) bieten lediglich eine kleine Untermenge der Funktionalität von Python-Ausdrücken. Diese fehlenden Ausdrucksmöglichkeiten wirken sich so einschränkend aus, dass sie den Nutzen der `str.format`-Methode insgesamt infrage stellen.

In Anbetracht dieser Unzulänglichkeiten und der verbleibenden Probleme mit Formatierungsausdrücken im C-Stil (das zweite und vierte der oben genannten Probleme) empfehle ich, auf die `str.format`-Methode im Allgemeinen zu verzichten. Es ist zwar wichtig, die neue »Minisprache« zu kennen, die in Formatbezeichnern verwendet wird (alles, was dem Doppelpunkt folgt), und zu wissen, wie man die integrierte `format`-Funktion benutzt, aber der übrige Teil der `str.format`-Methode sollte als historisches Überbleibsel betrachtet werden, das Ihnen hilft zu verstehen, wie Pythons neue F-Strings funktionieren und weshalb sie so ungemein nützlich sind.

Interpolierte Formatierungsstrings

Seit der Python-Version 3.6 gibt es *interpolierte Formatierungsstrings* – kurz *F-Strings* –, um derartige Probleme ein für alle Mal zu lösen. Diese neue Sprachsyntax erfordert es, dass Formatierungsstrings der Buchstabe `f` vorangestellt wird, so wie bei Bytestrings ein `b` und bei Rohdaten (nicht escapt) ein `r` als Präfix verwendet wird.

F-Strings treiben die Ausdruckstärke von Formatierungsstrings sozusagen auf die Spitze und lösen das vierte der oben genannten Probleme, indem die Notwendigkeit, die zu formatierenden Werte und deren Schlüssel bereitzustellen, komplett entfällt. Das wird dadurch erreicht, dass innerhalb des Formatierungsstrings sämtliche Namen im aktuellen Gültigkeitsbereich verwendet werden können:

```
key = 'my_var'
value = 1.234
formatted = f'{key} = {value}'
print(formatted)
>>>
my_var = 1.234
```

Nach dem Doppelpunkt eines Platzhalters in einem F-String stehen sämtliche Optionen der »Minisprache« der integrierten `format`-Methode zur Verfügung, ebenso wie die Möglichkeit, Werte wie mit der `str.format`-Methode in Unicode und `repr`-Strings umzuwandeln:

```
formatted = f'{key!r:<10} = {value:.2f}'
print(formatted)
>>>
'my_var'   = 1.23
```

Die Formatierung mit F-Strings ist in allen Fällen kompakter als die Verwendung von Formatierungsstrings mit %-Operator im C-Stil oder der `str.format`-Methode. Hier verwende ich alle Optionen (in der Reihenfolge von kurz nach lang). Die linken Seiten der Zuweisungen sind bündig ausgerichtet, damit sie besser vergleichbar sind:

```
f_string = f'{key:<10} = {value:.2f}'
c_tuple  = '%-10s = %.2f' % (key, value)
str_args = '{:<10} = {:.2f}'.format(key, value)
str_kw   = '{key:<10} = {value:.2f}'.format(key=key, value=value)
c_dict   = '%(key)-10s = %(value).2f' % {'key': key, 'value': value}
```

```
assert c_tuple == c_dict == f_string
assert str_args == str_kw == f_string
```

Darüber hinaus erlauben F-Strings es, in den geschweiften Klammern des Platzhalters vollständige Python-Ausdrücke zu verwenden, was das zweite der oben genannten Probleme löst, weil nun kleine Modifikationen an den zu formatierenden Werten mit kompakter Syntax möglich sind. Wofür bei der Formatierung im C-Stil und bei der `str.format`-Methode mehrere Zeilen benötigt werden, passt jetzt problemlos in eine einzige Zeile:

```
for i, (item, count) in enumerate(pantry):
    old_style = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))

    new_style = '#{}: {:<10s} = {}'.format(
        i + 1,
        item.title(),
        round(count))
    f_string = f'#{i+1}: {item.title():<10s} = \
                                        {round(count)}'
    assert old_style == new_style == f_string
```

Wenn es der Deutlichkeit dient, kann ein F-String auch (ähnlich wie in C) durch Stringverkettung auf mehrere Zeilen aufgeteilt werden. Das ist zwar nicht mehr so kompakt wie die einzeilige Variante, aber immer noch viel besser verständlich als die anderen mehrzeiligen Versionen:

```
for i, (item, count) in enumerate(pantry):
    print(f'#{i+1}: '
    f'{item.title():<10s} = '
    f'{round(count)}')
>>>
#1: Avocados   = 1
#2: Bananen    = 2
#3: Kirschen   = 15
```

Auch die Optionen der Formatbezeichner dürfen Python-Ausdrücke enthalten. Hier parametrisiere ich die Anzahl der auszugebenden Dezimalstellen, indem ich im Formatierungsstring statt eines festen Werts eine Variable verwende:

```
places = 3
number = 1.23456
print(f'Meine Zahl ist {number:.{places}f}')
>>>
Meine Zahl ist 1.235
```

Diese Kombination aus Ausdrucksstärke, Kompaktheit und Verständlichkeit, die F-Strings bieten, machen sie für Python-Programmierer zur besten integrierten Lösung. Wenn Sie Werte in Strings umwandeln möchten, sollten Sie F-Strings den Vorzug geben.

Kompakt

- Formatierungsstrings im C-Stil, die den %-Operator verwenden, leiden unter einer Reihe von Problemen und sind sehr umfangreich.
- Die `str.format`-Methode bringt in Form der »Minisprache« der Formatbezeichner einige nützliche Konzepte mit, leidet aber unter ähnlichen Problemen wie die Formatierungsstrings im C-Stil und sollte vermieden werden.
- F-Strings sind eine neue Syntax, um Werte als Strings zu formatieren, und lösen die größten Probleme von Formatierungsstrings im C-Stil.
- F-Strings sind kompakt, aber dennoch sehr leistungsstark, weil sie es erlauben, in Formatbezeichnern beliebige Python-Ausdrücke zu verwenden.

Punkt 5 Hilfsfunktionen statt komplizierter Ausdrücke

Pythons prägnante Syntax ermöglicht es, in einzeiligen Ausdrücken eine Menge Programmlogik unterzubringen. Nehmen Sie beispielsweise an, Sie möchten den Abfrage-String einer URL decodieren. Hier sind sämtliche Parameter Ganzzahlen:

```
from urllib.parse import parse_qs
my_values = parse_qs('red=5&blue=0&green=',
                     keep_blank_values=True)
print(repr(my_values))
>>>
{'red': ['5'], 'blue': ['0'], 'green': ['']}
```

Manche Parameter könnten mehrere Werte besitzen, einige auch nur einen, weitere sind zwar vorhanden, aber leer, und wieder andere fehlen womöglich ganz. Die `get`-Methode des Dictionarys kann also verschiedene Werte zurückliefern:

```
print('Rot:       ', my_values.get('red'))
print('Grün:      ', my_values.get('green'))
print('Deckkraft: ', my_values.get('opacity'))
>>>
Rot:       ['5']
Grün:      ['']
Deckkraft: None
```

Es wäre schön, wenn nicht vorhandenen oder leeren Parametern automatisch der Wert 0 zugewiesen wird. Sie könnten diese Aufgabe mittels boolescher Ausdrücke erledigen, weil die nötige Logik noch keine `if`-Anweisung oder eine Hilfsfunktion rechtfertigt.

Die Python-Syntax macht diese Entscheidung allzu leicht. Der Trick besteht darin, dass sowohl ein leerer String und eine leere Liste als auch die Zahl 0 implizit als `False` bewertet werden. Die nachstehenden Ausdrücke erhalten daher den Wert des Ausdrucks nach dem `or`-Operator, wenn der erste Ausdruck als `False` bewertet wird:

```
# Abfrage-String 'red=5&blue=0&green='
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
print('Rot:        {red!r}')
print('Grün:       {green!r')
print('Deckkraft:  {opacity!r}')
>>>
Rot:       '5'
Grün:      0
Deckkraft: 0
```

Die Abfrage des `red`-Werts funktioniert, weil der Schlüssel im `values`-Dictionary vorhanden ist. Dessen Wert ist eine Liste mit einem einzigen Element, nämlich dem String `'5'`, der implizit als `True` bewertet wird – `red` wird also der erste Teil des `or`-Ausdrucks zugewiesen.

Die Zuweisung von 0 an den `green`-Wert funktioniert, weil der Wert im `values`-Dictionary aus einer Liste besteht, die einen leeren String enthält, der implizit als `False` bewertet wird. Der `or`-Ausdruck wird daher zu 0 ausgewertet.

Die Zuweisung von 0 an den `opacity`-Wert funktioniert, weil der Wert im `values`-Dictionary überhaupt nicht vorhanden ist. Falls der Schlüssel im Dicitionary fehlt, liefert die `get`-Methode ihr zweites Argument als Resultat zurück, in diesem Fall

eine Liste, die nur aus einem leeren String besteht. Wenn also `opacity` nicht im Dictionary gefunden wird, verhält es sich genauso wie beim `green`-Wert.

Allerdings sind die Ausdrücke schwer verständlich und erledigen die Aufgabe auch nicht vollständig, weil nämlich alle Parameter Ganzzahlen sein sollen, damit sie in mathematischen Ausdrücken verwendet werden können. Zu diesem Zweck könnten Sie die Ausdrücke mittels der integrierten `int`-Funktion in Ganzzahlen umwandeln:

```
red = int(my_values.get('red', [''])[0] or 0)
```

Dieser Ausdruck ist nun wirklich schwer lesbar: Er ist voller störender Zeichen (all die Klammern und Anführungszeichen) und unverständlich. Ein Betrachter, der diesen Code erstmals in Augenschein nimmt, bräuchte viel zu lange, um den Ausdruck zu entwirren und herauszufinden, was er eigentlich bewirkt. Es ist zwar durchaus sinnvoll, sich kurzzufassen, aber es lohnt sich nicht, all dies in einer einzigen Zeile unterzubringen.

In Python gibt es die sogenannten ternären Bedingungsoperatoren, also `if/else`-Ausdrücke, mit denen sich Ausdrücke wie dieser zugleich klarer, aber auch kurz und bündig formulieren lassen:

```
red_str = my_values.get('red', [''])
red = int(red_str[0]) if red_str[0] else 0
```

Schon besser. In weniger komplizierten Fällen können `if/else`-Ausdrücke das Verständnis wirklich sehr erleichtern. Das obige Beispiel ist jedoch noch immer nicht so gut verständlich wie eine vollständige, über mehrere Zeilen verteilte `if/else`-Anweisung. Tatsächlich erscheint die dicht gedrängte Version noch komplizierter, wenn man sie mit der auf mehrere Zeilen aufgeteilten Version vergleicht:

```
green_str = my_values.get('green', [''])
if green_str[0]:
    green = int(green_str[0])
else:
    green = 0
```

Sie sollten in solchen Fällen eine Hilfsfunktion programmieren, insbesondere wenn diese Programmlogik mehrmals verwendet wird:

```
def get_first_int(values, key, default=0):
    found = values.get(key, [''])
    if found[0]:
```

```
        return int(found[0])
    return default
```

Der die Hilfsfunktion aufrufende Code ist viel besser verständlich als der komplizierte `or`-Ausdruck oder die zweizeilige Version, die einen `if/else`-Ausdruck verwendet:

```
green = get_first_int(my_values, 'green')
```

Sobald Ausdrücke komplizierter werden, sollten Sie in Betracht ziehen, die Programmlogik in Hilfsfunktionen auszulagern. Die bessere Lesbarkeit bringt fast immer größere Vorteile als knapp gehaltener Code. Lassen Sie sich von Pythons prägnanter Syntax nicht dazu verleiten, derart schwer durchschaubaren Code zu verfassen. Folgen Sie dem *DRY-Prinzip*: Don't repeat yourself (wiederholen Sie sich nicht).

Kompakt

- Pythons Syntax macht es allzu einfach, komplizierte einzeilige Ausdrücke zu programmieren, die schwer verständlich sind.
- Verlagern Sie komplexe Ausdrücke in Hilfsfunktionen, insbesondere dann, wenn Sie die Programmlogik wiederholt verwenden.
- `if/else`-Ausdrücke stellen eine besser verständliche Alternative zur Verwendung boolescher Operatoren wie `or` oder `and` bereit.

Punkt 6 Mehrfachzuweisung beim Entpacken statt Indizierung

Der integrierte Datentyp `tuple` kann zum Erzeugen unveränderlicher, geordneter Sequenzen von Werten verwendet werden. Im einfachsten Fall ist ein `tuple` ein Paar zweier Werte, wie beispielsweise die Schlüssel und Werte eines Dictionarys:

```
snack_calories = {
    'Chips': 140,
    'Popcorn': 80,
    'Nüsse': 190,
}
items = tuple(snack_calories.items())
print(items)
>>>
(('Chips', 140), ('Popcorn', 80), ('Nüsse', 190))
```

Auf die Werte eines Tupels kann man über numerische Indizes zugreifen:

```
item = ('Erdnussbutter', 'Marmelade')
first = item[0]
second = item[1]
print(first, 'und', second)
>>>
Erdnussbutter und Marmelade
```

Nachdem ein Tupel erzeugt wurde, kann es nicht durch Zuweisung eines neuen Wertes an eine Indexposition geändert werden:

```
pair = ('Schokolade', 'Erdnussbutter')
pair[0] = 'Honig'
>>>
Traceback ...
TypeError: 'tuple' object does not support item assignment
```

Python verfügt zudem über eine Syntax zum *Entpacken*, die Mehrfachzuweisungen durch eine einzige Anweisung ermöglicht. Die in der Entpackanweisung angegebenen Muster haben große Ähnlichkeit mit dem Versuch, Tupel zu verändern – was nicht erlaubt ist –, die Funktionsweise unterscheidet sich tatsächlich aber deutlich. Wenn Sie beispielsweise wissen, dass ein Tupel ein Wertepaar ist, können Sie, statt die Indizes zu verwenden, zum Zugriff auf die Werte eine Zuweisung an ein Tupel mit zwei Variablennamen vornehmen:

```
item = ('Erdnussbutter', 'Marmelade')
first, second = item  # Entpacken
print(first, 'und', second)
>>>
Erdnussbutter und Marmelade
```

Das Entpacken ist optisch weniger auffällig als der Zugriff über die Indizes des Tupels und nimmt oft weniger Zeilen in Anspruch. Die gleiche Pattern-Matching-Syntax kann auch für die Zuweisung an Listen, Sequenzen oder mehrfach verschachtelte Objekte verwendet werden. Ich kann nur davon abraten, den folgenden Code zu verwenden, es ist jedoch wichtig zu wissen, dass so etwas möglich ist und wie es funktioniert:

```
favorite_snacks = {
    'salzig': ('Brezeln', 100),
    'süß': ('Kekse', 180),
```

```
    'vegetarisch': ('Karotten', 20),
}
((type1, (name1, cals1)),
 (type2, (name2, cals2)),
 (type3, (name3, cals3))) = favorite_snacks.items()
print(f'Lieblingsessen {type1} sind {name1} mit \
{cals1} Kalorien')
print(f'Lieblingsessen {type2} sind {name2} mit \
 {cals2} Kalorien')
print(f'Lieblingsessen {type3} sind {name3} mit \
 {cals3} Kalorien')
>>>
Lieblingsessen salzig sind Brezeln mit 100 Kalorien
Lieblingsessen süß sind Kekse mit 180 Kalorien
Lieblingsessen vegetarisch sind Karotten mit 20 Kalorien
```

Python-Neulinge sind vielleicht überrascht, dass beim Entpacken sogar Werte vertauscht werden können, ohne temporäre Variablen verwenden zu müssen. Hier verwende ich für einen Algorithmus zum Sortieren in aufsteigender Reihenfolge zunächst die typische Syntax mit Indizes zum Vertauschen der Werte zweier Positionen in einer Liste:

```
def bubble_sort(a):
    for _ in range(len(a)):
        for i in range(1, len(a)):
            if a[i] < a[i-1]:
                temp = a[i]
                a[i] = a[i-1]
                a[i-1] = temp
names = ['Rucola', 'Schinken', 'Brezeln', 'Karotten']
bubble_sort(names)
print(names)
>>>
['Brezeln', 'Karotten', 'Rucola', 'Schinken']
```

Mit der Entpacken-Syntax ist es jedoch auch möglich, die Indizes in einer einzigen Zeile zu vertauschen:

```
def bubble_sort(a):
    for _ in range(len(a)):
        for i in range(1, len(a)):
```

```
            if a[i] < a[i-1]:
                a[i-1], a[i] = a[i], a[i-1] # Vertauschen
names = ['Rucola', 'Schinken', 'Brezeln', 'Karotten']
bubble_sort(names)
print(names)
>>>
['Brezeln', 'Karotten', 'Rucola', 'Schinken']
```

Das Vertauschen funktioniert so: Die rechte Seite der Zuweisung (`a[i], a[i-1]`) wird als Erstes ausgewertet und in einem neuen unbenannten Tupel zwischengespeichert (beispielsweise `('Rucola', 'Schinken')` bei der ersten Iteration der Schleifen). Anschließend wird das Entpack-Muster auf der linken Seite der Zuweisung (`a[i-1], a[i]`) verwendet, um die Werte des Tupels zu erhalten und sie den Variablen `a[i-1]` bzw. `a[i]` zuzuweisen. Dadurch wird beim Index 0 `'Rucola'` durch `'Schinken'` und beim Index 1 `'Schinken'` durch `'Rucola'` ersetzt. Und schließlich verschwindet das temporäre unbenannte Tupel wieder lautlos.

Eine andere nützliche Anwendung des Entpackens sind die Ziellisten von `for`-Schleifen und ähnlichen Konstrukten, wie etwa Listen-Abstraktionen oder Generator-Ausdrücke (siehe Punkt 27: *Listen-Abstraktionen statt map und filter*). Zur Unterscheidung hier ein Beispiel, in dem ich eine Liste von Snacks ohne Entpacken durchlaufe:

```
snacks = [('Schinken', 350), ('Donut', 240), ('Muffin', 190)]
for i in range(len(snacks)):
    item = snacks[i]
    name = item[0]
    calories = item[1]
    print(f'#{i+1}: {name} hat {calories} Kalorien')
>>>
#1: Schinken hat 350 Kalorien
#2: Donut hat 240 Kalorien
#3: Muffin hat 190 Kalorien
```

Das funktioniert zwar, ist aber unschön. Es sind viele zusätzliche Zeichen erforderlich, um auf die verschiedenen Ebenen der `snacks`-Struktur über Indizes zuzugreifen. Hier erziele ich das gleiche Ergebnis, indem ich Entpacken zusammen mit der integrierten `enumerate`-Funktion verwende (siehe Punkt 7: *enumerate statt range*):

```
for rank, (name, calories) in enumerate(snacks, 1):
    print(f'#{rank}: {name} hat {calories} Kalorien')
```

```
>>>
#1: Schinken hat 350 Kalorien
#2: Donut hat 240 Kalorien
#3: Muffin hat 190 Kalorien
```

Das ist die für Python typische Art und Weise, eine solche Schleife zu programmieren: Sie lässt sich schnell und einfach verstehen. Für gewöhnlich ist es nicht erforderlich, für den Zugriff auf irgendetwas Indizes zu verwenden.

Python bietet die Funktionalität des Entpackens unter anderem auch für Listen (siehe Punkt 13: *Vollständiges Entpacken statt Slicing*), Funktionsargumente (siehe Punkt 22: *Klare Struktur dank variabler Anzahl von Positionsargumenten*), Schlüsselwortargumente (siehe Punkt 23: *Optionale Funktionalität durch Schlüsselwort-Argumente*) und mehrere Rückgabewerte (siehe Punkt 19: *Bei Funktionen mit mehreren Rückgabewerten nicht mehr als drei Variablen entpacken*).

Die richtige Nutzung des Entpackens erlaubt es, Zugriffe über Indizes möglichst zu vermeiden, was zu besser verständlichem Code führt.

Kompakt

- In Python gibt es eine spezielle Syntax zum Entpacken, mit der in einer einzigen Anweisung mehrere Wertzuweisungen vorgenommen werden können.
- Das Entpacken ist in Python allgemein verfügbar und kann auf alle abzählbaren Objekte, auch mehrfach verschachtelte, angewendet werden.
- Durch das Entpacken wird die optische Auffälligkeit verringert und die Verständlichkeit des Codes verbessert, weil sich so der explizite Zugriff über Indizes vermeiden lässt.

Punkt 7 enumerate statt range

Die integrierte `range`-Funktion kann für Schleifen verwendet werden, die einen Zahlenbereich durchlaufen:

```
from random import randint
random_bits = 0
for i in range(32):
    if randint(0, 1):
        random_bits |= 1 << i
>>>
0b11101000100100000111000010000001
```

Wenn Sie eine Datenstruktur wie eine Liste mit Strings durchlaufen möchten, ist dies unmittelbar möglich:

```
flavor_list = ['Vanille', 'Schoko', 'Nuss', 'Erdbeer']
for flavor in flavor_list:
    print('%s ist lecker' % flavor)
>>>
Vanille ist lecker
Schoko ist lecker
Nuss ist lecker
Erdbeer ist lecker
```

Es kommt häufiger vor, dass man eine Liste durchlaufen möchte und der Index des aktuellen Elements dabei bekannt ist. Wenn Sie beispielsweise die Reihenfolge Ihrer Lieblingseissorten ausgeben möchten, können Sie dazu `range` verwenden:

```
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print(f'{i + 1}: {flavor}')
>>>
1: Vanille
2: Schoko
3: Nuss
4: Erdbeer
```

Im Vergleich zu den anderen Beispielen, die über `flavor_list` oder `range` iterieren, sieht das ein wenig unbeholfen aus. Sie müssen erst die Länge der Liste ermitteln und über den Index auf die Elemente des Arrays zugreifen – es liest sich einfach nicht gut.

Für solche Aufgaben gibt es in Python die integrierte `enumerate`-Funktion, die dem Iterator einen Generator bereitstellt, der Paare aus Schleifenindex und dem nächsten Wert des zugehörigen Iterators liefert (siehe Punkt 30: *Generatoren statt Rückgabe von Listen*). Hier verwende ich die integrierte `next`-Funktion, um zu demonstrieren, wie das abläuft:

```
it = enumerate(flavor_list)
print(next(it))
print(next(it))
>>>
(0, 'Vanille')
(1, 'Schoko')
```

Die von `enumerate` gelieferten Wertepaare können in einer `for`-Schleife entpackt werden (siehe Punkt 6: *Mehrfachzuweisung beim Entpacken statt Indizierung*). Der Code ist viel klarer:

```
for i, flavor in enumerate(flavor_list):
    print(f'{i + 1}: {flavor}')
>>>
1: Vanille
2: Schoko
3: Nuss
4: Erdbeer
```

Das lässt sich weiter verkürzen, indem man angibt, bei welchem Wert (in diesem Fall 1) `enumerate` mit der Zählung beginnen soll:

```
for i, flavor in enumerate(flavor_list, 1):
    print(f'{i}: {flavor}')
```

Kompakt

- `enumerate` bietet eine kompakte Syntax zum Durchlaufen eines Iterators und stellt den Index des jeweiligen Elements zur Verfügung.
- Verwenden Sie statt `range` oder Indizes lieber `enumerate`.
- Sie können `enumerate` einen zweiten Parameter übergeben, der festlegt, bei welchem Wert die Zählung beginnt (Standardwert ist 0).

Punkt 8 Gleichzeitige Verarbeitung von Iteratoren mit zip

In Python hat man es häufig mit mehreren Listen ähnlicher Objekte zu tun. Mittels Listen-Abstraktionen können Sie durch Anwendung eines Ausdrucks auf die Elemente einer Liste leicht eine weitere Liste erzeugen (siehe Punkt 27: *Listen-Abstraktionen statt map und filter*).

```
names = ['Cecilia', 'Lise', 'Marie']
countss = [len(n) for n in names]
print(counts)
>>>
{7, 4, 5
```

Die Elemente der abgeleiteten Liste sind mit den ursprünglichen Elementen durch ihren Index miteinander verknüpft. Um beide Listen gleichzeitig zu verarbeiten, kann ich über die Länge der Liste `names` iterieren:

```
longest_name = None
max_count = 0
for i in range(len(names)):
    count = counts[i]
    if count > max_count:
        longest_name = names[i]
        max_count = count

print(longest_name)
>>>
Cecilia
```

Das Problem ist hier die unschöne Schleife. Die Zugriffe auf die `names`- und `counts`-Arrays über die Indizes machen den Code schwer lesbar. Der Schleifenindex `i` wird gleich zweimal benutzt. Das lässt sich durch den Einsatz von `enumerate` etwas verbessern (siehe Punkt 7: *enumerate statt range*), aber ideal ist das noch lange nicht.

```
for i, name in enumerate(names):
    count = counts[i]
    if count > max_count:
        longest_name = name
        max_count = count
```

Code wie dieser lässt sich mittels der in Python integrierten `zip`-Funktion vereinfachen. Sie stellt für zwei oder mehr Iteratoren einen Generator bereit. Dieser `zip`-Generator liefert Tupel mit den jeweils nächsten Werten der Iteratoren. Diese Tupel können unmittelbar in einer `for`-Schleife entpackt werden (siehe Punkt 6: *Mehrfachzuweisung beim Entpacken statt Indizierung*). Der resultierende Code ist erheblich besser verständlich als die Verwendung von Indizes für mehrere Listen.

```
for name, count in zip(names, counts):
    if count > max_count:
        longest_name = name
        max_count = count
```

`zip` wertet die übergebenen Iteratoren der Reihe nach einzeln aus, was bedeutet, dass beliebig lange Eingaben verwendet werden können, ohne dass man Gefahr läuft, dass es zu großem Speicherbedarf kommt, der das Programm zum Absturz bringt.

Allerdings verhält sich die `zip`-Funktion eigentümlich, wenn die zu den Iteratoren zugehörigen Listen unterschiedlich lang sind. Wenn ich beispielsweise der Liste `names` einen weiteren Namen hinzufüge, dabei aber übersehe, auch die Liste `counts` zu aktualisieren, liefert die Ausführung der `zip`-Funktion ein unerwartetes Ergebnis:

```
names.append('Rosalinde')
for name, count in zip(names, counts):
print(name)
>>>
Cecilia
Lise
Marie
```

Der neue Name »Rosalinde« fehlt. Das liegt an der Funktionsweise der `zip`-Funktion: Sie liefert Tupel zurück, bis irgendeiner der übergebenen Iteratoren abgearbeitet ist. Die Ausgabe ist genau so lang wie die kürzeste Eingabe. Dieser Ansatz funktioniert gut, sofern feststeht, dass die Listen gleich lang sind – was bei durch Listen-Abstraktionen erzeugten Listen meist zutrifft.

In vielen anderen Fällen ist dieses Verhalten der `zip`-Funktion jedoch überraschend und ärgerlich. Falls Sie sich nicht sicher sind, dass die von der `zip`-Funktion zu verarbeitenden Listen von gleicher Länge sind, sollten Sie in Betracht ziehen, die `zip_longest`-Funktion des integrierten `itertools`-Moduls zu verwenden:

```
import itertools

for name, count in itertools.zip_longest(names, counts):
    print(f'{name}: {count}')
>>>
Cecilia: 7
Lise: 4
Marie: 5
Rosalinde: None
```

`zip_longest` ersetzt fehlende Werte – in diesem Fall die Länge des Strings »Rosalinde« – durch einen übergebenen Wert `fillvalue`, für den standardmäßig `None` verwendet wird.

Kompakt

- Zur gleichzeitigen Verarbeitung mehrerer Iteratoren kann die integrierte `zip`-Funktion verwendet werden.
- Die `zip`-Funktion ist ein Generator, der Tupel erzeugt. Sie kann also für beliebig lange Eingaben verwendet werden.
- Die `zip`-Funktion bricht die Ausgabe stillschweigend ab, falls Sie ihr Iteratoren für Listen unterschiedlicher Länge übergeben.
- Die `zip_longest`-Funktion des integrierten `itertools`-Moduls ermöglicht die gleichzeitige Verarbeitung mehrerer Iteratoren unabhängig von der Länge der zugehörigen Listen.

Punkt 9 Verzicht auf else-Blöcke nach for- und while-Schleifen

Die in Python verfügbaren Schleifen besitzen eine Funktionalität, die in den meisten anderen Programmiersprachen nicht vorhanden ist: Sie können unmittelbar am Ende des Schleifenkörpers einen `else`-Block platzieren:

```
for i in range(3):
    print('Schleife %d' % i)
else:
    print('Else-Block!')
>>>
Schleife 0
Schleife 1
Schleife 2
Else-Block!
```

Überraschenderweise wird er nach Beendigung der Schleife sofort ausgeführt. Warum nur lautet die Bezeichnung für diesen Block »else« und nicht »and«? Bei einer `if/else`-Anweisung bedeutet `else`: »Führe diesen Block aus, sofern der vorhergehende nicht ausgeführt wurde.« Und auch bei einer `try/except`-Anweisung bedeutet `except`: »Führe diesen Block aus, sofern die Ausführung des `try`-Blocks fehlschlägt.«

Das `else` in einer `try/except/else`-Anweisung verhält sich ebenfalls nach diesem Muster (siehe Punkt 65: *Alle Blöcke einer try/except/else/finally-Anweisung nutzen*), denn es bedeutet: »Führe diesen Block aus, sofern der vorhergehende nicht fehlgeschlagen ist.« Und schließlich ist auch eine `try/finally`-Anweisung intuitiv, da sie besagt: »Führe stets diesen abschließenden Block aus, nachdem die Ausführung des vorhergehenden Blocks versucht wurde.«

Angesichts all dieser Verhaltensweisen könnte ein Programmierneuling auf den Gedanken kommen, dass der `else`-Block einer `for/else`-Anweisung die Bedeutung hat: »Führe diesen Block aus, wenn die Schleife nicht vollständig durchlaufen wurde.« Tatsächlich bewirkt sie das Gegenteil. Eine `break`-Anweisung in der Schleife sorgt dafür, dass der `else`-Block übersprungen wird:

```
for i in range(3):
    print('Schleife', i)
    if i == 1:
        break
else:
    print('Else-Block!')
>>>
Schleife 0
Schleife 1
```

Hier gibt es noch weitere Überraschungen: Wenn man eine leere Liste »durchläuft«, wird der `else`-Block ebenfalls ausgeführt:

```
for x in []:
    print('Wird nie ausgeführt.')
else:
    print('Else-Block der for-Schleife!')
>>>
Else-Block der for-Schleife!
```

Der `else`-Block wird auch ausgeführt, wenn die Bedingung einer `while`-Schleife eingangs den Wert `False` besitzt:

```
while False:
    print('Wird nie ausgeführt.')
else:
    print('Else-Block der while-Schleife!')
>>>
Else-Block der while-Schleife!
```

Der Grund für dieses Verhalten ist, dass `else`-Blöcke nach Schleifen nützlich sind, wenn in der Schleife etwas gesucht wird, beispielsweise wenn Sie feststellen möchten, ob zwei Zahlen teilerfremd sind (der einzige gemeinsame Teiler ist 1). Sie durchlaufen dann alle möglichen Teiler und überprüfen die Zahlen. Nachdem sämtliche Möglichkeiten durchprobiert wurden, endet die Schleife. Der `else`-

Block wird nur ausgeführt, wenn beide Zahlen tatsächlich teilerfremd sind, da in diesem Fall keine `break`-Anweisung in der Schleife ausgeführt wurde.

```
a=4
b=9
for i in range(2, min(a, b) + 1):
    print('Teste ', i)
    if a % i == 0 and b % i == 0:
        print('Nicht teilerfremd')
        break
else:
    print('Teilerfremd')
>>>
Teste 2
Teste 3
Teste 4
Teilerfremd
```

In der Praxis würde ich das sicher nicht so programmieren, sondern eine Hilfsfunktion für die Berechnung verwenden. Solche Hilfsfunktionen folgen üblicherweise den beiden folgenden Mustern. Beim ersten Ansatz wird die Funktion sofort beendet, wenn die gesuchte Bedingung erfüllt ist. Anderenfalls wird ein Standardwert zurückgeliefert:

```
def coprime(a, b):
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            return False
    return True
```

Das zweite Verfahren verwendet eine Variable, die anzeigt, ob das Gesuchte gefunden wurde. Wenn dieser Fall eintritt, wird die Schleife mittels `break` beendet:

```
def coprime_alternate(a, b):
    is_coprime = True
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            is_coprime = False
            break
    return is_coprime
assert coprime_alternate(4, 9)
assert not coprime_alternate(3, 6)
```

Beide Vorgehensweisen sind für Betrachter, denen der Code nicht vertraut ist, erheblich besser verständlich. Den Umständen entsprechend kann der eine oder der andere Ansatz eine gute Wahl sein. Die durch den `else`-Block gewonnene Ausdrucksfähigkeit wird von der Mühe, die Sie sich selbst und anderen beim Betrachten des Codes auferlegen, in den Schatten gestellt. Was einfache Konstrukte wie Schleifen bewirken, sollte in Python offensichtlich sein. Sie sollten daher auf `else`-Anweisungen nach Schleifen am besten komplett verzichten.

Kompakt

- Python verfügt über eine spezielle Syntax, die es erlaubt, `else`-Blöcke unmittelbar hinter `for`- und `while`-Schleifen zu platzieren.
- Dieser `else`-Block wird nur ausgeführt, wenn in der Schleife keine `break`-Anweisung stattfindet.
- Verzichten Sie lieber auf `else`-Blöcke nach Schleifen, da ihr Verhalten nicht intuitiv ist und verwirrend sein kann.

Punkt 10 Wiederholungen verhindern durch Zuweisungsausdrücke

Ein *Zuweisungsausdruck*, der mitunter auch als *Walross-Operator* bezeichnet wird, ist eine neue Syntax, die mit der Python-Version 3.8 eingeführt wurde, um ein schon lange bestehendes Problem zu lösen, das zu doppeltem Code führen kann. Bei einer normalen Zuweisung schreibt man `a = b` und sagt »a gleich b«, bei dieser neuen Zuweisung hingegen schreibt man `a := b` und sagt »a Walross b« (weil die Zeichen »:=« wie die Augen und Stoßzähne eines Walrosses aussehen).

Nehmen wir beispielsweise an, ich möchte den Inhalt eines Korbs frischen Obsts für eine Obstsaftbar verwalten. Hier definiere ich den Inhalt:

```
fresh_fruit = {
    'Apfel': 10,
    'Banane': 8,
    'Zitrone': 5, }
```

Wenn ein Kunde an die Theke kommt, um eine Zitronenlimonade zu bestellen, muss gewährleistet sein, dass im Korb mindestens eine Zitrone vorhanden ist. Hier rufe ich die Anzahl der Zitronen ab und verwende eine `if`-Anweisung, um zu überprüfen, dass der Wert nicht 0 ist:

```
def make_lemonade(count):
    ...
```

```
def out_of_stock():
    ...

count = fresh_fruit.get('Zitrone', 0)
if count:
    make_lemonade(count)
else:
    out_of_stock()
```

Dieser scheinbar einfache Code ist jedoch umständlicher als nötig. Die Variable `count` wird nur im ersten Block der `if`-Anweisung verwendet. Die Variable oberhalb der `if`-Anweisung zu definieren, lässt sie wichtiger erscheinen, als sie tatsächlich ist, weil man den Eindruck bekommt, dass der nachfolgende Code, einschließlich des `else`-Blocks, darauf zugreifen müsste, obwohl das gar nicht der Fall ist.

Diese Vorgehensweise, einen Wert abzurufen, dann zu überprüfen, ob er von null verschieden ist, und ihn anschließend zu verwenden, kommt in Python extrem häufig vor. Manche Programmierer versuchen, mehrfache Verweise auf `count` durch eine Reihe von Tricks zu umgehen, die jedoch der Verständlichkeit abträglich sind (siehe Punkt 5: *Hilfsfunktionen statt komplizierter Ausdrücke*). Erfreulicherweise wurde die Sprache durch Zuweisungsausdrücke ergänzt, um genau diese Art von Code zu optimieren. Hier verwende ich für das letzte Beispiel den Walross-Operator:

```
if count := fresh_fruit.get('Zitrone', 0):
    make_lemonade(count)
else:
    out_of_stock()
```

Das ist zwar nur eine Zeile weniger, aber viel besser verständlich, weil nun deutlich wird, dass `count` nur für den ersten Block der `if`-Anweisung von Bedeutung ist. Der Zuweisungsausdruck weist der Variablen `count` zunächst einen Wert zu und wertet ihn anschließend im Kontext der `if`-Anweisung aus, um zu ermitteln, wie fortzufahren ist. Diese beiden Schritte – Zuweisung und anschließende Auswertung – stellen die grundsätzliche Verhaltensweise des Walross-Operators dar.

Zitronen sind sehr ergiebig, sodass ich für mein Zitronenlimonadenrezept nur eine benötige. Deshalb reicht es aus, zu überprüfen, ob die Anzahl größer als 0 ist. Falls ein Kunde einen Apfelsaft bestellt, muss ich mich vergewissern, dass mindestens vier Äpfel vorhanden sind. Das erledige ich, indem ich `count` vom Dictionary `fresh_fruit` abfrage und in der `if`-Anweisung einen Vergleich vornehme:

```
def make_cider(count):
    ...
count = fresh_fruit.get('Apfel', 0)
if count >= 4:
    make_cider(count)
else:
    out_of_stock()
```

Hier tritt das gleiche Problem wie beim vorherigen Beispiel auf, nämlich dass die Zuweisung zu `count` die Aufmerksamkeit zu Unrecht auf sich zieht. Ich verbessere die Verständlichkeit des Codes, indem ich wieder den Walross-Operator verwende:

```
if (count := fresh_fruit.get('Apfel', 0)) >= 4:
    make_cider(count)
else:
    out_of_stock()
```

Das funktioniert wie erwartet und verkürzt den Code um eine Zeile. Hier ist es wichtig, darauf hinzuweisen, dass ich den Zuweisungsausdruck in Klammern einschließen muss, um ihn in der `if`-Anweisung mit 4 vergleichen zu können. Beim Zitronenlimonaden-Beispiel war das nicht erforderlich, weil der Zuweisungsausdruck selbst einen Test auf die Verschiedenartigkeit von 0 darstellt und nicht Teil eines übergeordneten Ausdrucks war. Ebenso wie bei anderen Ausdrücken sollten Sie vermeiden, Zuweisungsausdrücke in Klammern einzuschließen, sofern das möglich ist.

Eine weitere Variante dieses häufigen Musters tritt auf, wenn es erforderlich ist, in Abhängigkeit von einer Bedingung eine Variablenzuweisung innerhalb des umschließenden Gültigkeitsbereichs vorzunehmen und kurz danach in einem Funktionsaufruf darauf zuzugreifen. Nehmen wir beispielsweise an, ein Kunde bestellt ein paar Bananen-Smoothies. Dann müssen mindestens zwei Bananen vorhanden sein, sonst wird eine `OutOfBananas`-Exception ausgelöst. Hier implementiere ich dieses Verhalten auf typische Weise:

```
def slice_bananas(count):
    ...

class OutOfBananas(Exception):
    pass

def make_smoothies(count):
```

```
    ...

pieces = 0
count = fresh_fruit.get('Banane', 0)
if count >= 2:
    pieces = slice_bananas(count)

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

Die Zuweisung `pieces = 0` in den `else`-Block zu verschieben, ist eine weitere Möglichkeit, diese Aufgabe zu erledigen:

```
count = fresh_fruit.get('Banane', 0)
if count >= 2:
    pieces = slice_bananas(count)
else:
    pieces = 0

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

Der zweite Ansatz wirkt etwas merkwürdig, weil die Variable `pieces` jetzt an zwei Stellen vorkommt – in jedem Block der `if`-Anweisung –, an denen sie erstmals definiert werden kann. Rein technisch gesehen funktioniert das dank Pythons Regeln für Gültigkeitsbereiche (siehe Punkt 21: *Closures und der Gültigkeitsbereich von Variablen*). Es ist allerdings nicht so gut verständlich oder auffindbar, deshalb bevorzugen die meisten Leute die vorherige Vorgehensweise, bei der die Zuweisung `pieces = 0` zuerst erfolgt.

Der Walross-Operator kann erneut dazu verwendet werden, das Beispiel um eine Zeile zu verkürzen. Diese kleine Änderung beseitigt jegliche Hervorhebung der `count`-Variablen. Jetzt ist klar, dass `pieces` nicht nur in der `if`-Anweisung von Bedeutung ist:

```
pieces = 0
if (count := fresh_fruit.get('Banane', 0)) >= 2:
    pieces = slice_bananas(count)
```

```
try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

Die Verwendung des Walross-Operators verbessert auch die Verständlichkeit der verteilten Definition von `pieces` in den beiden Teilen der `if`-Anweisung. Es ist einfacher, die Variable `pieces` im Auge zu behalten, wenn die Definition von `count` der `if`-Anweisung nicht mehr vorausgeht:

```
if (count := fresh_fruit.get('Banane', 0)) >= 2:
    pieces = slice_bananas(count)
else:
    pieces = 0

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

Python-Einsteiger sind oft frustriert, weil es keine flexible `switch/case`-Anweisung gibt. Um eine ähnliche Funktionalität zu erzielen, muss man im Allgemeinen tief verschachtelte `if`-, `elif`- und `else`-Anweisungen verwenden.

Nehmen wir beispielsweise an, ich möchte ein Priorisierungssystem einrichten, damit alle Kunden automatisch den besten verfügbaren Fruchtsaft bekommen, ohne ihn bestellen zu müssen. Hier definiere ich die Programmlogik so, dass zuerst Bananen-Smoothies serviert werden, dann Apfelsaft und zum Schluss Zitronenlimonade:

```
count = fresh_fruit.get('Banane', 0)
if count >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
else:
    count = fresh_fruit.get('Apfel', 0)
    if count >= 4:
        to_enjoy = make_cider(count)
    else:
        count = fresh_fruit.get('Zitrone', 0)
        if count:
```

```
            to_enjoy = make_lemonade(count)
        else:
            to_enjoy'= 'Nichts'
```

Unschöne Konstrukte wie dieses findet man überraschend oft in Python-Code. Erfreulicherweise bietet der Walross-Operator eine elegante Lösung, die fast so vielseitig wie eine Syntax für `switch/case`-Anweisungen ist:

```
if (count := fresh_fruit.get('Banane', 0)) >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
elif (count := fresh_fruit.get('Apfel', 0)) >= 4:
    to_enjoy = make_cider(count)
elif count := fresh_fruit.get('Zitrone', 0):
    to_enjoy = make_lemonade(count)
else:
    to_enjoy = 'Nichts'
```

Diese Version, die Zuweisungsausdrücke verwendet, ist nur fünf Zeilen kürzer als das Original, aber weniger verschachtelt und nicht so häufig eingerückt und dadurch viel besser verständlich. Wenn Sie feststellen, dass sich solche unschönen Konstrukte in Ihren Code einschleichen, dann sollten Sie versuchen, sie mithilfe des Walross-Operators umzuformulieren, sofern das möglich ist.

Python-Einsteiger sind zudem auch oft frustriert, weil es keine `do/while`-Schleife gibt. Nehmen wir an, ich möchte nach einer Obstlieferung Fruchtsaft in Flaschen abfüllen, bis kein Obst mehr da ist. Hier implementiere ich diese Logik mit einer `while`-Schleife:

```
def pick_fruit():
    ...
def make_juice(fruit, count):
    ...
bottles = []
fresh_fruit = pick_fruit()
while fresh_fruit:
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
    fresh_fruit = pick_fruit()
```

Hier gibt es eine Wiederholung, denn `fresh_fruit = pick_fruit()` wird an zwei verschiedenen Stellen aufgerufen: vor der Schleife, um die Anfangsbedingungen einzurichten, und nach der Schleife, um die Liste für das gelieferte Obst wieder aufzustocken.

Um hier die Wiederholung zu umgehen, kann man die Strategie einer »Anderthalbfach-Schleife« verfolgen, wodurch allerdings der Beitrag der `while`-Schleife konterkariert wird, die nur noch eine simple Endlosschleife ist. Der Programmablauf hängt nur noch von der Bedingung für die `break`-Anweisung ab:

```
bottles = []
while True:                    # Eine Schleife
    fresh_fruit = pick_fruit()
    if not fresh_fruit:        # und eine 'halbe'
        break
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
```

Der Walross-Operator macht dieses Konstrukt überflüssig, denn er ermöglicht bei jedem Durchlaufen der `while`-Schleife eine Neuzuweisung zur `fresh_fruit`-Variablen und eine erneute Auswertung der Bedingung. Diese Lösung ist kurz und bündig, gut verständlich und sollte in Ihrem Code der bevorzugte Ansatz sein:

```
bottles = []
while fresh_fruit := pick_fruit():
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
```

Es gibt eine Vielzahl weiterer Situationen, in denen Zuweisungsausdrücke verwendet werden können, um Redundanz zu beseitigen (siehe beispielsweise Punkt 29: *Doppelte Arbeit in Abstraktionen durch Zuweisungsausdrücke vermeiden*). Wenn Sie den gleichen Ausdruck oder die gleiche Zuweisung innerhalb weniger Zeilen mehrfach verwenden, sollten Sie in Betracht ziehen, Zuweisungsausdrücke zu verwenden, um die Verständlichkeit zu verbessern.

Kompakt

- Zuweisungsausdrücke verwenden den Walross-Operator (`:=`) zur Zuweisung und Auswertung von Variablen in einem einzigen Ausdruck und verhindern so Wiederholungen.

- Wenn ein Zuweisungsausdruck Teil eines umfassenderen übergeordneten Ausdrucks ist, muss er in Klammern eingeschlossen werden.
- `switch/case`-Anweisungen und `do/while`-Schleifen sind in Python zwar nicht verfügbar, ihre Funktionalität lässt sich jedoch viel verständlicher durch die Verwendung von Zuweisungsausdrücken emulieren.

Stichwortverzeichnis