Dan Bader

Python Tricks

Praktische Tipps
für Fortgeschrittene

dpunkt.verlag

Dan Bader

Python-Tricks

Praktische Tipps für Fortgeschrittene

dpunkt.verlag

Dan Bader
mail@dbader.org

Lektorat: Melanie Feldmann
Übersetzung: G&U Language & Publishing Services GmbH, Flensburg (www.GundU.com)
Copy-Editing: Sandra Gottmann, Münster-Nienberge
Satz: G&U Language & Publishing Services GmbH, Flensburg (www.GundU.com)
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

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 978-3-86490-568-1
PDF 978-3-96088-599-3
ePub 978-3-96088-600-6
mobi 978-3-96088-601-3

1. Auflage 2018

Wieblinger Weg 17
69123 Heidelberg

Title of the English original: Python Tricks: A Buffet of Awesome Python Features
ISBN 978-1775093305

5 4 3 2 1 0

Inhalt

Danksagung

Vielen herzlichen Dank an die Betaleser der deutschen Übersetzung:

Jürgen Gmach
Marc Richter
Jan Wagner
Oliver Kraitschy
Alexander Junge
Pascal Bawidamann
Peer Wagner

Euer Feedback war super hilfreich!

Vorwort

Es ist fast zehn Jahre her, seit ich meine erste Begegnung mit Python als Programmiersprache hatte. Python lernte ich zunächst nur widerwillig. Ich hatte bereits in verschiedenen anderen Sprachen programmiert, als ich bei der Arbeit plötzlich einem anderen Team zugeteilt wurde, dessen Mitglieder allesamt Python verwendeten. Das war der Anfang meiner eigenen Erfahrungen mit Python.

Bei meiner Einführung in Python hatte man mir gesagt, dass es leicht sei und dass ich es sehr schnell begreifen würde. Als ich meine Kollegen um Lernmaterial bat, gaben sie mir lediglich einen Link zur offiziellen Python-Dokumentation. Die Lektüre dieser Dokumentation war zu Anfang ziemlich verwirrend, und ich brauchte einige Zeit, bevor ich mich auch nur in der Dokumentation zurechtfand. Sehr oft musste ich auf Stack Overflow nach Antworten suchen.

Da ich schon Erfahrungen mit anderen Programmiersprachen hatte, brauchte ich keine Einführung in die Programmierung und auch keine Erklärungen, was Klassen und Objekte sind. Ich suchte nach Quellen, die konkret die Merkmale von Python beschrieben, die Besonderheiten aufzeigten und mir sagten, wie sich das Programmieren in Python vom Programmieren in anderen Sprachen unterscheidet.

Es hat viele Jahre gedauert, Python richtig schätzen zu lernen. Als ich Dans Buch las, wünschte ich mir, ich hätte es schon zur Hand gehabt, als ich vor vielen Jahren Python lernte.

Um ein Beispiel zu geben: Eines der vielen besonderen Python-Merkmale, die mich zu Anfang überraschten, war die Listennotation. Wie Dan in diesem Buch erwähnt, ist die Gestaltung von `for`-Schleifen etwas, woran man jemanden erkennen kann, der von einer anderen Programmiersprache zu Python gewechselt ist. Ich weiß noch, dass einer der ersten Kommentare, die ich zu Anfang meiner Python-Zeit bei einem Code Review erhielt, lautete: »Warum verwendest du hier keine Listennotation?« Dan erklärt diese Vorgehensweise sehr gut in Kapitel 6, wobei er damit beginnt, wie Schleifen auf pythonische Weise geschrieben werden, und sich dann zu Iteratoren und Generatoren vorarbeitet.

In Kapitel 2.5 erklärt Dan die verschiedenen Möglichkeiten zur Stringformatierung in Python. Die Stringformatierung ist eines der Dinge, die dem *Zen of Python* widersprechen, laut dem es nur eine offensichtliche Möglichkeit geben soll, etwas zu tun. Dan führt die verschiedenen Möglichkeiten auf, darunter auch meine Lieblingsergänzung der Sprache, nämlich F-Strings, und nennt dabei auch die Vor- und Nachteile der einzelnen Methoden.

Das Kapitel über Techniken zur Produktivitätssteigerung ist eine weitere hervorragende Quelle. Sie deckt Aspekte ab, die über die Programmiersprache Python hinausgehen, und gibt Tipps für das Debugging und die Verwaltung von Abhängigkeiten. Außerdem erhältst du hier einen kleinen Einblick in Python-Bytecode.

Es ist mir wirklich eine Ehre und eine Freude, die Einführung zu diesem Buch von meinem Freund Dan Bader schreiben zu dürfen.

Durch meine Beiträge zu Python als CPython-Hauptentwicklerin habe ich Verbindungen zu vielen Mitgliedern der Community. Auf meinem Weg mit Python habe ich Mentoren, Verbündete und viele neue Freunde gefunden. Sie machen mir klar, dass es bei Python nicht nur um den Code geht: Python ist eine Gemeinschaft.

Die Programmierung mit Python zu meistern, bedeutet nicht nur, die theoretischen Aspekte der Sprache zu verstehen. Genauso wichtig ist es auch, die Konventionen und empfohlenen Vorgehensweisen, die von der Community genutzt werden, zu verstehen und zu übernehmen.

Dans Buch hilft dir dabei. Ich bin überzeugt, dass du nach dieser Lektüre Python-Programme mit viel mehr Selbstvertrauen schreiben wirst.

– Mariatta Wijaya, Python-Core-Entwicklerin (*mariatta.ca*)

1 Einführung

1.1 Was sind Python-Tricks?

> ***Python-Trick:*** *Ein kurzes Python-Codefragment, das als Lernmittel gedacht ist. Ein Python-Trick dient entweder zur einfachen Veranschaulichung eines Aspekts von Python oder als motivierendes Beispiel, um es Dir zu ermöglichen, tiefer zu schürfen und ein intuitives Verständnis zu entwickeln.*

Der Ausgangspunkt dieses Buches war eine kurze Reihe von Codeausschnitten, die ich eine Woche lang auf Twitter veröffentlicht hatte. Zu meiner Überraschung ernteten sie begeisterte Rückmeldungen und wurden noch tagelang weitergegeben.

Immer mehr Entwickler fragten mich, wo sie »die komplette Reihe« bekommen könnten. In Wirklichkeit hatte ich einfach nur einige wenige Tricks zu verschiedenen Python-Themen zusammengestellt. Dahinter steckte kein umfassendes Konzept; es war einfach nur ein kleines Twitter-Experiment.

Diese Anfragen machten mir jedoch klar, dass es sich lohnen würde auszuprobieren, meine kurzen Codebeispiele als Lernmittel zu verwenden. So stellte ich eine Reihe weiterer Python-Tricks zusammen und veröffentlichte sie in Form einer Serie von E-Mails. Nach nur wenigen Tagen hatten sich mehrere Hundert Python-Entwickler dafür registriert. Von dieser Reaktion war ich schier überwältigt.

In den folgenden Tagen und Wochen erhielt ich einen nicht enden wollenden Zustrom an Rückmeldungen von Python-Entwicklern. Sie dankten mir dafür, dass ich ihnen Aha-Erlebnisse über Aspekte der Sprache bescherte, mit deren Verständnis sie zu kämpfen hatten. Es war großartig, diese Kommentare zu hören. Während ich die Python-Tricks lediglich für Codeausschnitte gehalten hatte, zogen viele Entwickler großen Nutzen daraus.

Zu diesem Zeitpunkt entschied ich, mein Python-Tricks-Experiment zu forcieren und es zu einer Serie von ca. 30 E-Mails auszubauen. Jede dieser Mails bestand nur aus einer Überschrift und einem Screenshot mit einem Codeausschnitt, aber schon bald erkannte ich die Einschränkungen, die dieses Format mit sich brachte. In einer E-Mail brachte ein blinder Python-Entwickler seine Enttäuschung darüber zum Ausdruck, dass die Python-Tricks als Bilder geliefert wurden, sodass er sie mit seinem Screenreader nicht lesen konnte.

Um mein Projekt ansprechender zu gestalten und einem größeren Publikum zugänglich zu machen, musste ich also mehr Zeit darin investieren. Also setzte ich mich hin und schrieb die ganze Serie von Python-Tricks-E-Mails in Textformat mit übersichtlicher Syntaxhervorhebung in HTML um. Diese neue Inkarnation der Python-Tricks lief eine Weile sehr gut. Laut den Rückmeldungen, die ich bekam, waren die Entwickler froh darüber, dass sie die Codebeispiele nun einfach kopieren konnten, um selbst damit herumzuspielen.

Als sich immer mehr Entwickler für diese E-Mail-Serie registrierten, begann ich in den Kommentaren und Fragen, die mir geschickt wurden, ein Muster zu erkennen. Einige Tricks funktionierten als motivierende Beispiele sehr gut für sich allein. Bei den komplizierteren Beispielen fehlte jedoch eine erklärende Stimme, um die Leser anzuleiten oder sie auf weitere Quellen hinzuweisen, sodass sie ein tieferes Verständnis gewinnen konnten.

Dies war ein weiterer Bereich, in dem Verbesserungen notwendig waren. Der Leitspruch von *dbader.org* lautet: »Python-Entwicklern helfen, noch großartiger zu werden.« Hier bot sich mir offensichtlich eine Gelegenheit, um diesem Motto gerecht zu werden.

Ich entschied mich, auf der Grundlage der besten und wertvollsten Python-Tricks meines E-Mail-Kurses ein neuartiges Python-Buch zu schreiben:

- Ein Buch, das die faszinierendsten Aspekte der Sprache anhand von kurzen und leicht verständlichen Beispielen erklärt
- Ein Buch, das eine Reihe großartiger Python-Merkmale aufzeigt und die Motivation der Leser stets auf hohem Niveau hält
- Ein Buch, das dich an die Hand nimmt und anleitet, um dir zu helfen, dein Verständnis von Python zu vertiefen

Dieses Buch war mir eine Herzensangelegenheit und auch ein enormes Experiment. Ich hoffe, dass du bei der Lektüre viel Freude hast und etwas über Python lernst.

– Dan Bader

1.2 Der Zweck dieses Buches

Dieses Buch soll aus dir einen besseren – einen erfolgreicheren und kenntnisreicheren – Python-Programmierer machen. Vielleicht fragst du dich, wie es dir dabei helfen kann. Es handelt sich eben nicht um ein Python-Tutorial oder einen Einsteigerkurs. Wenn du Python erst lernst, wird dieses Buch dich nicht zu einem professionellen Python-Entwickler machen. Die Lektüre wird zwar auch dann von Vorteil für dich sein, aber du musst dir auch noch anderes Material anschauen, um grundlegende Python-Kenntnisse zu entwickeln.

Um den größten Nutzen aus diesem Buch zu ziehen, solltest du bereits über Python-Kenntnisse verfügen, die du erweitern möchtest. Am besten ist es, wenn du schon eine Weile in Python programmierst und bereit bist, in die Tiefe zu gehen, deine Kenntnisse abzurunden und deinen Code »pythonischer« zu machen.

Dieses Buch ist auch hervorragend für dich geeignet, wenn du schon Erfahrungen mit anderen Programmiersprachen hast und dich schnell in Python einarbeiten möchtest. Du wirst hier einen wahren Schatz an praktischen Tipps und Entwurfsmustern finden, die dir helfen, ein erfolgreicherer und qualifizierterer Python-Programmierer zu werden.

1.3 Wie du dieses Buch lesen solltest

Die beste Möglichkeit, dieses Buch zu lesen, besteht darin, sich ihm wie einem Büffet zu nähern. Jeder Python-Trick steht für sich selbst, weshalb du einfach zu denen springen kannst, die dich am meisten interessieren. Das ist die Vorgehensweise, die ich empfehle. Du kannst natürlich auch alle Python-Tricks in der Reihenfolge lesen, in der sie im Buch erscheinen. Dadurch verpasst du nichts. Wenn du auf der letzten Seite angekommen bist, kannst du sicher sein, dass du alle gesehen hast.

Einige der Tricks sind unmittelbar verständlich, weshalb du sie problemlos in deine tägliche Arbeit übernehmen kannst, nachdem du einfach das entsprechende Kapitel gelesen hast. Bei anderen Tricks ist etwas mehr Zeit für das Verständnis vonnöten. Solltest du Schwierigkeiten damit haben, einen bestimmten Trick in deinen eigenen Programmen anzuwenden, ist es oft hilfreich, die entsprechenden Codebeispiele im Python-Interpreter durchzuspielen. Wenn es dann immer noch nicht klick macht, wende dich getrost an mich, sodass ich dir helfen und die Erklärungen verbessern kann. Langfristig nützt das nicht nur dir, sondern allen Pythonistas, die das Buch lesen.

2 Muster für saubereres Python

2.1 Sich mit Zusicherungen absichern

Manchmal erhalten wirklich hilfreiche Merkmale einer Sprache nicht die Aufmerksamkeit, die sie verdienen. Aus irgendeinem Grund hat dies auch die Python-Anweisung `assert` getroffen.

In diesem Abschnitt gebe ich dir eine Einführung in die Verwendung von Zusicherungen (*Assertions*) in Python. Du erfährst hier, wie du damit Fehler in deinen Python-Programmen automatisch erkennen lassen kannst. Dadurch werden deine Programme zuverlässiger und lassen sich leichter debuggen.

Wahrscheinlich fragst du dich jetzt: »Was sind Zusicherungen, und wozu dienen sie?« Die Antworten darauf wollen wir uns in diesem Abschnitt ansehen. Im Grunde genommen handelt es sich bei der Python-Anweisung `assert` um eine Debugginghilfe, die eine Bedingung überprüft. Wenn diese Bedingung wahr ist, wird das Programm ganz normal weiter ausgeführt. Ist sie aber falsch, wird die Ausnahme `AssertionError` mit einer optionalen Fehlermeldung ausgeworfen.

Ein Beispiel

Das folgende einfache Beispiel zeigt, in welchen Fällen Zusicherungen praktisch sind. Zur Veranschaulichung habe ich ein möglichst praxisnahes Problem ausgewählt, wie es in deinen Programmen tatsächlich auftreten kann.

Nehmen wir an, du schreibst einen Onlineshop in Python. Dazu fügst du dem System die folgende Funktion `apply_discount` für die Einlösung eines Rabattcoupons hinzu:

```
def apply_discount(product, discount):
    price = int(product['price'] * (1.0 - discount))
    assert 0 <= price <= product['price']
    return price
```

Die `assert`-Anweisung garantiert, dass die von dieser Funktion berechneten rabattierten Preise niemals kleiner als 0 € und niemals höher als der reguläre Produktpreis sein können.

Um zu prüfen, ob das wirklich wie vorgesehen funktioniert, rufen wir diese Funktion auf, um einen gültigen Rabatt abzuziehen. In unserem Beispiel stellen wir die Produkte des Shops durch einfache Dictionarys dar. In der Praxis wird das wahrscheinlich nicht funktionieren, aber zur Veranschaulichung von Zusicherungen ist es gut geeignet. Als Beispielprodukt erstellen wir ein Paar schicker Schuhe für 149,00 €:

```
>>> shoes = {'name': 'Fancy Shoes', 'price': 14900}
```

Um Rundungsprobleme zu vermeiden, habe ich den Preis in einem Integer in Cent angegeben, was ganz allgemein eine sinnvolle Vorgehensweise ist. Aber ich schweife ab. Wenn wir auf diese Schuhe einen Rabatt von 25 % gewähren, sollten wir einen Verkaufspreis von 111,75 € erhalten:

```
>>> apply_discount(shoes, 0.25)
11175
```

Das funktioniert wie erwartet. Versuchen wir nun aber, einen ungültigen Rabatt anzuwenden, z. B. einen von 200 %, bei dem wir den Kunden für den Kauf der Schuhe noch bezahlen müssten:

```
>>> apply_discount(shoes, 2.0)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    apply_discount(prod, 2.0)
  File "<input>", line 4, in apply_discount
    assert 0 <= price <= product['price']
AssertionError
```

Jetzt hält das Programm mit einem Zusicherungsfehler (`AssertionError`) an, da ein Rabatt von 200 % die zugesicherte Bedingung in der Funktion `apply_discount` verletzt.

Die Ablaufverfolgung der Ausnahme zeigt an, welche Codezeile die verletzte Zusicherung enthält. Wenn du oder andere Entwickler in deinem Team beim Testen des Onlineshops auf einen solchen Fehler stoßen, kannst du anhand der Ablaufverfolgung leicht herausfinden, was geschehen ist. Das beschleunigt das Debugging erheblich und macht deine Programme langfristig wartungsfreundlicher. Das ist der große Nutzen von Zusicherungen.

Warum nicht einfach eine reguläre Ausnahme?

Vielleicht fragst du dich, warum ich in dem vorstehenden Beispiel nicht einfach eine `if`-Anweisung und eine Ausnahme verwendet habe. Zusicherungen dienen dazu, Entwickler über *nicht behebbare* Fehler in einem Programm zu informieren, aber *nicht* dazu, zu erwartende Fehlerbedingungen zu signalisieren – etwa eine

nicht gefundene Datei –, bei denen die Benutzer Abhilfemaßnahmen ergreifen oder einen erneuten Versuch unternehmen können.

Bei Zusicherungen handelt es sich um *interne Selbsttests* in einem Programm. Sie deklarieren bestimmte Bedingungen als *unmöglich*. Tritt eine solche Bedingung trotzdem ein, so bedeutet das, dass sich in dem Programm ein Bug befindet.

Ist das Programm fehlerfrei, kommt diese Bedingung niemals vor. Wenn sie aber doch auftritt, hält das Programm mit dem Zusicherungsfehler an, der dir mitteilt, welche unmögliche Bedingung den Absturz ausgelöst hat. Das macht es viel einfacher, Bugs zu finden und zu korrigieren. Und mir ist alles willkommen, was die Arbeit erleichtert. Dir nicht auch?

Merk dir, dass es sich bei der Python-Anweisung `assert` um eine Debugginghilfe handelt und nicht um einen Mechanismus für das Handhaben von Laufzeitfehlern. Zusicherungen sollen Entwicklern helfen, die Ursache eines Bugs schneller zu finden. Zusicherungsfehler sollten nur dann ausgelöst werden, wenn es in deinem Programm einen Bug gibt.

Im Folgenden wollen wir uns noch einige weitere Dinge ansehen, die wir mit Zusicherungen anstellen können, sowie zwei häufige Stolpersteine in der praktischen Anwendung.

Die Syntax von assert

Es ist immer sinnvoll, sich die Implementierung eines Merkmals in Python anzusehen, bevor du sie nutzt. Schauen wir uns also kurz die Syntax der Anweisung `assert` laut Python-Dokumentation[1] an:

```
assert_stmt ::= "assert" expression1 ["," expression2]
```

Hier ist `expression1` die Bedingung, die geprüft wird, und der optionale Ausdruck `expression2` die Fehlermeldung, die angezeigt werden soll, wenn die Zusicherung nicht erfüllt ist. Zur Ausführungszeit wandelt der Python-Interpreter jede `assert`-Zuweisung ungefähr in die folgende Abfolge von Anweisungen um:

```
if __debug__:
    if not expression1:
        raise AssertionError(expression2)
```

Hier fallen zwei Dinge auf: Vor der Prüfung der zugesicherten Bedingung erfolgt schon eine Überprüfung der globalen Variablen `__debug__`. Dabei handelt es sich um ein integriertes boolesches Flag, das unter normalen Umständen `True` ist, bei der Anforderung von Optimierungen aber `False`. Mehr darüber erfährst du im Abschnitt »Stolpersteine« auf S. 8. Zudem kannst du mit `expression2` eine optionale Fehlermeldung übergeben, die zusammen mit `AssertionError` in der Ablaufverfolgung angezeigt wird, um das Debugging noch weiter zu vereinfachen. Beispielsweise habe ich schon Code wie den folgenden gesehen:

1 Siehe Python-Dokumentation, »The Assert Statement«

```
>>> if cond == 'x':
...     do_x()
... elif cond == 'y':
...     do_y()
... else:
...     assert False, (
...         'This should never happen, but it does '
...         'occasionally. We are currently trying to '
...         'figure out why. Email dbader if you '
...         'encounter this in the wild. Thanks!')
```

Der Code ist zwar ziemlich hässlich, aber trotzdem handelt es sich hierbei um eine sinnvolle und hilfreiche Technik, um einen Heisenbug[2] in deinen Anwendungen aufzuspüren.

Stolpersteine bei der Verwendung von Zusicherungen

Bevor wir weitermachen, muss ich noch zwei Warnungen im Zusammenhang mit Zusicherungen in Python aussprechen. Die erste hat mit Sicherheitsrisiken und Bugs in deinen Anwendungen zu tun, die zweite mit einer syntaktischen Eigenheit, die es einfach macht, *nutzlose* Zusicherungen zu schreiben.

Warnung Nr. 1: Verwende Zusicherungen nicht zur Datenvalidierung

Zusicherungen können mit den Befehlszeilenschaltern `-O` und `-OO` und in Python auch mit der Umgebungsvariablen `PYTHONOPTIMIZE` global deaktiviert[3] werden. Dadurch wird jede `assert`-Zuweisung zu einer Nulloperation: Sie wird nicht kompiliert und auch nicht ausgewertet, und damit wird auch der Ausdruck der Bedingung nicht ausgeführt. Das ist eine Designentscheidung, die auch bei vielen anderen Programmiersprachen getroffen wurde. Als Nebenwirkung wird es dadurch sehr gefährlich, `assert`-Zuweisungen als schnelle und einfache Möglichkeit zur Validierung von Eingangsdaten einzusetzen.

Wenn du in einem Programm Zusicherungen einsetzt, um zu prüfen, ob ein Funktionsargument einen falschen oder unerwarteten Wert enthält, kann das nach hinten losgehen und zu Bugs und Sicherheitslücken führen. Sehen wir uns zur Veranschaulichung ein einfaches Beispiel an: Der Code deiner Onlineshop-Anwendung enthält eine Funktion, um auf Anforderung eines Benutzers ein Produkt zu löschen. Nachdem du Zusicherungen gerade kennengelernt hast, bist du nun vielleicht begierig, sie in deinem Code anzuwenden. Mir ginge es auf jeden Fall so! Deshalb implementierst du die Funktion vielleicht wie folgt:

```
def delete_product(prod_id, user):
    assert user.is_admin(), 'Must be admin'
    assert store.has_product(prod_id), 'Unknown product'
    store.get_product(prod_id).delete()
```

2 Siehe *https://en.wikipedia.org/wiki/Heisenbug*

3 Siehe Python-Dokumentation, »Constants (__debug__)«

Was aber geschieht, wenn Zusicherungen deaktiviert sind?

Durch die unsachgemäße Verwendung von Zusicherungen weist diese Funktion von nur drei Zeilen gleich zwei schwere Probleme auf:

1. **Die Überprüfung auf Administratorrechte mithilfe einer Zusicherung ist gefährlich.** Wenn Zusicherungen im Python-Interpreter deaktiviert sind, werden sie zu Nulloperationen, und dann kann *jeder Benutzer Produkte löschen*. Die Rechteüberprüfung wird nicht ausgeführt. Das ist ein Sicherheitsproblem, denn es öffnet Angreifern Tür und Tor, um Daten in deinem Onlineshop zu löschen oder zu beschädigen.
2. **Die Überprüfung mit `has_prodcut()` findet bei deaktivierten Zusicherungen nicht statt.** Dadurch kann `get_product()` auch mit einer ungültigen Produkt-ID aufgerufen werden. Das kann je nachdem, wie dein Programm geschrieben ist, weitere schwere Fehler hervorrufen. Schlimmstenfalls kann damit jemand einen Denial-of-Service-Angriff gegen deinen Shop starten. Wenn die Anwendung abstürzt, sobald jemand ein unbekanntes Produkt zu löschen versucht, könnte ein Angreifer sie durch ein Bombardement ungültiger Löschanforderungen zum Erliegen bringen.

Wie kannst du solche Probleme vermeiden? Dadurch, dass du Zusicherungen *niemals* zur Datenvalidierung verwendest. Stattdessen kannst du dazu reguläre `if`-Anweisungen und Ausnahmen einsetzen:

```
def delete_product(product_id, user):
    if not user.is_admin():
        raise AuthError('Must be admin to delete')
    if not store.has_product(product_id):
        raise ValueError('Unknown product id')
    store.get_product(product_id).delete()
```

Das bietet den zusätzlichen Vorteil, dass dabei keine unspezifischen `AssertionError`-Ausnahmen, sondern aussagekräftige `ValueError`- oder `AuthError`-Ausnahmen ausgelöst werden, die wir selbst definieren müssen.

Warnung Nr. 2: Zusicherungen, die immer erfüllt sind

Es ist erstaunlich einfach, in Python `assert`-Zuweisungen zu schreiben, die immer `True` sind. Das ist mir selbst auch schon passiert. Wenn du in einer `assert`-Anweisung ein Tupel als erstes Argument übergibst, wird die Zusicherung immer `True` sein und niemals fehlschlagen. Das ist beispielsweise bei der folgenden Zusicherung der Fall:

```
assert(1 == 2, 'This should fail')
```

Das liegt daran, dass nicht leere Tupel in Python immer als »truthy« angesehen werden. Wenn du ein Tupel an eine `assert`-Anweisung übergibst, wird die Bedin-