Migration auf Python 3

Warum die Uhr immer lauter tickt

Rainer Grimm, Modernes C++

Python 3 wurde im Jahr 2008 veröffentlicht. Da Python 3 nicht abwärtskompatibel zu Python 2 ist, wurde in der Regel der bestehende Code in Python 2 weiterentwickelt und der neue Code direkt in Python 3 geschrieben. Die Existenz zweier Parallelwelten nimmt am 1.1.2020 ein abruptes Ende. Mit dem 1.1.2020 wird der Unterstützung von Python 2 eingestellt. Die Migration der Codebasis von Python 2 nach Python 3 ist daher unvermeidlich.

Neue Feature in Python 3

print ist eine Funktion

Die neue Syntax von Print ist allgemein `print(*args, sep = " ", end = "\n", file = sys.stdout, flush = True)`, wobei `args` die Argumente, `sep` den Separator zwischen den Argumenten, `end` das Zeilenendzeichen, `file` das Ausgabemedium und `flush` den Puffer bezeichnet. Die folgende Tabelle stellt die syntaktischen Veränderungen der print-Funktion inklusive ihrer Defaultwerte gegenüber.

Beispiele	Python 2.*	Python 3.*
Allgemeine Form	`print "x= ",5`	`print("x= ",5)`
Zeilenumbruch	`print`	`print()`
Unterdrücken des Zeilenumbruchs	`print x,`	`print(x, end = "")`
Unterdrücken des Zeilenumbruchs ohne Leerzeichen		`print(1, 2, 3, 4, 5, sep = "")`
Umleitung der Ausgabe	`print >> sys.stderr, "error"`	`print("error", file = sys.stderr)`

Der Vorteil der neuen Version offenbart sich aber erst auf den zweiten Blick, denn die print-Funktion lässt sich jetzt überladen: Das Listing zeigt eine print-Funktion, die sowohl in die Standardausgabe als auch in eine Logdatei schreibt. Dazu instrumentalisiert sie die Built-in-Funktion `__builtins__.print`.

```
def print(*args, sep = " ", end = "\n", file = sys.stdout, flush = True):
  __builtins__.print(*args, sep = sep, end = end, file = file, flush = flush)
  __builtins__.print(*args, sep = sep, end = end, file = open("log.txt", "a"))
```

Lazy Evaluation

Nur das Nötigste tun, das ist bei Programmiersprachen durchaus eine Tugend. In Python 3 erhält Lazy Evaluation deutlich mehr Gewicht. Listen, Dictionaries oder die funktionalen Bausteine von Python erzeugen jetzt nicht mehr die gesamte Liste, sondern eben nur noch so viel, wie für die Auswertung des Ausdrucks notwendig ist. Diese Bedarfsauswertung spart kostbaren Speicherplatz und Zeit. Das erreicht der Python-Interpreter dadurch, dass er nur noch einen Generator zurückgibt, der auf Anfrage die Werte erzeugt. Dies war schon in Python 2 der feine Unterschied zwischen den Funktionen `range()` und `xrange()`. Mit Python 3 verhält sich nun `range()`wie `xrange()`, weshalb die zweite Funktion überflüssig wird.

Entsprechend liegt der Fall bei den funktionalen Bausteinen `map()`, `filter()` und `zip()`. Diese Funktionen wurden durch ihre Äquivalente aus der Bibliothek `itertools` ersetzt. Bei Dictionaries heißen die resultierenden Generatoren Views. Benötigt der Programmierer hingegen die voll expandierte Liste, hilft ein einfaches Kapseln des Generators in einem `list()`-Konstruktor, wie das folgende Beispiel zeigt: `list(range(11))` ergibt `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`.

True Division

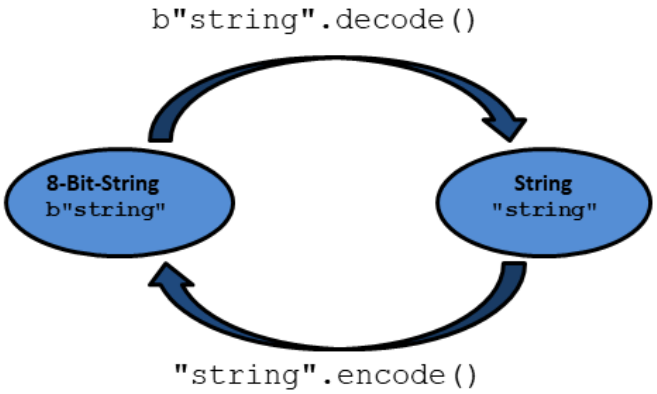
Gerade Python-Einsteiger sind häufig erstaunt, dass `1/2 == 0` ergibt. Python 3 beseitigt diesen Zustand und unterscheidet zwischen True Division und Floor Division. Während für die True Division `1/2 == 0.5` ergibt, verhält sich die Floor Division wie die Division in Python 2. Ihre Notation verwendet zwei Schrägstriche: `1//2 == 0`.

Unicode-Strings und Byte-Strings

Musste der Programmierer in Python 2 Strings noch explizit als Unicode-Strings deklarieren, so sind diese Zeichenketten jetzt implizit Unicode-Strings. Python 3 kennt nur noch Text und Daten. Text (`str` [1]) sind Strings und entsprechen dem Unicode-String aus Python 2. Daten (`bytes` [2]) sind 8-Bit-Strings und entsprechen den Python-2-Strings. Daten muss der Python-3-Entwickler deklarieren: `b"8-Bit-String"`. Die Tabelle zeigt dies in der Übersicht.

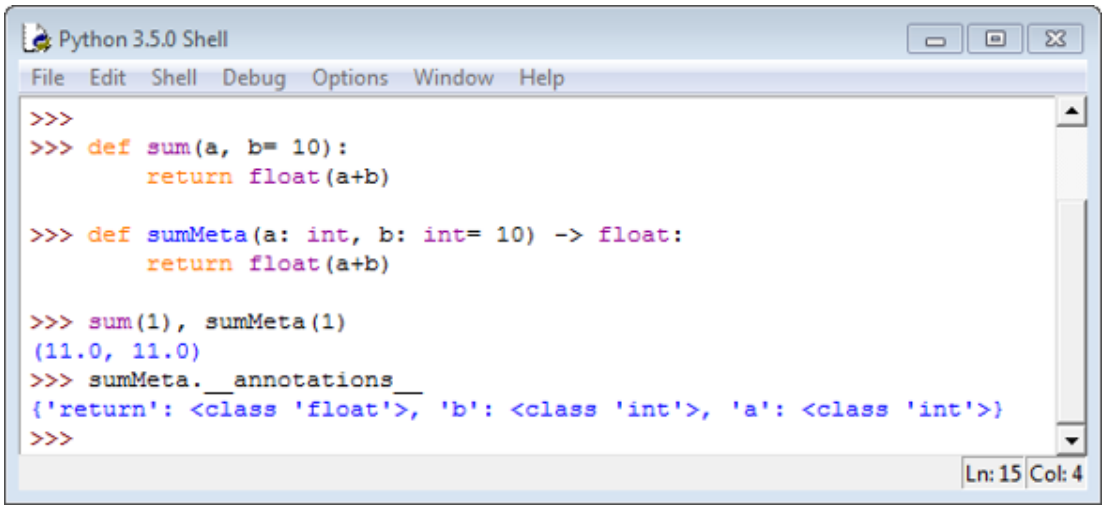
Typ	Python 2.*	Python 3.*
8-Bit-String	`"string"`	`b"string"`
Unicode-String	`u"string"`	`"string"`

Um zwischen den Datentypen zu konvertieren, gibt es die Funktionen `str.encode()` und `bytes.decode()`. Diese Konvertierung benötigt der Entwickler in Python 3, wenn er beide Datentypen verwendet, denn es findet keine implizite Typkonvertierung mehr statt.



Function Annotations

Mit Function Annotations bietet Python 3 die Möglichkeit, Metadaten an eine Funktion zu binden. Die Funktion lässt sich im zweiten Schritt auch mit Dekoratoren [3] versehen, die automatisch aus den Metadaten eine Dokumentation erzeugen oder die Typen zur Laufzeit prüfen. Die äquivalenten Funktionen `sumOrig()` und `sumMeta()` zeigen die Funktionsdeklaration mit und ohne Metadaten. Die zweite Funktion ist um Metadaten zur Signatur und zum Rückgabewert der Funktion erweitert. Die Metadaten lassen sich mit dem Funktionsattribut `__annotations__` referenzieren.



Aufräumarbeiten in Python 3

Wo es Veränderungen gibt, wollen auch Altlasten bereinigt und entsorgt sein. Dies betrifft Bibliotheken, die entfernt wurden, die nun gemäß dem Python Style Guide [4] klein geschrieben werden, die neu in Pakete verpackt wurden oder in einer C- und einer Python-Implementierung koexistieren.

Import Idiom

Das bekannte Python-Idiom, erst die schnelle C-Implementierung eines Moduls zu importieren und im Fehlerfall auf die Python-Implementierung zurückzugreifen, ist nicht mehr notwendig. Python erledigt dies automatisch.

```
try:
    import cPickle as pickle
except ImportError:
    import pickle
```

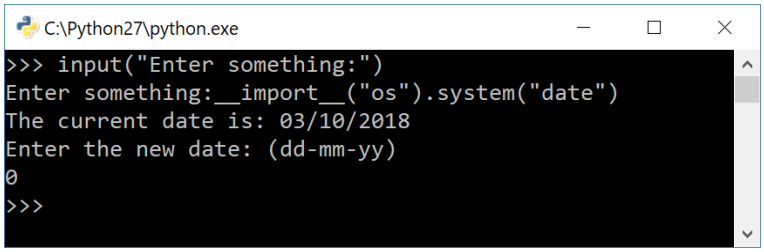
Genaueres zu den Änderungen der Standardbibliothek ist unter [5] zu finden.

Kooperative `super`-Aufrufe und Old-Style Klassen

Es gibt weitere Punkte, die das Leben des Python-Programmierers erleichtern. So muss er bei kooperativen `super`-Aufrufen nicht mehr die Instanz der Klasse und den Klassennamen nennen. Die Old-Style-Klassen, existieren mit Python 3 nicht mehr, sodass das lästige Ableiten von `object` nicht mehr notwendig ist, um die neueren Features von Python anzusprechen.

`input` entfernt

Das unmittelbare Evaluieren der Eingabe mit Hilfe des Kommandos `input()` ist nicht mehr möglich, da die Eingabe als Eingabestring zur Verfügung steht. Damit hat sich ein extremes Sicherheitsloch geschlossen.



Konsequenterweise wurde die Funktion `raw_input()` in `input()` umbenannt, sowie `raw_input()` entfernt.

Rückportierung von Python Featuren

Sinn und Zweck von Python 2.7 ist es, den Umstieg auf die Version 3 so einfach wie möglich zu vollziehen. Aus diesem Grund hat das Projekt viele Features von Python 3.0 auf Python 2.7 rückportiert.

Kontext-Manager

Der Kontext-Manager mit `with` ist ein wichtiges neues Feature, das mit Python 2.6 zur Verfügung steht. Eine Ressource (Datei, Socket, Mutex etc.) bindet Python automatisch beim Eintritt in den `with`-Block und gibt sie beim Austritt wieder frei. C++-Programmierern wird dieses Idiom an „Resource Acquisition Is Initialization“ (RAII) erinnern [6].

Das `with`-Statement verhält sich aus Anwendersicht wie ein `try` … `finally`, da sowohl der `try`-Block als auch der `finally`-Block immer ausgeführt werden. Dies alles geschieht aber ohne explizite Ausnahmebehandlung.

Wie funktioniert nun das Ganze? In einem `with`-Block lässt sich jedes Objekt verwenden, das das Kontext-Management-Protokoll anbietet, das also die internen

Methoden `__enter()__` und `__exit()__` besitzt. Beim Eintritt in den `with`-Block ruft Python die `__enter()__` und beim Austritt die `__exit()__` Methode automatisch auf. Das Datei-Objekt bringt die entsprechenden Methoden von Haus aus mit.

```
with open('/etc/passwd', 'r') as myFile:
    for line in myFile:
        print line
# myFile automatically closed
```

Ressourcen-Management ist aber auch schnell selbst durch die Methoden `__enter()__` und `__exit()__` implementiert. Wem dies noch zu viel Arbeit ist, der kann den Dekorator `contextmanager` aus der Bibliothek `contextlib` [6] verwenden, um vom Kontext-Management zu profitieren. Weitere Anwendungsfälle sind im Python Enhancement Proposal (PEP) 0343 [8] zu finden.

Abstrakte Basisklassen

Die wohl größte syntaktische Erweiterung vollzieht sich in Python 2.6 mit der Einführung von abstrakten Basisklassen. Ob ein Objekt sich in einem Kontext verwenden lässt, hing bisher von den Merkmalen des Objekts ab und nicht von dessen formaler Schnittstellenspezifikation.

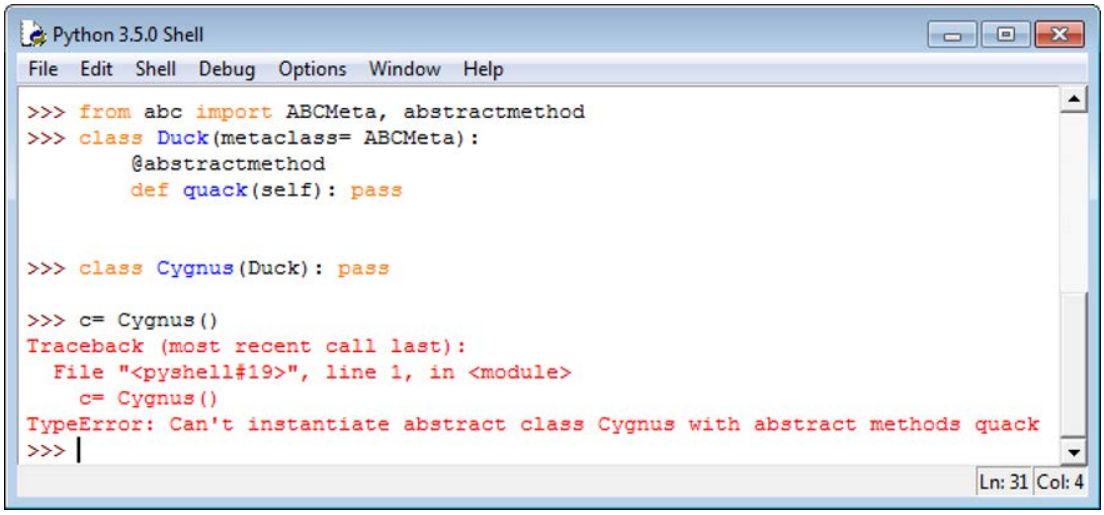
Dieses Idiom wird Duck-Typing genannt, frei nach dem Gedicht von James Whitcomb Riley: „When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck".

Sobald eine Klasse eine abstrakte Methode besitzt, wird sie zur abstrakten Basisklasse und lässt sich nicht instanziieren. Von ihr abgeleitete Klassen können nur erzeugt werden, wenn sie diese abstrakten Methoden implementieren. Abstrakte Basisklassen in Python verhalten sich ähnlich wie abstrakte Basisklassen in C++, insbesondere dürfen abstrakte Methoden eine Implementierung enthalten.

Neben den abstrakten Methoden kennt Python auch abstrakte Properties. Die Python-Version 3 verwendet abstrakte Basisklassen in den Modulen `numbers` [9] und `collections` [10].

Die entscheidende Frage steht noch aus: Wie wird eine Klasse zur abstrakten Klasse? Die Klasse benötigt die Metaklasse `ABCMeta`. Daraufhin lassen sich die entsprechenden Methoden als `@abstractmethod` oder Properties als `@abstactproperty` mit Hilfe des entsprechenden Dekorators deklarieren. Die Verwendung von abstrakten Basisklassen bedeutet darüber hinaus, dass in die dynamisch typisierende Sprache statische Typisierung Einzug hält.

Das Beispiel kann die Klasse `Cygnus` nicht instanziieren, da es die abstrakte Methode `quack()` nicht implementiert.

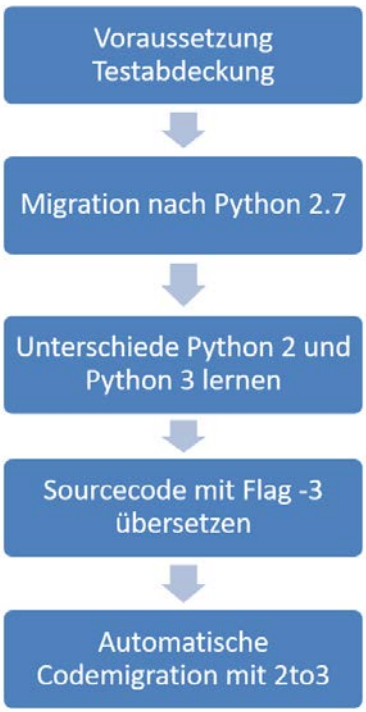


Mehrere Prozessoren

Pythons Antwort auf Multiprozessor-Architekturen ist die neue Bibliothek `multiprocessing` [11]. Dieses Modul imitiert das bekannte Python-Modul `threading`, nur erzeugt es statt eines Thread einen Prozess, und dies auch noch plattformunabhängig. Das Multiprocessing-Modul war notwendig, da in CPython, der Standardimplementierung von Python, nur ein Thread im Interpreter laufen kann. Geschuldet ist dieses Verhalten dem so genannten Global Interpreter Lock, kurz GIL [12].

Migration auf Python 3

Zum Portieren von Python-2-Code nach Python 3 zeichnet sich ein klar definierter Pfad ab, wobei der Entwickler nach jedem Schritt den Code testen und Probleme beseitigen muss.

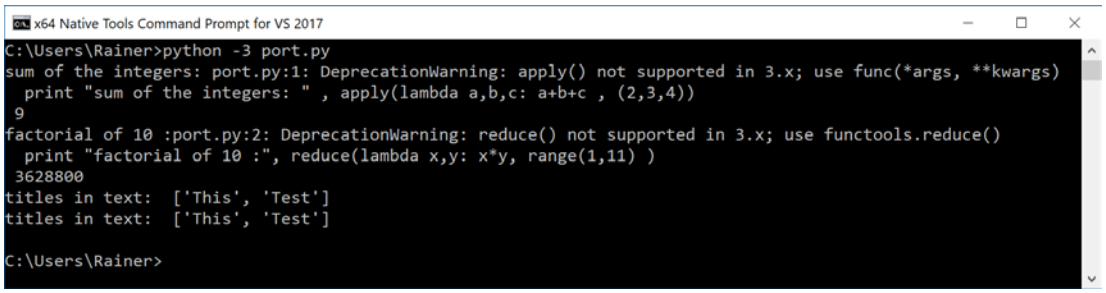


Die vier Codezeilen sollen als Beispiel für die Migration von Python 2 nach 3 dienen. Alle vier Zeilen des Beispiels verwenden funktionale Komponenten von Python, da sich bei diesen built-in-Funktionen einige Veränderungen vollzogen haben.

```
print "sum of the integers: " , apply(lambda a,b,c: a+b+c , (2,3,4))
print "factorial of 10 :", reduce(lambda x,y: x*y, range(1,11) )
print "titles in text: ", filter( lambda word: word.istitle(),"This is a long Test".split())
print "titles in text: ", [ word for word in "This is a long Test".split() if word.istitle()]
```

Die erste Funktion berechnet die Summe der drei Zahlen 2, 3 und 4, indem sie diese Argumente auf die Lambda-Funktion anwendet. Das built-in `reduce()` reduziert sukzessive die Liste aller Zahlen von 1 bis einschließlich der 10, indem sie das Ergebnis der letzten Multiplikation mit der nächsten Zahl aus der Sequenz multipliziert. Die letzten zwei Funktionen filtern aus dem String alle Wörter heraus, die mit einem Großbuchstaben beginnen. Der Code funktioniert bereits unter Python 2.6, sodass nur noch die Schritte 3 und 4 für die Portierung zu vollziehen sind.

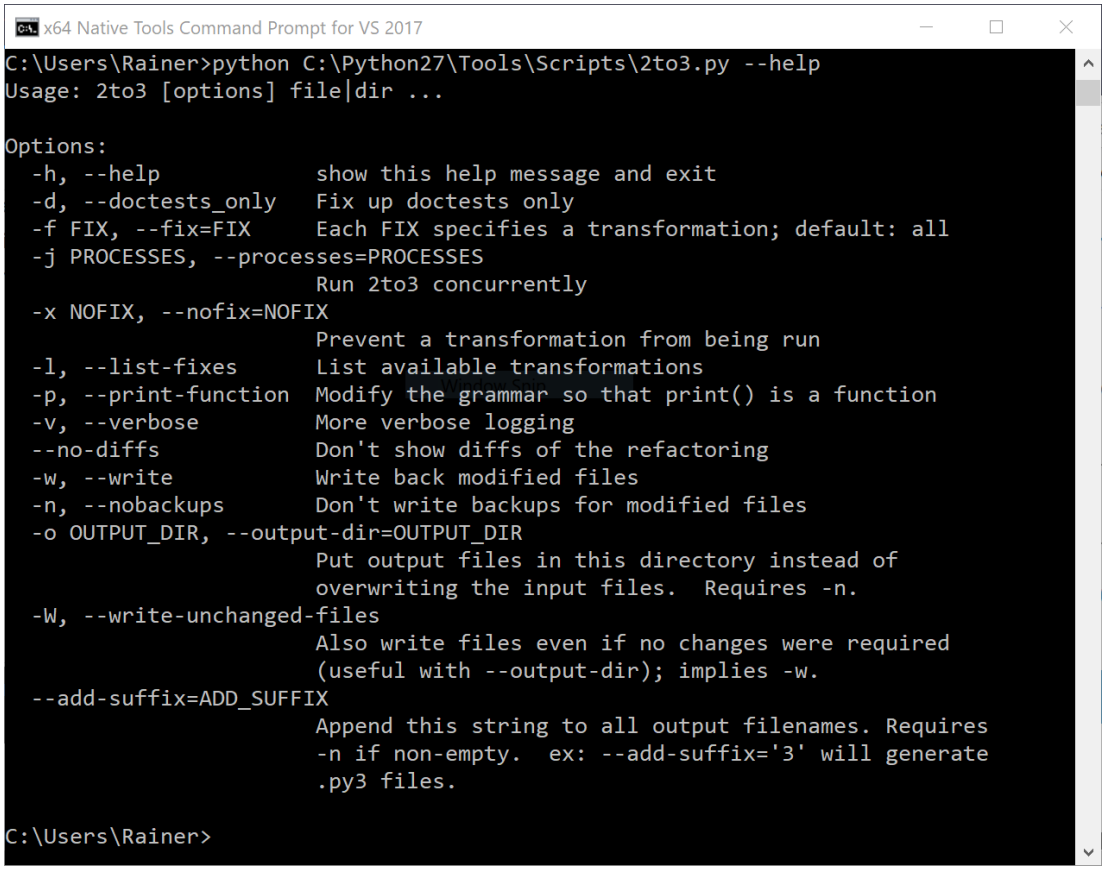
Ein Aufruf des Python-2.7-Interpreters mit der Option `-3` zeigt die Inkompatibilitäten zur Version 3: Sowohl die Funktion `apply()` als auch die Funktion `reduce()` sind in Python 3 keine built-ins mehr.



Der Code ist schnell repariert und die Deprecation-Warnungen unterbleiben.

```
print "sum of the integers: " , (lambda a,b,c: a+b+c)(*(2,3,4))
import functools
print "factorial of 10 :", functools.reduce(lambda x,y: x*y, range(1,11) )
print "titles in text: ", filter( lambda word: word.istitle(),"This is a long Test".split())
print "titles in text: ", [ word for word in "This is a long Test".split() if word.istitle()]
```

Das Script `2to3.py` erweist sich bei der Korrektur des Python-2-Code als sehr hilfreich, denn er erzeugt im letzten Schritt automatisch Code für Python 3. Dazu bietet das Tool mehrere Optionen an.



Der direkte Weg besteht darin, die Ursprungsdatei zu überschreiben: `python <path to 2to3.py> port.py -w`. Das Ergebnis ist der nach Python 3 portierte Quelltext. Interessanterweise hat der Codegenerator den `filter()`-Ausdruck durch eine äquivalente List-Comprehension ersetzt.

```
print("sum of the integers: " , (lambda a,b,c: a+b+c)(*(2,3,4)))
import functools
print("factorial of 10 :", functools.reduce(lambda x,y: x*y, list(range(1,11)) ))
print("titles in text: ", [word for word in "This is a long Test".split() if word.istitle()])
print("titles in text: ", [ word for word in "This is a long Test".split() if word.istitle()])
```

Python 2 und Python 3 unterstützen

Soll der Python-Code so umgeschrieben werden, so dass er sowohl durch Python 2 als auch durch Python 3 unterstützt wird, dann helfen die Scripte `futurize` [13] und `modernize` [14]. Das Script `modernize` ist bei der Modifikation des Python Codes konservativer als das Script `futurize`. Wie im vorherigen Abschnitt gibt es zwei Voraussetzung für die Modifikation des Python-Codes: Eine Testabdeckung zu besitzen und den Python-Code bereits Python 2.7 migriert zu haben.
Welche syntaktischen Unterschiede Python 2 und Python 3 im Detail mitbringen und wie sich Python-Code schreiben lässt, der sowohl von Python 2 als auch von Python 3 unterstützt wird, das beschreibt der Spickzettel „Cheat Sheet: Writing Python 2-3 compatible code“ [15] im Detail.

Weiterführende Information

[1]: Datentyp `str`: http://docs.python.org/3/library/functions.html#str
[2]: Datentyp `bytes`: http://docs.python.org/3/library/functions.html#bytes
[3]: Rainer Grimm, „Dekoratoren in Python“: Linux-Magazin 06/09, S. 96
[4]: PEP 8 – Style Guide for Python Code: https://www.python.org/dev/peps/pep-0008/
[5]: What’s New in Python 3: https://docs.python.org/3/whatsnew/3.0.html#library-changes
[6]: RAII: http://de.wikipedia.org/wiki/Ressourcenbelegung_ist_Initialisierung
[7]: Bibliothek contextlib: http://docs.python.org/3/library/contextlib.html#module-contextlib
[8]: PEP 0343: http://www.python.org/dev/peps/pep-0343
[9]: Bibliothek numbers: http://docs.python.org/3/library/numbers.html#module-numbers
[10]: Bibliothek collections: http://docs.python.org/3/library/collections.html#module-collection
[11]: Bibliothek multiprocessing: http://docs.python.org/3/library/multiprocessing.html#module-multiprocessing
[12]: Der Global Interpreter Lock (GIL): http://docs.python.org/c-api/init.html#thread-state-and-the-global-interpreter-lock
[13]: Futurize: http://python-future.org/automatic_conversion.html
[14]: Modernize: https://python-modernize.readthedocs.io/en/latest
[15]: Cheat Sheet: Writing Python 2-3 compatible code: http://python-future.org/compatible_idioms.html

Autor

Rainer Grimm ist seit vielen Jahren als Softwarearchitekt, Team- und Schulungsleiter tätig. In seiner Freizeit schreibt er gerne Artikel zu den Programmiersprachen C++, Python und Haskell, spricht aber auch gerne auf Fachkonferenzen. Auf seinem Blog Modernes C++ (Heise Developer) beschäftigt er sich intensiv mit seiner Leidenschaft C++. Seit 2016 steht er auf selbstständigen Beinen. Insbesondere das Vermitteln von Wissen zu modernem C++ ist ihm eine Herzensangelegenheit.
Seine Bücher "C++11 für Programmierer", "C++" und "C++-Standardbibliothek" für die kurz und gut Reihe sind beim Verlag O'Reilly erschienen. Seine englischsprachigen Werke "The C++ Standard Library" und "Concurrency with Modern C++" sind in mehrere Sprachen übersetzt worden.