Python 3 - die Zukunft?

Bekannte und weniger bekannte Probleme
und ein subjektiver Blick

Reinhard Wobst
UNIX-Stammtisch Dresden, 5.2.14

Inhaltsverzeichnis

1. Hintergrund

- Python war, zumindest seit der ersten Version 2, immer fast perfekt aufwärtskompatibel geblieben. Störend waren chronischer Ärger mit der String-Codierung, und die Ästheten störten sich an "schrägen" Syntaxkonstruktionen wie

```
print "ab =", ab
exec ("ab%d" % nn) + "=" + ("a%d" % nn)
   # führt z.B. "ab95 = a95" aus
```

- Als Fernziel gab es ein ominöses **Python3000** (oder Python3K), das nie erreicht (erst im Jahr 3000) und inkompatibel werden würde, aber als Ziel dienen sollte (so wie der Kommunismus ☺).

- Guido van Rossum publizierte **2006** den PEP 3000 und eröffnete eine Mailingliste.
- Andere Mitglieder der PSF wurden allmählich überzeugt, das Jahr 3000 vorzuziehen.
- Der Rest der Schafherde soll nachfolgen (meine grobe Darstellung).
- **Oktober 2008:** Python2.6 erscheint und soll das letzte Release vor Python 3 sein und gibt nervige Warnungen bei allen zukünftigen Inkompatibilitäten aus (Abschalten war schwierig).
- **Dezember 2008:** Python 3.0 erscheint, bei I/O viel zu langsam. Wie KDE 4.0: Software-Frühchen.
- **Juli 2010:** Python 2.7 erscheint mit vielen Features von Python 3, **ohne** die nervigen Warnungen by default. Wird mindestens bis Juli 2015 noch weiterentwickelt.
- **August 2010:** Python 3.2 erscheint und enthält erste Versionen von Codecs, die mit Python 3.0 entfernt wurden.
- **November 2011:** PEP 404 = "nein, es wird nie ein Python 2.8 geben".

- **September 2012:** Python3.3 erscheint und erlaubt u.a. wieder die Syntax `u'unicode_string'` (leichteres Portieren) und endlich ein speichereffektives Stringmodell.
- **Dezember 2013:** RedHat wird Python2.7 bis mindestens 2023 unterstützen
- Generell: "Wir ziehen alles auf Python3!"

Interessante Links

http://python-notes.curiousefficiency.org/en/latest/python3/questions_and_answers.html (Fundgrube - vor allem zum Ende hin)

http://lucumr.pocoo.org/2011/12/7/thoughts-on-python3/ (Armin Ronacher, alles sehr lesenswert von ihm!)

2. Codierung

Das war der Hauptgrund, Python 3 einzuführen - in Python2 ist folgendes erlaubt:

```
>>> u'abc' + 'de'
u'abcde'
>>> u'abc' + 'de\xe9'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode
byte 0xe9 in position 2: ordinal not in
range(128)
```

(intern sind u-Strings z.B. 16-Bit-Zeichen, Strings aber 8-Bit; implizite Konvertierung). (In Python 3 geht es pro forma!)

Lösung in Python3:

- *Jeder* String ist Unicode
- Python2-Strings werden Bytearrays: `b'abc'`
- code/decode "only at boundaries" (I/O)
- decode: nur string -> bytearray
- encode: nur bytearray -> string

Hat aber Nebenwirkungen, und auch nicht gemeldete Fehler sowie unerwartete Effekte gibt es noch genügend.

Nebenwirkungen:

- `b'Haette gern %d Murmeln' % 5` geht nicht mehr
- `b'Hätte gern Murmeln'` auch nicht
- Lösung:

- Python 2.7 -- `'a%dc' % 2`
- Python 3.3 –

```
(b'a%dc'.decode(encoding='latin1') % \
     2).encode('latin1')
```

  schööööön.

- Filenamen mit mixed encoding: Bytearrays, alles umschreiben
- Fileinhalte mit mixed encoding (Database, many users worldwide): dito
- ConfigParser: dito, da Encoding der Einträge nicht garantiert
- Files entweder im Mode `'rb'` als Bytearray einlesen, dann geht aber `splitlines()` nicht mehr, oder: `encoding = latin1`
- stdout/stdin by default mit encoding; Umgehen (Christian Heimes):

```
# switch to binary mode
>>> sys.stdout = sys.stdout.buffer

switch back
>>> sys.stdout = sys.__stdout__
```

schööööön.

- `os.walk(topdir)` liefert Strings, wenn `topdir` String ist, sonst Bytearrays (nicht dokumentiert, aber entscheidend)
- Py3.1/3.2 brauchten 2-4fachen Speicher für Strings (da Unicode); Py3.3 belegt clever nur die Breite des "größten" Zeichens. Aber wehe, wenn 16-Bit-Zeichen zu 8-Bit-String addiert wird (zeitweilig dreifacher Speicher, Performance) - man arbeitet oft mit "Riesenstrings" in Python (bei mir z.B.: Garmin-Karte patchen, knapp 2GB). Python 2 erfordert theoretisch bei entsprechender Optimierung fast keine Extra-Ressourcen (alter String wird verworfen = "überschrieben").
- Probleme sind vorprogrammiert:

```
python3.3
...
>>> mb = set(dir(b'abc'))
>>> ms = set(dir('abc'))
>>> mb-ms
{'decode', 'fromhex'}
>>> ms-mb
{'__rmod__', 'format', 'format_map', 'encode',
'casefold', 'isprintable', '__mod__',
'isdecimal', 'isnumeric', 'isidentifier'}
>>>
```

Was also, wenn man in Python einen Parser schreiben will, der beliebige Encodings akzeptieren muss (Bytearray)? Oder sich aus Binärfiles Zahlen herauspicken will? Oder eben schlicht mal formatieren will?

Besserung für Python 3.4 angekündigt. Oder 3.5. Vgl. a. http://bugs.python.org/issue3982#msg199265, PEP 383.

- Wirklich eine Lösung?? Bin noch immer beim Lernen und Zweifeln. Vielleicht hätte es auch ein optionales Stringattribut "encoding" getan?

3. Weitere Inkompatibilitäten

- 3//2 = 1, 3/2 = 1.5
  Wird von den Entwicklern als "endlich richtig" angesehen, von mir als C/Fortran/Numerikprogrammierer überhaupt nicht (Feldindizes ausrechnen). Sichere Umstellung erfordert genaues Wissen über den Kontext!
  Bei duck-typing-Sprachen sollte man die Finger von solchen Änderungen lassen. Saubere Umstellung bei großen Projekten kaum möglich.
- Python2: Iterator ist im Prinzip jedes Objekt, das eine `next()`-Methode hat (genial).
  Python3: `__next__()`, weil intern.
  Und `next(iterator)` statt `iterator.next()`.

  Riesenproblem für "allgemeine Servicemodule", deren Eingabe ungewiss ist (von Plugins oder Usercode etwa).

Meine Meinung: Problem ignoriert.
Und warum dann nicht gleich `__run__()` im threading-Modul?

- `dict.keys()` ist keine Liste mehr, sondern ein "view" (ähnlich Iterator, ändert sich zudem synchron). Sicherlich vernünftig, aber warum nicht wie in Python2.7 `dict.viewkeys()`?
  Und wenn nun einer schreibt

```
L = dict.keys()
...
# much code, another function in another file
L += ['a', 'AA']
```

- `list.sort()` nun nur noch mit `key=...`, compare-Funktion muss umständlich emuliert werden: "Braucht keiner". So? Warum nicht so gelassen?
- **Oktalzahlen:** 0777 in Python2, 0o777 in Python3. Geschmackssache, aber Python lehnt sich an C an, und das wird offenbar als "legacy" betrachtet.

- `os.popen()` und Verwandte sollen durch das Modul `subprocess` ersetzt werden - leistungsfähiger, aber deutlich komplizierter. Warum nimmt man uns das Einfache weg?
- `print` wird zur Funktion (schon in Python 2.7 nutzbar) - vernünftig, einfach, nur muss alter Code umgestellt werden (es gibt bereits Ansätze, `print()` implizit zu rufen bei Verwendung als Keyword). `2to3` (s.u.) wird nicht immer korrekt arbeiten. Warum nicht neues `printf`?
  Schlimmer: Immer noch droht man mit der Abschaffung der %-Formatierung - der Ersatz sieht so aus:

```
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
```

  Nichts, was man von der neuen, kryptischen Syntax nicht viel einfacher und expliziter in Python2 ausdrücken könnte.
  Bis jetzt ist `%` noch erlaubt ☺.
- Viele Module, Methoden u.a. umbenannt oder ganz gestrichen:

- `thread` -> `_thread`
- `ConfigParser` -> `configparser`
- `callable(f)` -> `hasattr(f, __call__)`
- `d.has_key(k)` -> `if k in d`
- `xrange` -> `range`
- `apply(f, args)` -> `f(*args)`
- `threading.Lock.release_lock()` -> `threading.Lock.release()` ...
- `raise "error"` -> `raise Exception("error")`

Ein Aberglaube, zu denken, das könne alles `2to3` erledigen! (s.u.)

- import-Logik verbessert (kann notwendig sein, aber noch eine Änderung, die bei der Portierung das Verstehen des alten Codes erfordert)
- `pickle`-Interna geändert, C-API ebenfalls ...
- vgl. *python2python3.pdf*

4. Portierung

Ursprüngliche Vorstellung der Entwickler:

- Code in Python2.6/2.7 schreiben, alle Warnungen aktivieren
- Unmassen Unittests schreiben (damit fehlerfrei ☺)
- `2to3` anwenden (automatische Konvertierung)
- Tests laufen lassen, Änderungen nur im Python2-Code.

Nicht praktikabel, weil:

- zu langsam bei großen Projekten
- unvollständige Konvertierung
- Änderungen nicht alle in Python2 durchführbar
- Probleme bei Versionsverwaltung
- Python2-Zweig wird nicht parallel unterstützt

Heute besser:

- Projekt `six` (Code soll unter Python2 wie Python3 laufen), aber bläht den Code auf, Performance
- andere Projekte: `pies` (http://timothycrosley.github.io/pies/) - setzt auf Python 2.6/2.7 auf, Code soll wie Python 3 aussehen; `python-modernize`, `python-future` (http://python-future.org/index.html) als möglicherweise zukunftsträchtigste Lösung, aber dort muss man nicht nur `b'abc'` schreiben, sondern `bytes(b'abc')`, und z.B. auf Iteratoren muss man selbst achten (müssen mit Dekorator markiert werden). Dokumentation ist gut; man ahnt, wie viele Detailprobleme hier lauern.

`python-future` contains, in addition to the `future` compatibility package, a `futurize` script that is similar to `python-modernize.py` in intent and design. Both are based heavily on `2to3`.

Whereas `python-modernize` converts Py2 code into a common subset of Python 2 and 3, with `six` as a run-time dependency, `futurize` converts either Py2 or Py3 code into (almost) standard Python 3 code, with `future` as a run-time dependency.

Because `future` provides more backported Py3 behaviours from `six`, the code resulting from `futurize` is more likely to work identically on both Py3 and Py2 with less additional manual porting effort.

5. Kritik

- Python 3 ist keineswegs einfacher, wie behauptet (Ersatz von Shellskripten, binäres Patchen - überall, wo Encoding sekundär ist, macht es Ärger).
- Existierender individueller Code wird nicht portiert werden, nur bekannte Libs. `mySQLdb` geht noch nicht (Details unklar), `Trac` dürfte wegen der Hunderte Plugins (viele ohne Maintainer, aber im kritischen Einsatz) Probleme bekommen ...
- Viele wichtige Projekte laufen bereits unter Python 3, aber die Masse an existierendem, kritischen Code wird nicht gesehen, nur die öffentlichen Projekte in Bewegung.
- Notwendigkeit von Python3 wird von vielen nicht so gesehen, nur: "Python 2 soll nicht mehr unterstützt werden".
- Reale Welt ist nicht so schön, wie es sich die Entwickler vorstellen:
  - reden meist von Linux

- sehen nur die OpenSource-Repos
- überschätzen die Entwicklerkapazitäten in Betrieben
- C ist nach wie vor fast die Hauptsprache, kann nicht als "legacy" abgetan werden (Syntax, C-API)
- Unicode-Identifier sind eine Schnapsidee in der Praxis (sieht man nicht ein!)
- Webprogrammierung wird wohl von den Py-Entwicklern als Schwerpunkt gesehen - es gibt aber noch Anderes (Datenkonvertierung und -auswertung, in meinen Lehrgängen Schwerpunkt)

- *deprecated* heißt noch lange nicht: *kann weg*. Wird aber so gesehen.

6. Zukunft

Ich sehe drei Möglichkeiten:

1. Entwickler forcieren eine Lösung, Py2 und Py3 kooperieren zu lassen, ohne alten Code wesentlich anfassen zu müssen. Vielleicht beginnend mit Python 3.4 und python-future.
2. Python 3 wird in der Breite ignoriert.
3. Der Großteil des Python-Codes wird verworfen, man wechselt zu anderer Sprache.

Es ist noch alles offen ... und ich lerne noch ...