



Python 3000

Georg Brandl

10. April 2008

Python 3000 – Was?

- Nächste Major-Version, Python 3
- Keine Pflicht zur Rückwärtskompatibilität
- Aktueller Stand: Alpha 4
- Final-Release zusammen mit 2.6 geplant: September 2008



Warum?

Nächster Schritt in der Entwicklung von Python:

- **Beseitigung von Altlasten**
(alte Klassen, print-Statement, Integer-Division)
- **Vereinfachung und Modernisierung**
(Unicode-Strings, Longs)
- **Neue sinnvolle Features, zu Altem inkompatibel**
(Dict-Views, mehr Iteratoren)



Änderungen im Detail



String vs Unicode

- Es gibt kein `unicode` mehr, `str` ist jetzt ein Unicode-String
- Dafür gibt es `bytes` und `bytearray` (mutable) für Binärdaten
- Keine implizite Konversion zwischen beiden Typen mehr: Saubere Trennung
- Bytes-Literale: `b"ASCII"`
- In IO integriert: Strings für kodierte Textdateien, Bytes für Lowlevel-Dateien, Sockets, etc.



print wird eine Funktion

```
print('hello', 'world')
```

Anstatt ">>"-Syntax

```
print(a, b, file=sys.stderr)
```

Anstatt Komma am Schluss

```
print('hello ', end='')
```

```
print('world')
```

Neu: Trennstring

```
print(*columns, sep=', ')
```



Stringformatierung

Neue Methode `format()`, anstatt `%` zu „missbrauchen“:

```
>>> "{0} {1}".format("hello", "world")
'hello world'
>>> "{hello} {world}".format(hello="foo", world="bar")
'foo bar'
>>> a.attr = 5
>>> "{0.attr}".format(a)
'5'
>>> "{0:05}".format(42)
'00042'
>>> "{0!r}".format("foo")
'"foo"'
```



Unicode-Quelltext

- Standardencoding ist UTF-8
- Identifier nicht mehr auf ASCII beschränkt:

```
class Käse:  
    pass
```

```
Gruyère = Käse()
```

- Encoding-Cookies funktionieren weiterhin



Dictionary-Views

- `dict.keys()`, `dict.values()`, `dict.items()` geben *Views* zurück
- Key- und Items-Views sind mengenähnliche Objekte (ähnlich, da Values nicht hashable sein müssen)
- Values-Views unterstützen eigentlich nur Iteration, `len()` und `in`
- `dict.iteritems()` etc. gibt es nicht mehr



Set und Dict Comprehensions

- Mengenciliterale: {1, 2, 3}
- Kein Literal für die leere Menge :{}
- Comprehensions:

```
>>> {x for x in range(5) if x % 2 == 0}
{0, 2, 4}
```

```
>>> {x : x**2 for x in range(5) if x % 2 == 0}
{0: 0, 2: 4, 4: 16}
```



Iteratoren als Standard

- `map()`: jetzt wie `imap()`
- `filter()`: jetzt wie `ifilter()`
- `zip()`: jetzt wie `izip()`
- `range()`: jetzt wie `xrange()`
- Explizites `list()`, `set()` etc., um Collection zu bekommen



Extended Unpacking

```
>>> a, *b = range(5)
```

```
>>> a
```

```
0
```

```
>>> b
```

```
[1, 2, 3, 4]
```

```
>>> *a, b = range(5)
```

```
>>> a
```

```
[0, 1, 2, 3]
```

```
>>> b
```

```
4
```



Numerisches

- Kein Unterschied mehr zwischen `int` und `long`
- Oktal: `0o755`
- Binär: `0b1011000`
- Dazu passend: `bin()`
- $1/2$ ist jetzt endgültig `0.5`
- $1//2$ gibt aber `0`



Import

- Relativer Import in Packages nur noch mit führendem `..`:

```
# Importiert nur Toplevel-Modul  
import foo
```

```
# Importiert Modul im Package  
from . import foo
```

```
# Importiert Modul in Elternpackage  
from ..bar import foo
```



Exceptions

- Keine String-Exceptions mehr: ~~raise "error"~~
- Alte Syntax entsorgt: `raise TypeError, "foo"`
- Statt dessen: `raise TypeError("foo")`
- Raise mit explizitem Traceback: `raise TypeError("foo") from tb`
- Except-Syntax: `except TypeError as err:`



Neues super

- Jetzt ohne explizite Übergabe von Klasse und Instanz:

```
class Foo(Bar):  
    def do_something(self, a, x):  
        super().do_something(a, x)  
    ...
```

- Alter Aufruf mit Parametern funktioniert weiterhin



Neues I/O-System

- High Level: `open()` wie zuvor (aber Encodings built in)
- Low Level: `io`-Modul: `FileIO`, `BufferedWriter`, etc.
- Bytes für Binärdaten, Strings für Text
- `io.BytesIO` und `io.StringIO`



Annotations

- Beliebige Attribute for Funktionsparameter und Rückgabewert:

```
def seek(self, pos: int, whence: int = 0) -> int:  
    pass
```

- Standardmäßig keine weiteren Auswirkungen, aber Attribute über `__annotations__` im Funktionsobjekt auslesbar
- Libraries denkbar, die Typen überprüfen o.ä.



Tod den „old-style“-Klassen

- Klassen, die nicht von `object` abgeleitet waren
- Diverse kleinere Unterschiede: `type(x)`, keine Deskriptoren
- Neu: alles erbt von `object`

Außerdem:

- Keine „unbound methods“ mehr



Abstract Base Classes

- Python's Version von „Interfaces“

```
class Mapping(metaclass=ABCMeta):  
    @abstractmethod  
    def keys(self):  
        raise NotImplementedError  
  
    ...
```

- Konkrete Klassen müssen keys() überschreiben
- Viele vordefinierte ABCs: Integer, Sequence, Mapping etc.



isinstance/issubclass überschreibbar

- Eine Klasse C kann `__instancecheck__()` definieren, um das Ergebnis von `isinstance(x, C)` zu bestimmen
- Ebenso funktioniert `__subclasscheck__()` für `issubclass(A, C)`
- Eigentlich für ABCs gebraucht, aber auch generell nützlich?



Nur-Keyword-Argumente

- Argumente, die nur als Keyword übergeben werden können:

```
def foo(a, b, *, c=1):  
    pass
```

```
foo(1, 4, c=5)    # ok  
foo(a=1, 4, c=5) # ok  
foo(1, 4, 5)     # falsch!
```

*# Mit *args vereinbar:*

```
def bar(a, b, *args, c=1):  
    pass
```

```
bar(1, 4, 5, 7, c=6)
```

- Praktisch für variable Anzahl Argumente plus Flags (wie z.B. bei `min()` in 2.x)



Neue Metaklassen

- Metaklassen-Syntax:

```
class Foo(metaclass=Meta):  
    ...
```

- Mehr Kontrolle über Klassenerzeugung
- Zum Beispiel: geordnetes Klassen-Dict, gut für ORMs:

```
class Person(Table):  
    id = IntCol()      # Reihenfolge  
    name = StrCol()   # ist wichtig!  
    address = StrCol()
```



nonlocal-Statement

- Wie `global`, nur auf umgebende Funktionsnamensräume bezogen:

```
def foo():  
    p = 0  
    def inner():  
        nonlocal p  
        p = 1  
        return 42  
  
print(p)    # -> 0  
inner()  
print(p)    # -> 1
```



Kleinzeug

- Ellipsis-Literal: ... überall im Code
- Neues Buffer-Protokoll
- Klassendekoratoren
- ABCs für Zahlen: Integer, Rational etc.
- exec ist jetzt eine Funktion
- List-Comp-Variablen „leaken“ nicht mehr



Umbenannte Funktionen, Methoden, Attribute

- `reduce()` \Rightarrow `functools.reduce()`
- `reload()` \Rightarrow `imp.reload()`
- `raw_input()` \Rightarrow `input()`
- `__nonzero__()` \Rightarrow `__bool__()`
- `next()` \Rightarrow `__next__()`; dafür neues Builtin `next()`
- Funktionsattribute: `func_foo` \Rightarrow `__foo__`



Entfernter Ballast

- Vergleichsoperatoren für alle Objekte: `1 > "foo"`
- String-Konversion mit Backticks: `'a' == repr(a)`
- Alte Operatoren: `<>` und Literale: `1L`
- Alte Builtins: `apply()`, `input()`, `cmp()`, `callable()`, `coerce()`, `execfile()`
- Alte Typen: `basestring`



Entfernter Ballast (2)

- Tupel im Argument: `def foo((a, b), c):`
- `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback`
- `sys.exitfunc`



Und was ist mit der Library?

- Alte Module gelöscht (meist schon in 2.x deprecated)
- Einige Umbenennungen geplant: z.B. Queue → queue
- Neue Packages: dbm, html, http, tk, url, xmlrpc
- APIs bleiben unverändert
- Projekt läuft noch, neueste Infos in PEP 3108
- stdlib-SIG diskutiert Änderungen



Zusammenfassung



Umstieg leichtgemacht

- Mehrere Strategien:
 - Viele Backports in Python 2.6
 - 2to3: Konvertiert 2.x-Code nach 3.x (statische Analyse)
 - Automatischer Einsatz von 2to3 bei distutils-install
 - „Py3k“-Warnungen in 2.6 mit -3
- (Traum-)Ziel: Nur eine Version zu pflegen



Was jetzt?

- Keine Panik: 2.x-Serie wird noch länger weitergeführt
- Zukunftssicheren Code schreiben (keine alten Module, Iteratoren, new-style Klassen)
- Strikt zwischen Bytes und Text trennen, immer encode/decode verwenden
- Python 2.6 mit -3 verwenden



Der Umstieg

- Auf 2.6 portieren
- Wenn möglich, Backports in 2.6 nutzen (Problem für 2.5-Kompatibilität)
- Code auf Stand bringen, von dem mit 2to3 die Version für 3.0 automatisch generiert wird
- Abhängigkeiten berücksichtigen!
- C-Extensions: noch keine genauen Infos



Zu wenig gesehen?

- Dokumentation (größtenteils aktuell):
<http://docs.python.org/dev/3.0>
- „What's new?“:
<http://docs.python.org/dev/3.0/whatsnew/3.0>
- Download: <http://www.python.org/3.0>
- PEP 3xxx: <http://www.python.org/dev/peps/pep-3000>
- Mailingliste: python-3000@python.org
- Code (Subversion):
<http://svn.python.org/projects/python/branches/py3k>



Fragen?

