Python-Crashkurs

Andreas Abel

Lehrstuhl für Theoretische Informatik
Ludwig-Maximilians-Universität München

Wintersemester 2008/09
6.-10. Oktober 2008

Weitere Listenoperationen

- Modifikatoren:
  - `l.extend(l2)` bedeuted `l[len(l):] = l2`.
  - `l.remove(x)` löscht das erste Vorkommen von `x` in `l`. Fehler, falls `x not in l`.
- Read-only:
  - `l.index(x)` liefert die Position von `x` in `l`. Fehler, falls `x not in l`.
  - `l.count(x)` liefert Anzahl der Vorkommen von `x` in `l`.
  - `sorted(l)` liefert eine neue Liste, die Sortierung von `l`.
  - `reversed(l)` liefert einen Iterator, der die Umkehrung von `l` aufzählt.

Verwendungsmuster für Listen

- Stapel (*stack*): `append` und `pop()`.
- Warteschlange (*queue*): `append` und `pop(0)`.

Listenfunktionen höherer Ordnung

- `filter(test, sequence)` liefert eine neue Sequenz, deren Elemente den `test` bestehen.

```
filter(lambda x: x % 2 == 0, range(10))
```

- `map(f, sequence)` wendet `f` auf jedes Sequenzelement an und bildet daraus eine neue Sequenz.

```
map(lambda x: x*x*x, range(10))
map(lambda x,y: x+y, range(1,51), range(100,50,-1))
```

- `reduce(f, [a1,a2,a3,...,an])` berechnet `f(...f(f(a1,a2),a3),...,an)`

```
reduce(lambda x,y:x*y, range(1,11))
```

- `reduce(f, [a1,a2,...,an], e)` berechnet `f(...f(f(e,a1),a2),...,an)`

Listenkomprehension

- Lesbare Notation für `map`- und `filter`-Kombinationen.
- `[ e(x,y) for x in seq1 if p(x) for y in seq2 ]`

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
```

Löschen

- Löschen von Listenteilen:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

- Löschen von Variablen:

```
>>> del a
```

Weiterer Sequenztyp: Tupel

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> x, y, z = t
>>> empty = ()
>>> singleton = 'hello',    # trailing comma
```

Mengen

- `set(l)` erzeugt eine Menge aus Liste `l`.
- `list(s)` erzeugt eine Liste aus Menge `s`.
- `x in s` testet Mitgliedschaft.
- Operationen: `-` (Differenz), `|` (Vereinigung), `&` (Schnitt), `^` (Xor).
- `for v in s` iteriert über Menge (sortiert!).

Dictionaries

- Endliche Abbildungen, *hash maps*, *associative arrays*.
- Ungeordnete Menge von Paaren (Schlüssel, Wert).
- Jeder Schlüssel darf nur einmal vorkommen.
- Erzeugung mit `{ key1 : value1, ..., keyn : valuen }` oder

```
>>> tel = dict([('guido', 4127), ('jack', 4098)])
{'jack': 4098, 'guido': 4127}
```

- Zugriff mittels Schlüssel: `tel['jack']`.
- Einfügen/Ersetzen: `tel['me'] = 1234`.
- Löschen: `del tel['me']`.
- `tel.keys()` liefer Schlüsselliste, `tel.has_key('guido')` wahr oder falsch.

Dictionaries

- Python verwendet selbst dictionaries, z.B. um die exportierten Namen eines Moduls aufzulisten.
- Symboltabelle eines Interpreters.
- Iteration durch eine endliche Abbildung:

```
for k, v in tel.iteritems():
  print k, v
```

- Benamste Argumentsammlung:

```
def fun(arg, *args, **keyArgs): ...

fun (1, 2, 3, opt1=4, opt2=5)
```

- `args = [1,2]` und `keyArgs = {opt1:4, opt2:5}`.

Schleifentechniken

- Gleichzeitig Index und Element einer Sequenz iterieren:

```
l = ['tic', 'tac', 'toe']
for i, v  in enumerate(l):
    print i, v
```

- Simultan über zwei oder mehr Sequenzen iterieren:

```
for i, v in zip(xrange(len(l)),l):
    print i, v
```

- Sortierte/umgekehrte Reihenfolge:

```
for v in reversed(sorted(l)):
  print v
```

Wahrheitswerte (booleans)

- `0`, `''`, `[]`, `None`, etc. gelten als `False`.
- Andere Werte als `True` (auch Funktionen!).
- `is` vergleich Objektidentität: `[] == []` gilt, aber `[] is []` nicht. Jedoch wieder `5 is 5`.
- Vergleichskette: `a < b == c > d`.
- Die Boolschen Operatoren `not`, `and`, `or` *schließen kurz*.

```
def noisy(x): print x; return x

a = noisy(True) or noisy(False)
```

- Können auch mit nicht-boolschen Werten verwendet werden:

```
>>> '' or 'you' or 'me'
'you'
```

Vergleich von Sequenzen und anderen Typen

- Sequenzen werden lexikographisch verglichen, auch geschachtelt.

```
() < ('\x00',)
('a', (5, 3), 'c') < ('a', (6,) , 'a')
```

- Achtung! Der Vergleich von Werten unterschiedlichen Typs (außer Zahlen) liefert keinen Fehler, sondern ein willkürliches Ergebnis. (Dann werden die Typen verglichen!)

```
>>> "1" < 2
False
>>> () < ('\x00')
False
>>> [0] < (0,)
True
```

Module

- Jede Python-Datei ist ein Modul.
- `import myMod` importiert Modul `myMod`.
- Gesucht wird im aktuellen Verzeichnis und im `PYTHONPATH`.
- Zugriff auf den Modul-globalen Bezeichner `x` mit `myMod.x` (auch schreibend!).
- Der Modul-Kode wird ausgewertet, wenn das Modul zum *ersten Mal* importiert wird.
- Import in den Haupt-Namensraum mit

```
from myMod import myFun
from yourMod import yourValue as myValue

myFun(myValue) # qualification not necessary
```

- General abgeraten wird von `from myMod import *`.

Module als Skripts ausführen

- Mit `__name__` kann der Modulname abgefragt werden.
- Dieser ist `'__main__'` für das Hauptprogramm.

```
def fib(n): ...

if __name__ == '__main__':
  import sys
  fib(int(sys.argv[1]))
```

- Typische Anwendung: Unittests.

Module als Werte

- Ein Modul ist ein Objekt.

```
>>> fib = __import__('fibonacci')
>>> fib
<module 'fibonacci' from 'fibonacci.py'>
>>> fib.fib(10)
1 1 2 3 5 8
```

- `fib.__name__` ist der Name des Moduls.
- `fib.__dict__` enthält die definierten Namen des Moduls.
- `dir(fib)` ist `fib.__dict__.keys()`.

Standard- und eingebaute Module

- Siehe *Python Library Reference*, z.B. Modul `sys`.
- `sys.ps1` und `sys.ps2` enthalten die Prompts.
- `sys.path` den Modul-Suchpfad.
- Mit `import __builtin__` kann man die Liste der eingebauten Bezeichner explizit bekommen.

```
>>> import __builtin__
>>> dir(__builtin__)
```

Pakete

- Ein Verzeichnis, das eine (evtl. leere) `__init__.py` enthält ist ein Paket.
- Baumstrukturiert, Zugriff mit `paket1.paket2.modul`.

```
import packet.subpacket.module
print packet.subpacket.module.__name__

from packet.subpacket import module
print module.__name__
```

- Hat `paket/subpacket/__init__.py` die Anweisung `__all__ = ["module1", "module2"]`, so kann `from paket.subpacket import *` die beiden Module importieren.

Ausgabe-Formatierung

- `str(v)` erzeugt “Menschen-lesbare” String-Repräsentation von `v`
- `repr(v)` die für den Interpreter lesbare. Strings werden zitiert (quoted).
- `s.rjust(n)` füllt den String links mit Leerzeichen zu einer Gesamtlänge von `n` auf.
- `s.ljust(n)` und `s.center(n)` entsprechend.
- `s.zfill(n)` fügt führende Nullen an Zahl `s` in Stringrepräsentation an.
- `'-3.14'.zfill(8)` ergibt `'%08.2f' % -3.14`.
- Dictionary-Formatierung:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098 }
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d' % ta
Jack: 4098; Sjoerd: 4127
```

Datei Ein-/Ausgabe

- Standard-Ausgabe ist `sys.stdout`.
- `f = open(filename,mode)` erzeugt neues Datei-Objekt `f`.
- Modi sind: `'r', 'w', 'a', 'r+'` (read, write, append, read-write) plus evtl. suffix `b` (binary).
- `f.read()` gibt den ganzen Dateiinhalt als String zurück. `f.read(n)` die nächsten `n` Bytes.
- `f.readline()` eine Zeile, terminiert mit `'\n'`, falls die Zeile vollständig war. Leerer String bedeuted Dateiende.
- `f.readlines()` gibt eine Liste aller Zeilen zurück.
- Iteration über alle Zeilen:

```
for line in f: print l
```

Schreiben und Bewegen

- `f.write(s)` schreibt String `s`.
- `f.seek(offset,0)` geht zur Position `seek` ab Dateibeginn.
- `f.seek(offset,1)` geht zur Position `seek` ab der aktuellen Position.
- `f.seek(offset,2)` geht zur Position `seek` ab Dateiende. Bsp: `f.seek(-3,2)` zum drittletzten Byte.
- `f.close()` schliesst die Datei.

Pickling

- Beliebige Objekte können in eine Datei geschrieben werden.
- Modul `pickle`.
- `pickle.dump(x,f)` wandelt `x` in String um und schreibt ihn in Datei `f`.
- `x = pickle.load(f)` holt `x` aus Datei `f`.