Skriptsprachen: Python
Erweiterte Datenstrukturen

Jan Krüger, Alexander Sczyrba

Technische Fakultät
Universität Bielefeld

13. Februar 2018

Python

–

Listenbeschreibung

range(), xrange() - Python 2.x
Generatoren, Listbeschreibungen

Python 2.x : range() vs. xrange()

range([[start, [end]], step]) Erzeugt eine Liste bis `end`, ab `start` mit Schrittweite `step`

- Rückgabewert: `list` mit `int`
- Liste liegt persistent im Speicher vor

xrange([[start, [end]], step])) Generator mit denselben Eigenschaften wie `range` (lazy evaluation)

- Rückgabewert: `xrange`-Objekt, keine Liste
- Ist *iterable*
- Wird in Schleifen verwendet und ist speicheroptimiert
  $\rightarrow$ Besitzt dieselben Eigenschaft wie eine mit `range` erzeugte Liste
- Besonders performant, wenn nicht alle Elemente aufgerufen werden

Python 2.x : range() vs. xrange()

```
> range(3)
[0, 1, 2]
> g = xrange(3)
> g
xrange(3)
> for x, y in enumerate(range(2,4)):
>     print x, y/2.
0 1.0
1 1.5
> for x, y in enumerate(g):
>     print x, y/2.
0 1.0
1 1.5
```

Python 3 : range()

- Python 3 : `range()`, aber kein `xrange()`
- `range()` ersetzt `xrange()` aus Python 2
  - Rückgabewert: `range`-Objekt, keine Liste
  - Ist *iterable*
  - lazy
- kein vergleichbarer Ersatz fuer `range()` aus Python 2 !

Generatoren

Wie sieht das Erstellen einer Liste mit genau diesem (y) Inhalt aus?

```
> retL = []
> for y in range(2,4):
>     retL.append(y/2.)
> retL
[1.0, 1.5]
```

Geht das nicht auch effizienter? JA!

```
> retLfast = []
> g = ( y/2. for y in range(2, 4))
> for y in g:
>    retLfast.append(y)
> retL
[1.0, 1.5]
```

Generatoren

Der Ausdruck `( y/2. for y in range2, 4))` wird *Generator* genannt und kann in Schleifen als Erzeugendenfunktione ähnlich wie `xrange` verwendet werden.

- Definierte Rechenoperation wird nur ausgeführt, wenn wir einen weiteren Schleifendurchlauf haben
- Speichereffizient, weil nicht die gesamte Liste generiert wird
- Wenn ein `break` auftritt, kann an anderer Stelle ab dem nächsten Element weitergearbeitet werden

In unserem Fall ist die Listenerzeugung dennoch nicht schnell und elegant genug.

Listenbeschreibung

Der Generatorausdruck kann fast 1 : 1 verwendet werden, um eine Liste direkt zu erzeugen:

```
> [y/2. for y in range(2,4)]
[1.0, 1.5]
```

Dasselbe gilt auch für die Erzeugung von `dict` und `set`:

```
> #dict
> {y:y/2. for y in range2,4)}
{2: 1.0, 3: 1.5}


> #set
> {y/2. for y in range(2,4)}
{1.0, 1.5}
```

Listenbeschreibung

Was für eine Liste wird hier erzeugt?

```
> [ letter*N for N in range(1,5) for letter in 'abcd' ]
```

Listenbeschreibung

```
> [ letter*N for N in range(1,5) for letter in 'abcd' ]
['a',
 'b',
 'c',
 'd',
 'aa',
 'bb',
 'cc',
 'dd',
 'aaa',
 'bbb',
 'ccc',
 'ddd',
 'aaaa',
 'bbbb',
 'cccc',
 'dddd']
```

Übung – FASTA-Listenbeschreibung

Überlegt euch eine Listenbeschreibung für eine FASTA-Sequenz, bei der die resultierende Liste so aussehen könnte:
`["A", "A", "C", "T", "A", "G"]`

Erstellt anschließend eine Funktion `generate_fasta`, welche eine zufaellige FASTA Sequenz mit Hilfe der Listenbeschreibung generiert. Die Funktion erwartet die Länge der Sequenz und die Anzahl der Zeichen pro Zeile (optional, default:20) als Parameter. Der Rückgabewert ist ein formatierter Fasta-String.

Hinweis: Das Modul `random` enthaelt Funktionen um Zufallszahlen zu erzeugen. Nutzt die integrierte Hilfe von `ipython` um eine geeignete Funktion zu finden ...

Wahrheitgenerator

Das Folgende kleine Skriptlet verwendet einen besonders ausgefeilten Listengenerator, der euch die Wahrheit näher bringt.

```
> [ '%s␣is␣%s' % (item, bool(item)) for item in
   [0, 1, 0.0, 0.1, 0j, 1j, [], [0], (), (0,), {}, {'a':'A'}, None]]

['0␣is␣False',
 '1␣is␣True',
 '0.0␣is␣False',
 '0.1␣is␣True',
 '0j␣is␣False',
 '1j␣is␣True',
 '[]␣is␣False',
 '[0]␣is␣True',
 '()␣is␣False',
 '(0,)␣is␣True',
 '{}␣is␣False',
 "{'a':␣'A'}␣is␣True",
 'None␣is␣False']
```

Python

–

Funktionen höherer Ordnung

map(), filter()

Funktionen höherer Ordnung

Funktionen höherer Ordnung (high order functions) sind Funktionen, die Funktionen als Argumente erhalten können und wiederum Funktionen als Rückgabewerte haben können.

Beispiel aus Haskell:

```
> map :: (a -> b) -> [a] -> [b]
> map f []     = []
> map f (x:xs) = (f x):map f xs
```

`map (\x -> x^2) [1,2,3,4]` wird ausgewertet zu `[1,4,9,16]`

Aber wie sieht das nun in Python aus? Kurz zur Erinnerung: In Python ist selbst eine Funktion ein Objekt!

map()

`map(<FUNC>, <SEQ>[, <SEQ>, ...])` erwartet als Argumente genau eine Funktion und mindestens eine Sequenz. Werden mehr Sequenzen als eine Sequenz angeben, so werden diese als weitere Argumente an die Funktionen übergeben. Stimmen die Längen nicht überein, werden die Elemente entsprechend `None` gesetzt.

```
> def pow_2(n, e=2):
>  return n**e

> map(pow_2, range(1,5))
[1, 4, 9, 16]
> map(pow_2, range(1,5), range(2,6))
[1, 8, 81, 1024]
> map(lambda n:n**2, range(1,5))
[1, 4, 9, 16]
```

Übung – `map()` mit `lambda`

Forme den `lambda`-Ausdruck in `map()` von der vorherigen Folie entsprechend um, sodass die Funktionalität der `pow_2(n, e=2)`-Funktion entspricht.

Eigene Funktionen höherer Ordnung

Es lassen sich auch eigene Funktionen höherer Ordnung in Python schreiben. Hier ein kleines Beispiel für $f(x) = x + 3$ und $g(x) = f(f(x))$:

```
> def f(x):
>   return x + 3

> def twice(function, x):
>   return function(function(x))

> print(twice(f, 7))
```

Wie sähe hier das Ergebnis aus?

filter()

`filter(<FUNC> | None, <SEQ>)` erzeugt eine Liste, bei der die Anwendung von `FUNC` `True` ergibt. Ist `FUNC = None`, so werden nur die Elemente in die neue Sequenz übernommen, die `True` sind. Wenn `SEQ` ein Tuple oder ein String ist, so bleibt der Rückgabetyp entsprechend des Eingabetyps, ansonsten wird immer eine Liste zurückgegeben.

```
> def is_odd(n):
>  return n%2

> filter(is_odd, range(4))
[1, 3]
> filter(None, range(4)
[1, 2, 3]
```

Python

–

Vertiefung sort()

cmp(), key, reverse

sort()

Bisher habt ihr `sort()` nur mit den Standardwerten verwendet, dabei ist die Funktionen noch viel mächtiger. Kurze Erinnerung `sort([cmp[, key[, reverse]]])`:

cmp Eine neue Vergleichsfunktione, die -1, 0 oder 1 zurückgeben muss

key Funktion, die definiert, nach welcher Eigenschaft der Sequenzelemente sortiert werden soll (`key=str.lower` oder `lambda (k,v) : v`)

reverse Wenn `True`, wird die Sortierreihenfolge umgekehrt

sort()

```
> mydict = { 'b1': (1, 6),  'b3': (1, 3),
 'a4': (1, 2), 'b5': (1, 7), 'a6': (3, 7) }

 > sorted(mydict.items())
[('a4', (1, 2)),
 ('a6', (3, 7)),
 ('b1', (1, 6)),
 (b3',␣(1,␣3)),
␣('b5',␣(1,␣7))]
```

Die Funkton `sorted()` sortiert alle *iterable* Datentype und liefer eine Liste mit den sortierten Element zurück. Hier sehen wir, dass nach den *Keys* sortiert wird.

sort()

Wie können wir die Sortierung umkehren? → `reverse=True`

```
> sorted(mydict.items(), reverse=True)
[('b5', (1, 7)),
 ('b3', (1, 3)),
 ('b1', (1, 6)),
 ('a6', (3, 7)),
 ('a4', (1, 2))]
```

sort()

Nehmen wir an, dass das Value-Tupel einen Bruch definiert. Wie können wir nach diesem sortieren?

```
> def cmpN(x, y):
>  return cmp(float(x[1][0]) / x[1][1],
     float(y[1][0]) / y[1][1])

> sorted(mydict.items(), cmp=cmpN)
[('b5', (1, 7)),
 ('b1', (1, 6)),
 ('b3', (1, 3)),
 ('a6', (3, 7)),
 ('a4', (1, 2))]
```

sort()

Das war eine sehr unschöne Notation. Wieso nicht direkt auf das Tupel zugreifen? Wir müssen wenig rumbasteln...

```
> def cmpNew(x, y):
>  return cmp(float(x[0]) / x[1],
    float(y[0]) / y[1])

> sorted(mydict.items(), cmp=cmpNew,
    key=lambda t: t[1] )
[('b5', (1, 7)),
 ('b1', (1, 6)),
 ('b3', (1, 3)),
 ('a6', (3, 7)),
 ('a4', (1, 2))]
```

sort() – key() und cmp()

Bei der Verwendung von `cmp` und `key` gilt es einiges zu beachten:

- `cmp()` wird bei jedem Vergleich ausgeführt
- Die Funktion `key` wird pro Element einmalig ausgeführt, bevor diese im Sortierschritt mehrfach von `cmp()` Verwendung finden
- Optimal: Precomputation, die in `cmp()` vor dem eigentlichen Vergleich durchgeführt wird, bereits in `key` ausführen!

Übung – sort()

Lagert Funktionalität aus der `cmpNew`-Funktion aus, sodass das Precomputing bereits im `lambda`-Ausdruck einmalig durchgeführt wird und keine zusätzliche `cmp()` definiert werden muss.