Informatik I: Einführung in die Programmierung

27. Iteratoren und Generatoren

Albert-Ludwigs-Universität Freiburg

Bernhard Nebel

03.02.2017

1 Iteratoren

Iterierbare Objekte

- Wir haben den Begriff iterierbares Objekt bereits sehr oft erwähnt.
- Dies sind unter anderem Container-Objekte, über deren Elemente wir in `for`-Schleifen iterieren können.
- Sequenzen und ähnliche Objekte, wie Tupel, Listen, Strings, `dict`s und Mengen gehören dazu:

Python-Interpreter

```
>>> for el in set((1, 5, 3, 0)): print(el, end=' ')
...
0 1 3 5
```

- Was hier genau passiert, wollen wir uns genauer anschauen.

Das Iterator-Protokoll

- Ein Objekt ist iterierbar, wenn es das Iterator-Protokoll unterstützt.
- D.h. es muss die magische Methode `__iter__` besitzen, die einen neuen Iterator zurück liefert.
- Ein Iterator ist ein Objekt, das ebenfalls eine magische Methode `__iter__` besitzt, die `self` zurück gibt.
- Außerdem muss es eine magische Methode `__next__` besitzen, die das jeweilig nächste Element zurück liefert. Gibt es kein weiteres Element, soll die *Exception* StopIteration ausgelöst werden.
- Die `__iter__`-Methode kann auch mit der Funktion `iter(`*object*`)` aktiviert werden.
- Ebenso kann die `__next__`-Methode mit der Funktion `next(`*object*`)` aktiviert werden.

Die Implementation der `for`-Schleife

Mit Hilfe des Iterator-Protokolls können `for`-Schleifen auf die folgende Art auf `while`-Schleifen reduziert werden (im Python-Interpreter wird das tatsächlich effizienter gelöst):

`for`

```
for el in seq:
    do_something(el)
```

wird zu

`iterator`

```
iterator = iter(seq)          # erzeuge Iterator
while True:                   # durchlaufe Schleife
    try:
        el = next(iterator) # nächstes Element
        do_something(el)    # mache etwas damit
    except StopIteration:   # falls Ende-Ausnahme
        break               # verlasse die Schleife
```


Das Iterator-Protokoll bei der Arbeit

Python-Interpreter

```
>>> seq = ['spam', 'ham']
>>> iter_seq = iter(seq)
>>> iter_seq
<list_iterator object at 0x1094d8610>
>>> print(next(iter_seq))
spam
>>> print(next(iter_seq))
ham
>>> print(next(iter_seq))
Traceback (most recent call last): ...
StopIteration
```


Iterierbare Objekte vs. Iteratoren (1)

- Ein iterierbares Objekt ist ein Objekt, das (bei Aufruf von `iter()`) einen Iterator erzeugt, aber selbst keine `__next__`-Methode besitzt.
- Bei jedem Aufruf von `iter()` wird ein neuer Iterator erzeugt.
- Ein Iterator dagegen erzeugt keine neuen Iteratoren, aber liefert bei jedem Aufruf von `next()` ein neues Objekt aus dem Container.
- Da Iteratoren auch die `__iter__`-Methode besitzen, können Iteratoren an allen Stellen stehen, an denen ein iterierbares Objekt stehen kann (z.B. `for`-Schleife).
- Beim `iter()`-Aufruf wird der Iterator selbst zurück gegeben.
- `map` z.B. liefert ja beispielsweise einen Iterator und kann in `for-Schleifen` genutzt werden.

Iterierbare Objekte vs. Iteratoren (2)

- Iteratoren (z.B. `map`) können an den Stellen stehen, an denen ein iterierbares Objekt (z.B. eine Liste) stehen kann, aber es passiert etwas anderes!
- Iteratoren sind nach einem Durchlauf, der mit `StopIteration` abgeschlossen wurde, erschöpft, wie im nächsten Beispiel:

Python-Interpreter

```
>>> iterator = map(lambda x: x+1, range(2))
>>> for x in iterator:
...     for y in iterator:
...         print(x,y)
...
1 2
>>>
```

Iterierbare Objekte vs. Iteratoren (3)

- Wird bei jedem Start eines Schleifendurchlaufs ein neuer Iterator erzeugt, läuft alles wie erwartet:

Python-Interpreter

```
>>> for x in map(lambda x: x+1, range(2)):
...     for y in map(lambda x: x+1, range(2)):
...         print(x,y)
...
1 1
1 2
2 1
2 2
>>>
```


Weitere iterierbare Objekte

- Die range-Funktion liefert ein range-Objekt, das iterierbar ist.
- D.h. das Objekt liefert bei jedem iter()-Aufruf einen neuen Iterator.

Python-Interpreter

```
>>> range_obj = range(10)
>>> range_obj
range(0, 10)
>>> range_iter = iter(range_obj)
>>> range_iter
<range_iterator object at 0x108b10e70>
```


Iteratoren – selbst gestrickt

- Warum sind Iteratoren überhaupt interessant? Sie bieten:
  1. eine einheitliche Schnittstelle zum Durchlaufen von Elementen;
  2. die Möglichkeit eine Menge von Elementen zu durchlaufen ohne eine Liste aufbauen zu müssen (Speicher-schonend!);
  3. die Möglichkeit, unendliche Mengen zu durchlaufen (natürlich nur endliche Anfangsstücke!).
- Iteratoren können natürlich auch selbst definiert werden.
- Zum Beispiel können wir einen Iterator zum Aufzählen aller Fibonacci-Zahlen definieren (oder die Länge beschränken).

Fibonacci-Iterator

fibiter.py

```
class FibIterator():
    def __init__(self, max_n=0):
        self.max_n = max_n
        self.n, self.a, self.b = 0, 0, 1

    def __iter__(self):
        return self

    def __next__(self):
        self.n += 1
        self.a, self.b = self.b, self.a + self.b
        if not self.max_n or self.n <= self.max_n:
            return self.a
        else:
            raise StopIteration
```

FibIterator bei der Arbeit

Python-Interpreter


```
>>> f = FibIterator(10)
>>> list(f)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> list(f)
[]
>>> for i in FibIterator(): print(i)
...
1
1
2
3
5
...
```


2 Generatoren

Generatoren – die Idee

- Man kann zwar selbst Iteratoren erzeugen (siehe `FibIterator`), aber dies ist ziemlich umständlich.
- Generatoren bieten die Möglichkeit, Iteratoren mit Hilfe einer einfachen Funktionsdefinition zu erzeugen.
- Dazu wird innerhalb der Funktionsdefinition das Schlüsselwort `yield` benutzt.
- An dieser Stelle wird die Ausführung unterbrochen und ein Wert zurück gegeben. Danach wird beim nächsten `next()`-Aufruf direkt an dieser Stelle weiter gemacht.
- D.h. Generatoren speichern den Zustand in Form der Wertebelegung der lokalen Variablen und den aktuellen Ausführungspunkt.

Generatoren – ein erstes Beispiel

Python-Interpreter


```
>>> def gen(i): # sieht aus wie normale Funktion
...     i += 1
...     yield i # ist aber ein Generator
...     i += 1
...     yield i
...
>>> g = gen(5); g # Erzeuge Iterator
<generator object gen at 0x1043053c0>
>>> next(g) # erstes Element
6
>>> next(g) # zweites Element
7
>>> next(g) # Schluss!
Traceback ...
StopIteration
```

Generatoren vs. Funktionen

- Generatoren sehen aus wie Funktionen, geben aber Werte per `yield` (statt `return`) zurück.
- Wird ein Generator aufgerufen, so liefert er keinen Funktionswert sondern einen Iterator zurück.
- Dieser gibt dann bei den folgenden `next()`-Aufrufen die `yield`-Werte zurück.
- Kommt der Iterator zum Ende (bzw. wird ein `return` ausgeführt), dann wird die `StopIteration`-Ausnahme ausgelöst.

Fibonacci-Generator

`fibgen.py`

```
def fibgen(max_n=0):
    n, a, b = 0, 0, 1
    while max_n == 0 or n < max_n:
        n += 1
        a, b = b, a + b
        yield a
```

Python-Interpreter

```
>>> list(fibgen(10))
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> for i in fibgen(): print(i, end=' ')
...
1 1 2 3 5 8 13 21 34 55 89 ...
```


Rekursive Generatoren

- Genauso wie Funktionen können auch Generatoren rekursiv definiert werden.
- Beispiel: Alle Permutationen (Anordnungen) erzeugen.

Python-Interpreter

```
>>> def perm(seq):
...     if not seq: yield []
...     else:
...         for i in range(len(seq)):
...             for cc in perm(seq[:i]+seq[i+1:]):
...                 yield [seq[i]] + cc
...
>>> list(perm('lion'))
[['l', 'i', 'o', 'n'], ['l', 'i', 'n', 'o'],...]
```


3 Das Modul `itertools`

itertools (1)

- Das Modul `iterttols` bietet viele Generatoren an, die man in standardmäßig benötigt.
- Außerdem gibt es Kombinationen und Modifikationen von Iteratoren.
- Generell werden immer Iteratoren zurück gegeben.
- Wir schauen uns einige wichtige Beispiele an.
- `accumulate(iterable, func=operator.add)`:
  Akkumuliert über einen Iterator. Man kann statt der Addition auch eine andere 2-stellige Funktion nutzen.
  Beispiel: `accumulate([1, 2, 3, 4])` → `1 3 6 10`
- `chain(*iterables)`:
  Verkettet iterierbare Objekte. Beispiel:
  `chain('ABC, 'DEF')` → `'A' 'B' 'C' 'D' 'E' 'F'`

itertools (2)

- `combinations(iterable, r)`:
  Erzeugt alle Kombinationen der Länge *r*. Beispiel:
  `combinations('ABCD', 2)` → `('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')`
  Elemente werden dabei hier und im Folgenden auf Grund ihrer Position identifiziert, nicht auf Grund ihres Wertes!
- `combinations_with_replacement(iterable, r)`:
  Kombinationen mit Wiederholungen. Beispiel:
  `combinations_with_replacement('ABC', 2)` → `('A', 'A') ('A', 'B'), ('A', 'C'), ('B', 'B') ('B', 'C') ('C', 'C')`
- `cycle(iterable)`:
  Erzeugt einen unendlichen Iterator, der immer wieder über `iterable` iteriert. Beispiel: `cycle('ABC')` → `'A' 'B' 'C' 'A' 'B' 'C' 'A' ...`

itertools (3)

- `islice(iterable, stop)`
  `islice(iterable, start, stop)`
  `islice(iterable, start, stop, step)`:
  Slice-Funktion für Iteratoren. Beispiel:
  `islice('ABCDEF', 2, 4)` → `'C' 'D'`
- `permutations(iterable, r=None)`:
  Permutationen der Länge *r* (bzw. aller Elemente)
- `product(*iterables, repeat=1)`:
  Kartesisches Produkt bzw. `repeat`-faches Produkt.
- `repeat(object, times=None)`:
  Erzeugt Iterator, der Objekt `times`-fach oder unbegrenzt oft wiederholt. Beispiel:
  `map(pow, range(5), repeat(2))` → `0 1 4 9 16`

itertools (4)

- `starmap(function, iterable)`:
  Ähnlich wie `map`, allerdings für den Fall, dass die Argumente für `function` bereits in Tupel zusammengefasst wurden. Beispiel:
  `starmap(lambda x, y: x+' '+y, [('nice','restaurant'), ('dirty','fork')])` → `'nice restaurant' 'dirty fork'`

Zusammenfassung

UNI FREIBURG

- Iterierbare Objekte besitzen eine Methode `__iter__`, die (z.B. mit Hilfe der Funktion `Iter()` oder in einer `for`-Schleife einen Iterator erzeugen.
- Iteratoren besitzen die Methoden `__iter__` und `__next__`.
- Mit Aufrufen der `__next__`-Methode oder `next()`-Funktion, erhält man alle Elemente.
- Generatoren sehen aus wie Funktionen, geben ihre Werte aber mit `yield` zurück.
- Ein Generatoraufruf liefert einen Iterator, der bei jedem `next`-Aufruf bis zum nächsten `yield` läuft.
- Das Modul `itertools` stellt eine Menge von Generatoren bereit.