

Programmieren in Python

10. Iteratoren und Generatoren

Robert Mattmüller

Albert-Ludwigs-Universität Freiburg

Handlungsplanungs-Praktikum
Wintersemester 2010/2011

1 / 28

Iteratoren und Generatoren

Überblick über diese Lektion:

- ▶ Iteratoren
- ▶ Generatoren
- ▶ Generator Comprehensions, List Comprehensions und Set Comprehensions

2 / 28

Iteratoren und Generatoren

Überblick über diese Lektion:

- ▶ **Iteratoren**
- ▶ Generatoren
- ▶ Generator Comprehensions, List Comprehensions und Set Comprehensions

3 / 28

for und `__iter__` (1)

Die Methode `__iter__` ist im Zusammenhang mit `for`-Schleifen relevant. Jetzt lohnt es sich, die Funktionsweise von `for`-Schleifen im Detail zu besprechen:

- ▶ Zu Beginn der Schleife wird die (bisher noch nicht besprochene) Builtin-Funktion `iter` mit dem Argument `obj` aufgerufen.
- ▶ Die Aufgabe von `iter` besteht darin, ein Objekt zu liefern, mit dessen Methoden `obj` durchlaufen werden kann. Ein solches Objekt bezeichnet man als *Iterator*.
- ▶ `iter(obj)` erledigt diese Aufgabe, indem es `obj.__iter__()` aufruft. Hier liegt also die eigentliche Verantwortung.
- ▶ Das Resultat von `iter(obj)` wird in einer internen Variable gebunden, die wir `cursor` nennen wollen (sie ist nicht vom Programm aus sichtbar, hat also keine Namen):

```
cursor = iter(obj)
```

4 / 28

for und __iter__ (2)

- ▶ Bei jedem Schleifendurchlauf wird `next(cursor)` aufgerufen und das Ergebnis an die Schleifenvariable `x` gebunden:

```
x = next(cursor)
```

- ▶ Dabei ruft `next(cursor)` die Methode `cursor.__next__()` auf.
- ▶ Die Schleife endet, wenn `next(cursor)` eine Ausnahme des Typs `StopIteration` erzeugt.

5 / 28

for-Schleifen als while-Schleifen

Die folgenden beiden Programme sind also äquivalent, sieht man mal von der Sichtbarkeit der Cursor-Variable ab:

```
for x in obj:  
    <do something>
```

```
_cursor = iter(obj)  
while True:  
    try:  
        x = next(_cursor)  
    except StopIteration:  
        break  
    <do something>
```

6 / 28

Iterator-Beispiel (1)

Ein Iterator ist somit ein Objekt mit einer `__next__`-Methode, die ggf. die Ausnahme `StopIteration` erzeugt. Ein Beispiel:

squares_iter.py

```
class Squares(object):  
    def __init__(self, max_index):  
        self.max_index = max_index  
        self.current_index = 0  
    def __iter__(self):  
        return self  
    def __next__(self):  
        if self.current_index >= self.max_index:  
            raise StopIteration  
        result = self.current_index ** 2  
        self.current_index += 1  
        return result
```

7 / 28

Iterator-Beispiel (2)

Python-Interpreter

```
>>> from squares_iter import Squares  
>>> sq = Squares(5)  
>>> for x in sq:  
...     print(x)  
...  
0  
1  
4  
9  
16
```

8 / 28

Iteratoren und Generatoren

Überblick über diese Lektion:

- ▶ Iteratoren
- ▶ **Generatoren**
- ▶ Generator Comprehensions, List Comprehensions und Set Comprehensions

9 / 28

Generatoren

- ▶ Iteratoren sind so nützlich, dass es ein spezielles Konstrukt in Python gibt, das das Erzeugen von Iteratoren erleichtert: *Generatoren*.
- ▶ Generatoren sind Funktionen, die Iteratoren erzeugen. Äußerlich sieht ein Generator aus wie eine Funktion, nur dass er anstelle von (oder zusätzlich zu) `return`-Anweisungen `yield`-Anweisungen benutzt.
- ▶ Ein Beispiel:

`generate_food.py`

```
def food():  
    yield "ham"  
    yield "spam"  
    yield "jam"
```

10 / 28

Generatoren (2)

Im Gegensatz zu Funktionen können Generatoren *mehrere Werte hintereinander* erzeugen:

Python-Interpreter

```
>>> from generate_food import food  
>>> myiter = food()  
>>> print(next(myiter))  
ham  
>>> print("do something else")  
do something else  
>>> print(next(myiter))  
spam  
>>> print(next(myiter))  
jam  
>>> print(next(myiter))  
...(Traceback mit StopIteration)
```

11 / 28

Was tut ein Generator? (1)

- ▶ Ein Generator kann als Funktion aufgerufen werden. Er liefert dann ein Iterator-Objekt zurück. Der Code innerhalb der Generator-Definition wird zunächst nicht ausgeführt.
- ▶ Jedesmal, wenn die `__next__`-Methode des zurückgelieferten Iterators aufgerufen wird, wird der Code innerhalb des Generators so lange ausgeführt, bis er auf eine `yield`-Anweisung stößt.
- ▶ Bei Erreichen von `yield obj` wird `obj` als Ergebnis des `__next__`-Aufrufs zurückgeliefert und der Generator wieder (bis zum nächsten Aufruf) unterbrochen.
- ▶ Wenn die Funktion beendet wird oder eine `return`-Anweisung erreicht wird, wird eine `StopIteration`-Ausnahme erzeugt.

Anmerkung: In Generatoren ist nur `return`, nicht aber `return obj` erlaubt.

12 / 28

Was tut ein Generator? (2)

- ▶ Der gesamte Zustand des Generators wird zwischen den Aufrufen gespeichert; man kann ihn also so schreiben, als würde er nie unterbrochen werden.
- ▶ Man kann mit derselben Generator-Funktion mehrere Iteratoren erzeugen, ohne dass diese sich gegenseitig beeinflussen — jeder Iterator merkt sich den Zustand „seines“ Generator-Aufrufs.

13 / 28

Generatoren: Terminologie

- ▶ In Python wird sowohl der Generator selbst (`food` im Beispiel) als auch der Rückgabewert der Funktion (`myiter` im Beispiel) als *Generator* bezeichnet.
- ▶ Wenn man genau sein will, nennt man `food` eine *Generatorfunktion* und `myiter` einen *Generator-Iterator*.

14 / 28

Zurück zum Squares-Beispiel

Hier noch einmal unser altes Beispiel:

`squares_iter.py`

```
class Squares(object):
    def __init__(self, max_index):
        self.max_index = max_index
        self.current_index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current_index >= self.max_index:
            raise StopIteration
        result = self.current_index ** 2
        self.current_index += 1
        return result
```

15 / 28

Das Squares-Beispiel mit Generatoren

Mit Generatoren erreichen wir dasselbe sehr viel einfacher:

`squares_generator.py`

```
def squares(max_index):
    for i in range(max_index):
        yield i ** 2
```

Anwendung:

Python-Interpreter

```
>>> from squares_generator import squares
>>> for x in squares(5):
...     print(x, end=' ')
...
0 1 4 9 16
```

16 / 28

Unendliche Generatoren

Da Generator-Funktionen nur „on demand“ ausgeführt werden, können sie auch unendliche Berechnungen anstellen. Das ist nicht einmal selten:

infinite_squares.py

```
def squares():
    i = 0
    while True:
        yield i ** 2
        i += 1
```

17 / 28

Unendliche Generatoren (2)

Benutzung des unendlichen Generators:

Python-Interpreter

```
>>> from infinite_squares import squares
>>> for x in squares():
...     print(x, end=' ')
...     if x >= 100:
...         break
...
0 1 4 9 16 25 36 49 64 81 100
```

18 / 28

Das Modul itertools

Das Modul `itertools` enthält nützliche Hilfsfunktionen im Zusammenhang mit Iteratoren und Generatoren.

Einige einfache Beispiele:

- ▶ `count([start=0])`:
Generiert die Zahlen `start`, `start + 1`, `start + 2`, ...
 - ▶ Beispiel: `count()` generiert 0, 1, 2, 3, 4, 5, ...
- ▶ `cycle(iterable)`:
Durchläuft `iterable` zyklisch.
 - ▶ Beispiel: `cycle("ab")` generiert "a", "b", "a", "b", "a", "b", ...

19 / 28

Squares mit `itertools.count`

squares_for_the_last_time_i_promise.py

```
import itertools

def squares():
    for i in itertools.count():
        yield i ** 2
```

20 / 28

Pythagoräische Tripel als Generator

Für die, die sich noch an die Übungen zu Lektion 3 erinnern:

```
pythagorean_triples_generator.py
import itertools

def pythagorean_triples():
    for z in itertools.count(1):
        for x in range(1, z):
            for y in range(x, z):
                if x * x + y * y == z * z:
                    yield (x, y, z)

for triple in pythagorean_triples():
    print(triple)
```

Das Programm erzeugt Tripel bis zum Abwinken (mit Strg+C).

21 / 28

Iteratoren und Generatoren

Überblick über diese Lektion:

- ▶ Iteratoren
- ▶ Generatoren
- ▶ Generator Comprehensions, List Comprehensions und Set Comprehensions

22 / 28

Generator/List/Set Comprehensions

Häufig schreibt man Code nach diesem Schema:

```
mylist = []
for char in "spam":
    mylist.append(char.upper() + char)
```

oder

```
def mygen():
    for char in "spam":
        yield char.upper() + char
```

oder

```
myset = set()
for char in "spam":
    myset.add(char.upper() + char)
```

Für solche Zwecke gibt es Kurzschreibweisen, die man als *List*, *Generator* bzw. *Set Comprehensions* bezeichnet.

23 / 28

Generator/List/Set Comprehensions: Syntax

- ▶ Die Begriffe leiten sich von *set comprehensions* im mathematischen Sinn ab; so bezeichnet man im Englischen folgendes Schema zur Mengennotation aus der Mathematik:

$$M = \{ 3x \mid x \in A, x \geq 10 \}$$

- ▶ Eine ähnliche Notation ist auch in Python möglich:
 - ▶ Generator Comprehension:
`M = (3 * x for x in A if x >= 10)`
 - ▶ List Comprehension:
`M = [3 * x for x in A if x >= 10]`
 - ▶ Set Comprehension:
`M = {3 * x for x in A if x >= 10}`
- ▶ Ferner gibt es Dictionary Comprehensions:
 - ▶ `D = {x : 3 * x for x in A if x >= 10}`

24 / 28

Generator/List/Set Comprehensions: Beispiele

Zurück zu unseren Beispielen:

```
mylist = []
for char in "spam":
    mylist.append(char.upper() + char)
```

↪ `mylist = [char.upper() + char for char in "spam"]`

```
def mygen():
    for char in "spam":
        yield char.upper() + char
```

↪ `mygen = (char.upper() + char for char in "spam")`

```
myset = set()
for char in "spam":
    myset.add(char.upper() + char)
```

↪ `myset = {char.upper() + char for char in "spam"}`

25 / 28

Generator/List/Set Comprehensions: Allgemeine Form

Generator/List/Set Comprehensions haben allgemein die folgende Form (Einrückung dient nur der Verdeutlichung):

```
<ausdruck> for <var1> in <iterable1>
            for <var2> in <iterable2> ...
            if <bedingung1>
            if <bedingung2> ...
```

Dabei muss ein for-Teil vorhanden sein, während die if-Teile optional sind. Äquivalenter Code (für den Fall der Generator Comprehension):

```
for <var1> in <iterable1>:
    for <var2> in <iterable2>:
        ...
        if <bedingung1>:
            if <bedingung2>:
                yield <ausdruck>
```

26 / 28

Generator/List/Set: Anmerkungen

Einige abschließende Anmerkungen:

- ▶ Bei Generator Comprehensions kann man (genau wie bei Tupeln) die äußeren Klammern weglassen, sofern dadurch keine Mehrdeutigkeit entsteht:

Python-Interpreter

```
>>> print(sum(x * x for x in (2, 3, 5)))
```

```
38
```

- ▶ List Comprehensions sind eigentlich unnötig, da man mit `list(<generator comprehension>)` den gleichen Effekt mit ähnlichem Aufwand erreichen kann. Sie sind historisch älter als Generator Comprehensions.
- ▶ Set und Dictionary Comprehensions wurden zuletzt eingeführt (in Python 3).

27 / 28

Pythagoräische Tripel kompakt

Vielleicht nicht unbedingt übersichtlich, aber möglich:

`pythagoras_reloaded.py`

```
from itertools import count
p = ((x,y,z) for z in count(1) for x in range(1, z)
     for y in range(x, z) if x * x + y * y == z * z)
```

28 / 28