LU04h - Generatoren

Generatoren sind eine einfache Möglichkeit, Iteratoren in Python zu erstellen. Sie ermöglichen es, durch eine Sequenz von Werten zu iterieren, ohne die gesamte Sequenz im Speicher zu speichern. Dies wird durch die Verwendung des Schlüsselworts `yield` anstelle von `return` in einer Funktion erreicht.

Was sind Generatoren?

Ein Generator ist ein spezieller Typ von Iterator, der mit einer Funktion und dem Schlüsselwort `yield` erstellt wird. Im Gegensatz zu normalen Funktionen, die einen Wert zurückgeben und dann beendet werden, „pausieren“ Generatoren ihren Zustand und setzen die Ausführung fort, wenn der nächste Wert benötigt wird.

Wichtige Eigenschaften

- **Lazy**: Werte werden erst erzeugt, wenn sie gebraucht werden (z.B. im `for`-Loop oder mit `next()`).
- **Einmal konsumierbar**: Ein Generator läuft von vorne nach hinten durch. Was „verbraucht“ ist, ist weg.
- **Zustand bleibt erhalten**: Lokale Variablen behalten ihren Wert zwischen zwei `yield`-Schritten.

Animation: wann wird eigentlich gerechnet?

Diese drei Eigenschaften sind im Code nicht sichtbar – die Animation macht sie sichtbar. Der erste Tab zeigt, dass der Aufruf einer Generator-Funktion **keine einzige Zeile** ausführt, wo `yield` anhält und was dabei stehen bleibt. Der zweite Tab vergleicht Generator Expression und List Comprehension und zeigt, warum ein zweites `sum()` auf demselben Generator stillschweigend `0` ergibt.

Syntax

Die Erstellung eines Generators ist ähnlich wie die einer normalen Funktion, aber anstelle des `return`-Schlüsselworts verwenden wir `yield`:

```
def my_generator():
    yield 1
    yield 2
    yield 3
```

Generator vs. normale Funktion

Eine normale Funktion liefert genau einen Rückgabewert und endet. Ein Generator liefert mehrere Werte nacheinander:

```
def normal_function():
    return 1
    return 2  # wird nie erreicht

def generator_function():
    yield 1
    yield 2  # wird erreicht, wenn weiter iteriert wird

print(normal_function())        # 1
gen = generator_function()
print(next(gen))                # 1
print(next(gen))                # 2
```

Beispiele

Einfacher Generator

Ein einfacher Generator, der die Zahlen 1 bis 3 zurückgibt:

```
def simple_generator():
    yield 1
    yield 2
    yield 3

gen = simple_generator()
print(next(gen))  # 1
print(next(gen))  # 2
print(next(gen))  # 3
```

Generator mit Schleife

Ein Generator, der die Quadrate der Zahlen von 1 bis n zurückgibt:

```
def square_numbers(n):
    for i in range(1, n + 1):
        yield i * i

square = square_numbers(5)
print(next(square)) # 1
print(next(square)) # 4
print(next(square)) # 9
```

```
print(next(square)) # 16
print(next(square)) # 25
```

Generator Expression

Generator Expressions sind eine kompaktere Art, Generatoren zu erstellen, ähnlich wie List Comprehensions:

```
squared = (x * x for x in range(6))
print(next(squared))  # 0
print(next(squared))  # 1
```

Typische Verwendung: for-Loop statt next()

In der Praxis nutzt man Generatoren meistens im `for`-Loop, weil der Loop automatisch stoppt, wenn der Generator fertig ist:

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

for value in countdown(3):
    print(value)

# Output:
# 3
# 2
# 1
```

StopIteration: Was passiert, wenn der Generator fertig ist?

Wenn ein Generator keine Werte mehr liefern kann, endet er. Bei `next()` führt das zu einer Exception `StopIteration`:

```
gen = (x for x in range(2))
print(next(gen))  # 0
print(next(gen))  # 1

# nächster Aufruf -> StopIteration
# print(next(gen))
```

Generatoren sind Iteratoren

Ein Generator ist selbst ein Iterator: `iter(gen)` gibt den Generator zurück, und `next(gen)` liefert den nächsten Wert.

```
gen = (x for x in range(3))
print(iter(gen) is gen)  # True
```

Vorteile von Generatoren

- **Speichereffizienz**: Generatoren speichern nicht alle Werte im Speicher, sondern generieren sie „on-the-fly“. Dies ist besonders nützlich, wenn mit großen Datenmengen gearbeitet wird.
- **Einfachheit**: Generatoren sind einfacher zu implementieren als benutzerdefinierte Iteratoren.
- **Vielseitigkeit**: Generatoren können für jede Art von Datenquelle verwendet werden, nicht nur für Listen oder Sequenzen.
- **Abbruch möglich**: Man kann früh aufhören zu iterieren (z.B. nach dem ersten Treffer). Der Rest wird gar nie berechnet.

Verständnis des ''yield''-Schlüsselworts

Das Schlüsselwort `yield` in Python wird verwendet, um einen Wert aus einer Funktion zurückzugeben und den aktuellen Zustand der Funktion zu speichern. Wenn die Funktion das nächste Mal weiter iteriert wird, wird sie von der Stelle fortgesetzt, an der sie zuletzt angehalten wurde, und nicht von Anfang an.

Mini-Zusammenfassung

Begriff	**Bedeutung**
`yield`	Liefert einen Wert und pausiert die Funktion (Zustand bleibt erhalten).
Generator	Iterator, der Werte „lazy“ erzeugt.
`next(gen)`	Liefert den nächsten Wert (oder `StopIteration`, wenn fertig).
Generator Expression	Kurzform: `(expression for item in iterable if condition)`

M323-LU04, M323-D1B

© Kevin Maurizi

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/m323/learningunits/lu04/generatoren

Last update: **2026/09/01 11:16**