**Generators**

Generators are a simple and powerful possibility to create or to generate iterators. On the surface they look like functions, but there is both a syntactical and a semantic difference. Instead of return statements you will find inside of the body of a generator only yield statements, i.e. one or more yield statements.

1. Another important feature of generators is that the local variables and the execution start is automatically saved between calls.
2. Instead, the new call to a generator function will resume execution right after the yield statement in the code, where the last call exited
3. When the Python interpreter finds a yield statement inside of an iterator generated by a generator, it records the position of this statement and the local variables, and returns from the iterator.
4. The next time this iterator is called, it will resume execution at the line following the previous yield statement.
5. There may be more than one yield statement in the code of a generator or the yield statement might be inside the body of a loop.
6. But the crucial advantage of generators consists in automatically creating the methods \_\_iter\_\_() and next().
7. The following is a simple example of a generator, which is capable of producing four city names:

def city\_generator():

yield("Konstanz")

yield("Zurich")

yield("Schaffhausen")

yield("Stuttgart")

>>> from city\_generator import city\_generator

>>> x = city\_generator()

>>> print x.next()

Konstanz

>>> print x.next()

Zurich

>>> print x.next()

Schaffhausen

>>> print x.next()

Stuttgart

>>> print x.next()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

StopIteration

>>>

1. Can we send a reset to an iterator is a frequently asked question, so that it can start the iteration all over again. There is no reset, but it's possible to create another generator. This can be done e.g. by having the statement "x = city\_generator()" again.

Method of working:

1. A generator is called like a function. It's return value is an iterator object. The code of the generator will not be executed in this stage.
2. The iterator can be used by calling the next method. The first time the execution starts like a function, i.e. the first line of code within the body of the iterator. The code is executed until a yield statement is reached.
3. yield returns the value of the expression, which is following the keyword yield. This is like a function, but Python keeps track of the position of this yield and the state of the local variables is stored for the next call. At the next call, the execution continues with the statement following the yield statement and the variables have the same values as they had in the previous call.
4. The iterator is finished, if the generator body is completely worked through or if the program flow encounters a return statement without a value.

### Recursive Generators

1. ike functions generators can be recursively programmed.

Ex: def permutations(items):

n = len(items)

if n==0: yield []

else:

for i in range(len(items)):

for cc in permutations(items[:i]+items[i+1:]):

yield [items[i]]+cc

for p in permutations(['r','e','d']): print ''.join(p)

for p in permutations(list("game")): print ''.join(p)

### A Generator of Generators

The second generator of our Fibonacci sequence example generates an iterator, which can theoretically produce all the Fibonacci numbers, i.e. an infinite number. But you shouldn't try to produce all these numbers, as we would do in the following example:

list(fibonacci())

This will show you very fast the limits of your computer.   
In most practical applications, we only need the first n elements of an "endless" iterator. We can use another generator, in our example firstn, to create the first n elements of a generator g:

def firstn(g, n):

for i in range(n):

yield g.next()

The following script returns the first 10 elements of the Fibonacci sequence:

#!/usr/bin/env python

def fibonacci():

"""Ein Fibonacci-Zahlen-Generator"""

a, b = 0, 1

while True:

yield a

a, b = b, a + b

def firstn(g, n):

for i in range(n):

yield g.next()

print list(firstn(fibonacci(), 10))