Workshop Python

Teil III

1. **Kapitel 6: Exceptions**
2. **Kapitel 7: Funktionale Programmierung**
3. **Kapitel 8: Die Standard Library**

Teil 6

Exceptions

Exceptions oder *Falls mal was daneben geht:*

```
>>> 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Beim Auftreten einer Exception geht der Python-Interpreter so vor:

- Der Programmfluss wird sofort angehalten
- Vom Stack herunter wird geprüft, ob ein benutzerdefinierter *Exception Handler* die Exception verarbeitet
- Falls kein Exception Handler gefunden wurde, tritt der *Default Exception Handler* in Kraft:
  - Ausgabe eines Stack Traceback
  - Ausgabe der aufgetretenen Exception
  - Exit mit Return Code 1

```
def func1():
    print 5/0

def func2():
    func1()

def func3():
    func2()

func3()
```

```
Traceback (most recent call last):
  File "./except.py", line 10, in <module>
    func3()
  File "./except.py", line 8, in func3
    func2()
  File "./except.py", line 5, in func2
    func1()
  File "./except.py", line 2, in func1
    print 5/0
ZeroDivisionError: integer division or modulo by zero
```

Exceptions können abgefangen und verarbeitet werden mittels **try / except**:

```
try:
   <statements>
except <exceptionType>:
   <statements>
[except <exceptionType>: ...]
```

```
a = 5
b = 0
try:
   c = a / b
except ZeroDivisionError:
   print "Sorry, durch 0 teilen ist verboten!"
```

Eigene Exceptionhandler werden sehr häufig in Python-Code eingesetzt

Es ist *einfacher* und *performanter*, selten vorkommende Ausnahmen abzufangen als vor kritischem Code entsprechende Tests durchzuführen, und vermeidet (aufwendige und evtl. fehlerträchtige) Tests.

Python-Philosophie *"It is easier to ask forgiveness than permission"*

```
try:
   <code which might fail with quite low chance>
except <exceptionType>:
   <handling of very rare exception situation>
```

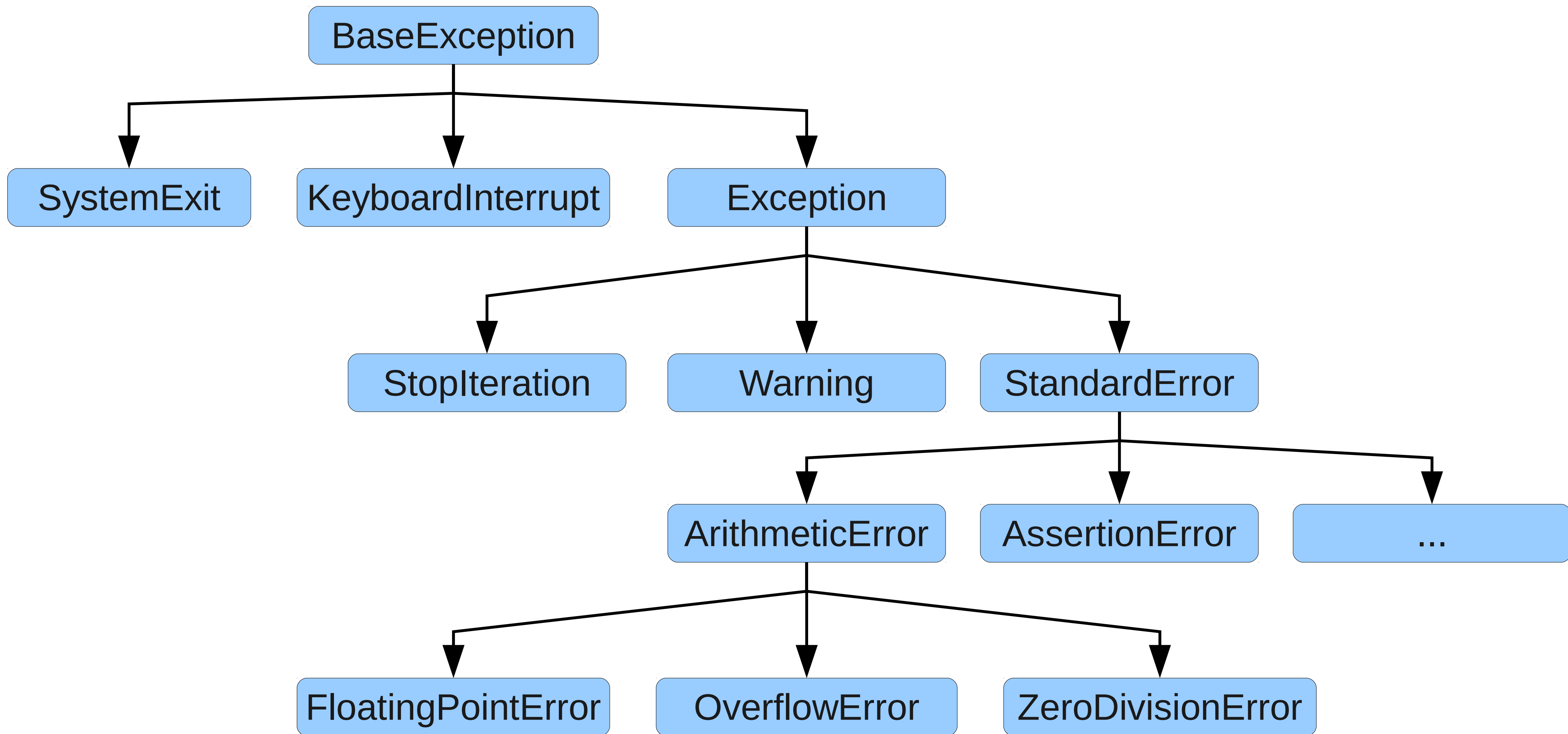
```
if test_if_everything_will_be_okay():
   <code>
else:
   <handling of very rare exception situation>
```

Die abgefangene Exception kann mittels **as** übergeben werden:

```
try:
   <statements>
except <exceptionType> as <value>:
   <statements>
```

```
a = 5
b = 0
try:
   c = a / b
except ArithmeticError as e:
   print "Ein Fehler beim Rechnen ist passiert:"
   print repr(e)
```

```
Ein Fehler beim Rechnen ist passiert:
ZeroDivisionError('integer division or modulo by zero',)
```



Die komplette `exceptions` - Hierarchie bekommt man mit:

```
>>> import exceptions
>>> help(exceptions)
```

`except` ohne Angabe einer Exceptionklasse fängt *jede* verbleibende Exception ab:

```
try:
   <statements>
except:
   <statements>
```

```
a = 5
b = 0
try:
   c = a / b
except ArithmeticError as e:
   print "Ein Fehler beim Rechnen ist passiert:" , repr(e)
except:
   print "Irgendwas anderes ging daneben ..."
```

`else` wird dann ausgeführt, falls keine Exception aufgetreten ist:

```
try:
   <statements>
else:
   <statements>
```

```
a = 5
b = 0
try:
   c = a / b
except ArithmeticError as e:
   print "Ein Fehler beim Rechnen ist passiert:" , repr(e)
except:
   print "Irgendwas anderes ging daneben ..."
else:
   print "Das ging aber glatt über die Bühne!"
```

`finally` wird immer ausgeführt, unabhängig ob eine Exception aufgetreten ist:

```
try:
   <statements>
finally:
   <statements>
```

```
a = 5
b = 0
try:
   c = a / b
except ArithmeticError as e:
   print "Ein Fehler beim Rechnen ist passiert:" , repr(e)
except:
   print "Irgendwas anderes ging daneben ..."
else:
   print "Das ging aber glatt über die Bühne!"
finally:
   print "Egal was passiert ist, erstmal aufräumen ..."
```

```
raise <exceptionType>
```

`raise` triggert eine Exception.

Selbst getriggerte Exceptions sind meist **RuntimeErrors**:

```
# irgendwas ging daneben
raise RuntimeError("Parameter außerhalb der Grenzen!")
```

```
raise <exceptionType>
```

raise triggert eine Exception.

Beispielanwendung: Schreiben eigener Exceptions

```
class MatrixError(ArithmeticError):

    def __init__(self, val):
        self.value = val

    def __str__(self):
        return "Wrong matrix dimensions: " + str(self.value)


class Matrix(object):
    ...
    def __mul__(self, matrix2):
        if (<wrong matrix dimensions>):
            raise MatrixError(<additional error information>)
```

```
raise
```

`raise` ohne Argument triggert nochmal die letzte Exception.

Anwendung z.B.: Um Exceptions in Exceptionshandlers 'durchzureichen':

```
try:
   <critical code>
except:
   print <some debugging information>
   raise
```

```
assert <test> [, <"Message">]
```

assert wirft eine Exception abhängig von einer Bedingung:

```
assert i < 5, "Der Index ist zu groß!"
```

assert nur für Debugging-Zwecke einsetzen!

assert Statements werden aus dem Python Bytecode entfernt, falls das Optimierungsflag **-O** gesetzt ist:

```
if __debug__:
    if not <test>:
        raise AssertionError(<"Message">)
```

```
> python -O assertiontest.py
```

```
with <expr> [as <var>]:
    <block>
```

`with` ruft einen Block im *Kontext* von <expr> auf

Verwendung: Als Alternative zu `try/finally` bei Objekten, die ihren eigenen *Context Manager* zur Behandlung von Entry / Exit Actions verfügen.

Der Context Manager besteht aus den speziellen Klassenmethoden `__enter__` sowie `__exit__`:

```
context = <expr>
var = context.__enter__()
try:
    <block>
finally:
    context.__exit__(exc_type, exc_value, traceback)
```

open öffnet ein File. Der Context Manager von open sorgt dafür, daß beim Beenden des **with**-blocks das File auf jeden Fall ordnungsgemäß geschlossen wird, auch wenn während der Verarbeitung eine Exception auftritt:

```
with open("apache.log") as filehandle:
    for line in filehandle:
        <do something with line ...>
```

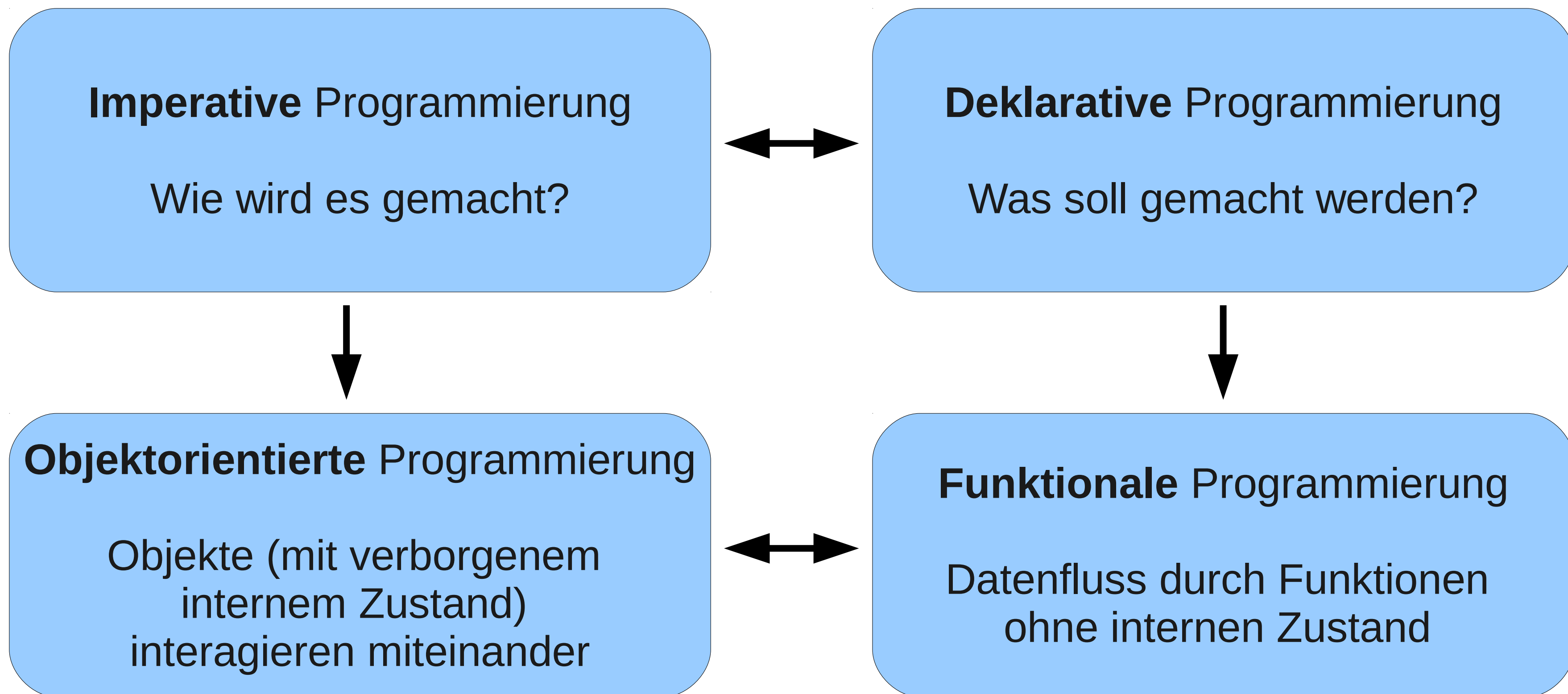
Verwendung von **with** im Multithreading; der Context Manager von **threading.lock** sorgt dafür, daß ein Lock vorhanden ist, und gibt den Lock nach dem **with**-block wieder frei:

```
lock = threading.lock()
with lock:
    <access shared memory>
```

```
try:                                # Start block to
   <statements>                     # catch exceptions

except <Exception> [as <value>]: # handler for exception
   <statements>                     # raised in try block

except:                             # handle all remaining
   <statements>                     # exceptions

else:                               # run if no exceptions
   <statements>                     # were raised

finally:                            # run regardless of any
   <statements>                     # raised exceptions

raise <Exception>                   # throw an exception

assert <test> [,"Message"]          # Raise AssertionError
                                    # if <test> fails
with <expr> [as <var>]:             # Call <block> in context
   <block>                          # of <expr>
```

Teil 7

Funktionale Programmierung



Assembler
Fortran
C
C++
Java
Python
Ruby

SQL
Make
XML
LISP
Prolog
Erlang
Haskell

Beispiel für deklarative Programmierung: **Makefiles vs. Shell Script**

```
# Makefile

CC = cc
LD = ld

prog: foo.o bar.o
	$(LD) -o prog foo.o bar.o

foo.o: foo.c
	$(CC) -c foo.c

bar.o: bar.c
	$(CC) -c bar.c
```

```
#!/usr/bin/env sh

if [ ! -f bar.o ] ||
   [ bar.c -nt bar.o ]
then
    cc -c bar.c
fi

if [ ! -f foo.o ] ||
   [ foo.c -nt foo.o ]
then
    cc -c foo.c
fi

if [ ! -f prog ] ||
   [ foo.o -nt prog ] ||
   [ bar.o -nt prog ]
then
    ld -o prog foo.o bar.o
fi
```

- *Funktion* in der Mathematik:
  - Begriff eingeführt 1694 von Leibniz
  - Zuordnung von jedem Element x aus Definitionsmenge D genau einem Element y aus der Zielmenge Z:

$$f: D \rightarrow Z, x \rightarrow y$$

$$f(x) = x^2, x \in \mathbb{R}$$

- *Funktion* in der Informatik:
  - Häufig synonym gebraucht zu *Unterprogramm*
  - Muß nicht werttreu oder nebenwirkungsfrei sein

```
float quadratic_or_cubic(float x) {
    static int num_calls = 0;

    if (++num_calls < 3) return x * x;
    return x * x * x;
}
```

Was ist nun funktionale Programmierung?

Die Gelehrten streiten sich um eine Definition...

- Zentrale Bausteine sind Objekte ähnlich mathematischer Funktionen
- Programme als Datenfluss durch Funktionen ohne internen Zustand
- Programmierung ohne Assignment Operator

Allerdings sind rein funktionale Sprachen sehr gewöhnungsbedürftig:

- Variablen verhalten sich eher wie *mathematische* Variablen.
- `X = X + 1` führt deshalb zu einem Fehler!
- Keine **`for`** Schleifen mehr, keine **`while`** Schleifen, alles rekursiv ...

Warum sollte man so programmieren ... ?

Vorteile der funktionalen Programmierung:

- **Programmbeweisbarkeit**: Leider nur theoretisch
- **Modularität**: Ein funktionales Programm besteht i.d.R. aus kleinen überschaubaren Funktionen, die alle definierte Aufgaben erfüllen
- **Composability**: Leichte Wiederverwendbarkeit und Neu-Kombinierbarkeit der Funktionen
- **Testbarkeit**: Ideal für Unit Testing (kein interner Zustand der Funktionen)
- **Inhärenter Parallelismus**: Da der Rückgabewert von Funktionen keine Seiteneffekte zeigt, ist der genaue Zeitpunkt der Berechnung zweitrangig. Man braucht keine fehlerträchtigen Konstrukte wie Semaphores o.ä.

Nachteile der funktionalen Programmierung:

- **Ungewohnt** / langer Lernprozess
- **Akzeptanz** ("Akademikersprachen" / "Elfenbeinturmsprachen")
- **Hohe Abstraktion** zur zugrundeliegenden maschinellen Implementierung

Funktionale Programmierung in **Python**:

Python ist **keine** dedizierte funktionale Sprache wie z.B. Haskell!

Sie hat aber gewisse stilistische Elemente von funktionalen Sprachen übernommen, und erleichtert manche "funktional-inspirierte" Programmiertechniken.

- Listen und Tupel
- Sequence Assignments
- Lazy Evaluation über Iteratoren / Generatoren
- List / Generator Comprehensions
- Funktionen als First Class Objects
- First Class / Higher Order Functions (map / filter / reduce)
- Anonyme Funktionen (lambda)

Beispiel für funktionale Programmierung: **Quicksort**

Haskell

```
qsort :: Ord a => [a] -> [a]
   qsort []     = []
   qsort (x:xs) = qsort left ++ [x] ++ qsort right
                  where
                      left   = [y | y <- xs, y <= x]
                      right  = [y | y <- xs, y >  x]
```

Python

```
def qsort(list):
   if len(list) == 0: return []
   x, xs = list[0], list[1:]
   left  = [y for y in xs if y <= x]
   right = [y for y in xs if y >  x]
   return qsort(left) + [x] + qsort(right)
```

Lazy Evaluation:

Nur den Teil eines Ausdruckes auswerten, der gerade tatsächlich benötigt wird.
(Gegenteil: Strict Evaluation, Default in den meisten Programmiersprachen)

```
ones :: [Int]
ones = 1 : ones
```

```
Hugs> take 5 ones
[1,1,1,1,1]
```

Vorteile von Lazy Evaluation:

- Performance- und Speichervorteile
- Möglichkeit, unendlich große Strukturen behandeln zu können

Rückblick Workshop I: range vs. xrange in Python 2

`range(start, stop, step)` erzeugt eine *Liste* (strict evaluation)

`xrange(start, stop, step)` erzeugt ein *Iteratorobjekt*, erzeugt Werte nur auf Anforderung (lazy evaluation)

```
for i in range(10000):
   print i,

for i in xrange(10000):
   print i,

print range(5)  # [0, 1, 2, 3, 4]
print xrange(5) # xrange(5)
```

For kann aber auch alles iterieren, was iterierbar ist:

```
for c in "ABCDEF":
   print c,

for t in (1, 4, 6, -1, 8):
   print t,

for key in { 'Name': 'Albert', 'Surname' : 'Einstein' }:
   print key,

for line in open("/var/log/messages"):
   print line
```

Genauso können viele Funktionen mit allen Arten von Iterables umgehen:

```
print sum(range(100))

print max([3,7,-1,4])

print min("FGNDAE")    # 'A'

t = tuple("ABCDEF")    # ('A', 'B', 'C', 'D', 'E', 'F')

liste = list(t)        # ['A', 'B', 'C', 'D', 'E', 'F']

d = dict([("Name","Albert"),("Surname","Einstein")])

if a in liste:
   <do something with a>
```

Wieso klappt das?

Antwort: Alle Iterables in Python implementieren das *Iteratorprotokoll*

```
<iterator> = iter(<sequence>)    # Return an iterator context

<value> = next(<iterator>)       # Return the next element
```

```
I = iter(xrange(5))

print next(I)   # 0
print next(I)   # 1
print next(I)   # 2
print next(I)   # 3
print next(I)   # 4
print next(I)   # Traceback: StopIteration
```

Wieso klappt das?

Antwort: Alle Iterables in Python implementieren das *Iteratorprotokoll*

```
for x in <iterable>:
    <statements>
```

```
_i = iter(<iterable>)
while True:
    try:
        x = next(_i)
    except StopIteration:
        break
    <statements>
```

Eigene Iteratoren über **Operatorenüberladung**:

```
class MyRange:
    def __init__(self, start, end):
        self.count = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.end:
            raise StopIteration
        else:
            self.count += 1
            return self.count - 1
```

```
for i in MyRange(0,10):
    print i,
```

Schreiben eigener Iteratoren geht aber einfacher, nämlich über:

Generatoren (resumeable functions)

```
yield <value>    # Returns next value in a generator function
```

Jede Funktion, die mindestens einmal das Keyword **`yield`** enthält, wird automatisch zu einem *Generator*, der sich nach dem Iteratorprotokoll verhält.

Generatoren (resumeable functions)

```
def my_range(start, end):
    count = start
    while count < end:
        yield count
        count += 1
    return
```

```
L = my_range(0, 5)
print next(L)    # 0
print next(L)    # 1
print next(L)    # 2
print next(L)    # 3
print next(L)    # 4
print next(L)    #  StopIteration
```

```
(1) L = my_range(0, 5)

(2) print next(L)    # 0

    print next(L)    # 1
    print next(L)    # 2
(3) print next(L)    # 3
    print next(L)    # 4

(4) print next(L)
    #  StopIteration
```

```
def my_range(start, end):      (1)
    count = start
    while count < end:
        yield count             (2)
        count += 1              (3)
    return                      (4)
```

1) Der Aufruf eines Generators führt noch keinen Code aus, sondern gibt nur einen Iterator zurück.
2) Der erste Aufruf über **next** führt den Generator bis zur **`yield`**-Anweisung aus. Die **`yield`**-Anweisung gibt das Element des Iterationsschrittes zurück
3) Alle weiteren Aufrufe setzen den Generator bei **`yield`** fort, bis der Programmablauf wieder zu einer **`yield`**-Anweisung gelangt, usw.
4) Aufruf von **`return`** beendet den Generator und wirft **`StopIteration`**.

Die Bibliohek **itertools** aus der Standard Library bringt viele Standard-Iteratoren mit (Auswahl):

`count(start [, step])`	Unendliche Reihe (ab <start>)
`cycle(seq)`	Wiederholt die Elemente von seq unendlich
`repeat(elem [,n])`	Wiederholt elem unendlich oder n-mal
`chain(seqA, seqB, ...)`	Fügt seqA, seqB, ... aneinander
`tee(iter, [n])`	Erzeugt n Kopien eines Iterators (default 2)
`product(seqA, seqB, ...)`	Erzeugt das cartesische Produkt (entspricht geschachtelten for-Schleifen)
`permutations(seq [,r])`	Alle möglichen Permutationen von seq (Länge r)
`combinations(seq, r)`	Alle möglichen Kombinationen von seq (Länge r)

```
import itertools

i = itertools.count(10)              # 10, 11, 12, 13, 14, 15, ...

i = itertools.cycle('ABCD')          # A, B, C, D, A, B, C, D, A, ...

i = itertools.repeat(10, 5)          # 10, 10, 10, 10, 10

i = itertools.chain('ABC', 'DEF') # A, B, C, D, E, F

t = itertools.tee((x for x in range(5)), 2)
i0 = t[0]                            # 0, 1, 2, 3, 4
i1 = t[1]                            # 0, 1, 2, 3, 4


i = itertools.product(range(4), range(3))
# (0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), ...

i = itertools.permutations((1, 2, 3))
# (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)

i = itertools.combinations((1, 2, 3, 4), 3)
# (1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)
```

List / Generator Comprehensions

Sehr häufig folgende Struktur o.ä. zum Initialisieren einer Liste:

```
# Weise L die ersten 5 Quadratzahlen zu

L = []
for x in range(5):
   L.append(x*x)
```

Verkürzte Form: *List Comprehensions* (aus der mathematischen Mengennotation)

```
# Weise L die ersten 5 Quadratzahlen zu

L = [x*x for x in range(5)]
```

List Comprehensions vs. Generator Comprehensions

List Comprehensions erzeugen eine Liste:

```
# Weise L die ersten 5 Quadratzahlen zu

L = [x*x for x in range(5)]
print L # [0, 1, 4, 9, 16]
```

Generator Comprehensions erzeugen einen Iterator:

```
# Generiere die ersten 5 Quadratzahlen

G = (x*x for x in range(5))

print next(G)    # 0
print next(G)    # 1
print next(G)    # 4
print next(G)    # 9
print next(G)    # 16
```

Eine List / Generator Comprehension kann einen **Test** enthalten:

```
L = []
for x in range(10):
    if x % 2 == 0:
        L.append(x*x)
```

```
L = [x*x for x in range(10) if x % 2 == 0]
```

Iterationen über **mehrere Variablen** sind möglich:

```
L = []
for x in range(5):
    for y in range(5):
        L.append(x*y)
```

```
L = [x*y for x in range(5) for y in range(5)]
```

List comprehensions, allgemeine Form:

Prozedurale Form:

```
L = []
for x1 in i1:
    if <condition1>:
        for x2 in i2:
            if <condition2>:
                ...
                for xN in iN:
                    if <conditionN>:
                        L.append(<expression>)
```

Äquivalente List comprehension:

```
[<expression> for <expr1> in <iterable1> [if <condition1>]
              for <expr2> in <iterable2> [if <condition2>]
          ... for <exprN> in <iterableN> [if <conditionN>] ]
```

Problem: Wieviele Bytes wurden von einem Webserver angefordert? (letzte Zeile in einem Apache-Logfile)

```
81.107.39.38 -  ... "GET /ply/ HTTP/1.1" 200 7587
81.107.39.38 -  ... "GET /favicon.ico HTTP/1.1" 404 133
81.107.39.38 -  ... "GET /ply/bookplug.gif HTTP/1.1" 200 23903
81.107.39.38 -  ... "GET /ply/ply.html HTTP/1.1" 200 97238
81.107.39.38 -  ... "GET /ply/example.html HTTP/1.1" 200 2359
66.249.72.134 - ... "GET /index.html HTTP/1.1" 200 4447
```

Außerdem: Die Logfiles können sehr groß sein ...

Beispiel aus: "Generator Tricks for System Programmers V2.0"

http://www.dabeaz.com/generators-uk/index.html

Mit freundlicher Genehmigung von David Beazley, (c) 2008 David M. Beazley

Wir analysieren das Problem ...

- Eine einzelne Zeile sieht z.B. so aus:

```
81.107.39.38 -   ... "GET /ply/ply.html HTTP/1.1" 200 97238
```

- Die Zahl an Bytes ist im letzten Feld:

```
bytestr = line.rsplit(None,1)[1]
```

- Allerdings kann stattdessen auch ein `'-'` stehen, falls der Wert fehlt:

```
81.107.39.38 -   ... "GET /ply/ HTTP/1.1" 304 -
```

- Falls eine Angabe in Bytes vorhanden, konvertiere nach int:

```
if bytestr != '-':
    bytes = int(bytestr)
```

Klassisch prozedural:

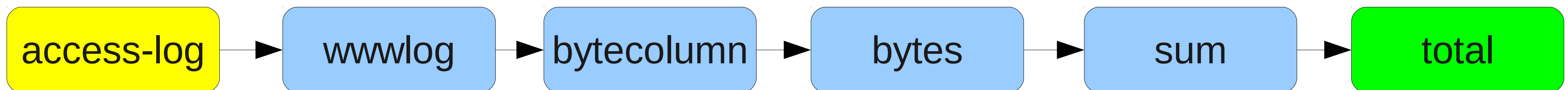
```
wwwlog = open("access-log")

total = 0
for line in wwwlog:
    bytestr = line.rsplit(None,1)[1]
    if bytestr != '-':
        total += int(bytestr)

print "Total", total
```

Und jetzt funktional:

```
wwwlog     = open("access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes      = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

```
wwwlog     = open("access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes      = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

Funktionale Programmierung als Datenfluss durch kleine, abgeschlossene Funktionen:



- Unix-Philosophie: Pipelines
- Aufteilen des Problems in kleine Tools, die alle isoliert getestet und modifiziert werden können
- Verwenden von Generator-Expressions bewirkt, daß nirgendwo eine große (speicherintensive) Liste generiert wird

Set und *Dictionary comprehensions* (ab Python 3)

2 / 3

Set comprehensions:

```
{ x*x for x in range(5) }

# {0, 1, 4, 9, 16}
```

Dictionary comprehensions:

```
{ x : x*x for x in range(5) }

# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

2 / 3

Vorsicht bei Python 2:

Variablen aus List Comprehensions gehören zum lokalen Scope!

"*Pythons dirty little secret*", geändert in Python 3

```
x = "abc"
a = [x for x in range(5)]
print (a, x)

# [0, 1, 2, 3, 4] 4                  (Python 2.x)
# [0, 1, 2, 3, 4] abc                (Python 3.x)
```

Für manche sehr häufig verwendete List comprehensions gibt es vordefinierte *Sequence processing functions:*

Verwendung nach persönlichem Programmiergeschmack.

```
[func(x) for x in <seq>]          →   map(func, <seq>)

[x for x in <seq> if func(x)]     →   filter(func, <seq>)

val = <ini>
for i in <seq>:                   →   reduce(func, <seq> [,<ini>])
    val = func(val, i)

[(seq1[i], seq2[i]) for i in      →   zip(<seq1>, <seq2>, ...)
range(min(len(seq1),len(seq2))]
```

map wendet eine Funktion auf alle Elemente einer Sequenz an:

```
<seq> = map(<function>, <sequence1> [,<sequence2>, ...])
```

```
import string

s = map(string.upper, ["abc", "def", "ghi"])

print s          # ['ABC', 'DEF', 'GHI']


def add(x,y):
   return x+y

print map(add, [1, 2, 3], [4, 5, 6])   # [5, 7, 9]
```

filter gibt alle Elemente einer Sequenz, die eine Bedingung erfüllen:

```
<seq> = filter(<function>, <sequence>)
```

```
def is_even(x)
    return x % 2 == 0

f1 = filter(is_even, range(5))

print f1  # [0, 2, 4]
```

reduce wendet eine Funktion auf alle Elemente einer Sequenz an (von links nach rechts), um die Sequenz auf einen Wert zu reduzieren:

```
<value> = reduce(<function>, <sequence> [,<initial>])
```

```
def add(x, y):
    return x + y

r1 = reduce(add, range(100))

print r1     # 4950

r2 = reduce(add, range(100), 50)

print r2     # 5000
```

all / any testen ob alle / mindestens ein Element einer Sequenz äquivalent zu *True* sind:

```
<bool> = all(<sequence>)
<bool> = any(<sequence>)
```

```
a1 = [1, 1, 1, 1]
a2 = [1, 1, 1, 0]
a3 = [0, 0, 0, 0]

print all(a1), any(a1)    # True, True
print all(a2), any(a2)    # False, True
print all(a3), any(a3)    # False, False
```

zip gibt eine Sequenz von Tupeln zurück, dessen i-te Elemente den i-ten Elementen der Eingangssequenzen entsprechen:

```
<new seq> = zip(<sequence> [,<sequence2>, ...])
```

```
z1 = zip([1,2,3],[4,5,6])

print z1     # [(1, 4), (2, 5), (3, 6)]
```

In der funktionalen Programmierung braucht man häufig in-place kleine Funktionsausdrücke; aber extra eine eigene Funktion dafür schreiben ist zu umständlich.

Erzeugungsmöglichkeit: **Lambda-Expressions**

```
def small_function(x):
    return x*x

small_function = lambda x: x*x
```

```
def is_even(n):
    return n % 2 == 0

f1 = filter(is_even, range(5))

f2 = filter(lambda x: x % 2 == 0, range(5))
```

Beispiel Lambda-Ausdrücke: Anwendung bei der *sort*-Funktion

```
# Beispiel: Sortieren nach dem letzten Element

def get_last(x):
   return x[-1]

s1 = sorted([(1,4),(4,2),(5,3)], key=get_last)

s2 = sorted([(1,4),(4,2),(5,3)], key=lambda x:x[-1])

print s2     # [(4, 2), (5, 3), (1, 4)]
```

2 / 3

1) Python 3 ist wesentlich konsequenter bei der Verwendung von Iteratoren!
Folgende Methoden und Funktionen liefern jetzt einen Iterator anstatt einer Liste:

- `map, filter, zip`
- `range` (`xrange` entfällt)
- Dictionary-Methoden `keys(), values(), items()`

2) `reduce` ist kein Built-In mehr, sondern ist über **`functools`** verfügbar

```
from functools import reduce     # Python 3

r = reduce(lambda x,y: x+y, range(100))

print r       # 4950
```

Rekursion ist ein zentrales Muster in der funktionalen Programmierung

Leider hat die Standard-Pythonimplementierung (CPython) gewisse Limitierungen bezüglich Rekursion:

- Default ist die maximale Rekursionstiefe auf 1000 beschränkt.
- CPython wendet keine *Tail recursion Optimization* an.

  (Veto von Guido van Rossum, siehe:
  http://neopythonic.blogspot.de/2009/04/tail-recursion-elimination.html)

Was ist Tail Recursion Optimization?

-> Entfernen sog. *endrekursiver Funktionsaufrufe* vom Stack
-> Sehr wichtiges Merkmal von funktionalen Programmiersprachen

```
import inspect

def printstack(st):
   for s in st[::-1]: print (s[3]),
   print

def func2():
   printstack(inspect.stack())
   return

def func1():
   printstack(inspect.stack())
   return func2()

printstack(inspect.stack())
func1()
```

```
<module>
<module> func1
<module> func1 func2   # <- TRO würde func1 entfernen
```

Workarounds gegen die Limitierung der Rekursionstiefe:

- Rekursive Aufrufe in iterative Muster umschreiben
- Andere Python-Implementierung verwenden (*Stackless Python*)
- Mittels `sys.setrecursionlimit(value)` erhöhen (Achtung, evtl. muss auch die Stackgröße erhöht werden, Plattform/Implementierungsabhängig)
- Verwenden eines sog. *Trampolin*-Musters

```
def count1(n):
    print n,
    printstack(inspect.stack())
    if n == 0: return
    else: count1(n-1)

count1(4)
```

```
4 <module> count1
3 <module> count1 count1
2 <module> count1 count1 count1
1 <module> count1 count1 count1 count1
0 <module> count1 count1 count1 count1 count1
```

```
def count2(n):
    print n,
    printstack(inspect.stack())
    if n == 0: return None, 0
    else: return count2, n-1

def count_trampolined(n):
    func = count2
    while True:
        func, n = func(n)
        if func == None: break
    return

count_trampolined(4)
```

```
4 <module> count_trampolined count2
3 <module> count_trampolined count2
2 <module> count_trampolined count2
1 <module> count_trampolined count2
0 <module> count_trampolined count2
```

```
<iterator> = iter(<sequence>)    # Startet einen Iterator
<value> = next(<iterator>)       # Nächster Wert aus Iterator

def <generator>(<params>):           # Generatorfunktion
   yield <value>

[<expr> for x in <iter> if <cond>]  # List comprehension
(<expr> for x in <iter> if <cond>)  # Generator comprehension

map(<func>, <seq>[,<seq>, ...])      # Mapping
filter(<cond>, <seq>)                # Filtering
reduce(<func>, <seq>[, <init>])      # Folding
zip(<seq1>[, <seq2, ...])            # Convolution

lambda <params>: <expr>              # Anonyme Funktion

import itertools                     # Bibliothek Iterables
```

Teil 8

Die Standard Library

- Python liefert von Haus aus eine äußerst umfangreiche Bibliothek zu verschiedensten Themen mit. (Python-Philosophie: "Batteries included")
- `__builtins__` enthält Funktionen und Exceptions, die ohne Import jederzeit verfügbar sind; andere Module der Standard Library sind mit **`import`** verfügbar
- Aktuelle Dokumentation (*"keep this under your pillow"*):

  http://docs.python.org/2/library/index.html
  http://docs.python.org/3/library/index.html
- *„Don't reinvent the wheel“.* Bevor man mit Coden anfängt, zuerst prüfen
  - ob die Standard Library schon eine Lösung enthält
  - ob die Python Package Library *„The cheese shop“* schon eine Lösung enthält

    http://pypi.python.org/pypi
  - ob es im Netz schon eine Lösung gibt

Achtung: Die Standard Library wurde beim Wechsel 2 -> 3
an manchen Stellen überarbeitet! Im Zweifelsfall die Dokumentation studieren!

Das Tool *2to3* konvertiert (meist) geänderte Aufrufe.

1) String services

- `string` - Common String Operations
- `re` - Regular Expressions
- `difflib` - Computing Deltas

2) Data Types

- `datetime` - Basic Date and Time Types
- `calendar` - General Calendar-related Functions
- `array` - Efficient Arrays of Fixed-typed Data
- `copy` - Duplicate Objects
- `pprint` - Pretty Printing of Data Structures

3) Mathematics

- `decimal` - Fixed and Floating-point Math
- `fractions` - Rational Numbers
- `random` - Pseudorandom Number Generators
- `math` - Mathematical Functions
- `itertools` - Iterators for efficient looping

4) File and Directory Access

- `os.path` - Common Pathname Manipulations
- `tempfile` - Temporary Files and Directories routines
- `filecmp` - Compare Files and Directories

5) Data Persistence

- `pickle` - Python Object Serialization
- `shelve` - Python Object Persistence
- `anydbm` - Generic access to DBM-style databases
- `sqlite3` - Access SQLite Databases

6) Data Compression and Archiving

- `zlib` - gzip compatible compression
- `bz2` - bzip2 compatible compression
- `gzip` - gzip File support
- `tarfile` - Access tar file archives

7) File formats

- `csv` - CSV file access
- `ConfigParser` - Configuration file parser
- `robotparser` - robots.txt parser

8) Cryptographic services

- `hashlib` - Hashes and message digests
- `hmac` - Keyed-Hashing for message authentication

9) Generic Operation System Services

- `os` - Misc. operating system interfaces
- `time` - Time Access and conversions
- `argparse` - Command line arguments and options parser
- `logging` - Logging facility
- `getpass` - Password input
- `curses` - Terminal character-cell displays
- `platform` - Underlying platform identification
- `ctypes` - Foreign function libraries interface

10) Optional Operating System Services

- `threading` - High-level threading interface
- `mmap` - Memory-mapped file support

11) Interprocess Communication and Networking

- `subprocess` - Subprocess management
- `socket` - Low-level networking interface

12) Internet Data Handling

- `email` - email / MIME handling package
- `json` - JSON encoder / decoder
- `base64` - base16 / 32 / 64 Data Encoding
- `uu` - UU encoding / decoding

13) Structured Markup Processing

- `HTMLparser` - HTML / XHTML Parser
- `sgmllib` - SGML Parser
- `xml.` - XML parsing package

14) Internet Protocols and Support

- `cgi` - Common Gateway Interface support
- `urllib/urllib2` - Libraries for URL opening
- `httplib` - HTTP protocol
- `ftplib` - FTP protocol
- `poplib` - POP3 protocol
- `imaplib` - IMAP4 protocol
- `smtplib` - SMTP protocol
- `SimpleHTTPServer` - Simple HTTP server
- `cookielib` - Cookie handling

15) Multimedia

- `audioop` - Raw audio data handling
- `imageop` - Raw image data handling
- `wave` - Read/write WAV files
- `colorsys` - Conversions between color systems
- `sndhdr` - Determine sound file type
- `imghdr` - Determine image file type

16) Internationalization

- `locale` - Internationalization services

17) Development Tools

- `pydoc` - Documentation generator
- `unittest` - Unit Testing Framework
  `2to3` - Python 2 to Python 3 automated code translation

18) Debugging / Profiling

- `pdb` - The Python Debugger
- `timeit` - Code execution timing measurement
- `trace` - Trace statement execution

19) Python Runtime Services

- `sys` - System specific parameters and functions
- `__builtins__` - Built-In functions
- `__future__` - Statement definition from future Python versions
- `warnings` - Warnings control
- `abc` - Abstract Base Classes
- `atexit` - Exit handlers
- `gc` - Garbage collector interface
- `inspect` - Inspect live objects

20) Python Language Services

- `keyword` - Testing for Python Keywords
- `py_compile` - Compile Python source files
- `dis` - Python Bytecode Disassembler

`tabnanny` - Ambiguous indentation detection

21) Unix specific services

- `posix` - POSIX system calls
- `pwd` - Password database
- `spwd` - Shadow Password database
- `grp` - Group database
- `tty` - Terminal control functions
- `pipes` - Shell pipes interface
- `syslog` - System logs interface
- `commands` - Execute shell commands

Und dann gibts noch spezielle Module für:

MS Windows, Mac OS X, SGI IRIX, Sun OS ...

open: Standard method for file access

```
open(<file name> [, <mode> [, <buffering>]] )
```

```
f = open("hello.txt")
for line in f:
    print line
f.close()


with open("hello.txt") as f:
    for line in f:
        print line
```

math: Mathematical functions

```
import math

print math.cos(math.pi / 4.0)    # 0.70710678118654757
print math.log(1024, 2)          # 10.0
```

random: (Pseudo)random numbers

```
import random

print random.random()        # random float (0.0 - 1.0)
print random.randrange(6)    # random int (0 - 5)

print random.choice([’apple’, ’pear’, ’banana’])
print random.sample(xrange(10), 5)

a = [1, 2, 3, 4]
random.shuffle(a)      # Randomly shuffle list in-place
print a
```

copy: Shallow / Deep Copy von Referenzen

```
import copy

a = [1, 2, [3, 4] ]
b = a
c = copy.copy(a)            # Shallow Copy
d = copy.deepcopy(a)        # Deep Copy

a[1] = 5
a[2][0] = 6

print a             # [1, 5, [6, 4] ]
print b             # [1, 5, [6, 4] ]
print c             # [1, 2, [6, 4] ]
print d             # [1, 2, [3, 4] ]

print a is b        # True
print a is c        # False
print a is d        # False
```

sqlite3: Access sqlite3 databases

```
import sqlite3

conn = sqlite3.connect('zoo.db')
c = conn.cursor()
c.execute("CREATE TABLE animals (name text, species text, age int)")
c.execute("INSERT INTO animals VALUES ('Lizzie', 'Katze', 7)")
c.execute("INSERT INTO animals VALUES ('Tiger', 'Katze', 8)")
c.execute("INSERT INTO animals VALUES ('Snoopy', 'Hund', 6)")
conn.commit()

c.execute("SELECT * FROM animals WHERE species = 'Katze'")
while True:
   entry = c.fetchone()
   if not entry: break
   print entry

c.close()
conn.close()
```

hashlib / hmac: Cryptographical hashing

```
import hashlib

# Available: md5, sha1, sha224, sha256, sha384, sha512

h = hashlib.md5()
h.update("Python ist super!")
print h.hexdigest()    # '44d82a88f03c76caea2cd63f12c3b762'
```

```
import hmac

d = hmac.new("Super Secret Key")
d.update("The quick brown fox jumps over the lazy dog")
print d.hexdigest()    # '7c79c194ad10e062062c99ad50fda5c2'

d = hmac.new("Another totally secret key")
d.update("The quick brown fox jumps over the lazy dog")
print d.hexdigest()    # 'e2ac274d222560fb67566c4feaf49c58'
```

unittest: Unit testing framework (Pythons version of *Junit*)

```
import unittest

def adder(a, b):
    if a == 0: return 0
    else: return a + b

def multiplier(a, b):
    return a * b

class IntegerArithmeticTestCase(unittest.TestCase):
    def testAdd(self):
        self.assertEquals(adder(1, 2), 3)
        self.assertEquals(adder(0, 1), 1)
    def testMult(self):
        self.assertEquals(multiplier(2, 3), 6)
        self.assertEquals(multiplier(0, 7), 0)

if __name__ == '__main__':
    unittest.main()
```

ctypes: Zugriff auf C-Libraries

```
import ctypes

libc = ctypes.CDLL("libc.so.6")

a = libc.printf("Hello, World!\n")  # "Hello, World!"
print a                             # 14

print libc.time(None)        # Zugriff auf time()

print libc.time()
# Segmentation fault
```

SimpleHTTPServer:

Implementierung eines einfachen Webservers unter Port 8000:

```
> cd /srv/www
> python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
localhost - - [15/Feb/2013 15:21:36] "GET / HTTP/1.1" 200 -
...
```

Zugriff auf `/srv/www` ist dann möglich unter:

`http://localhost:8000`