Sage 5: Lineare Algebra und Generatoren

Inhaltsverzeichnis

Lineare Algebra: Matrizen und lineare Gleichungen.

Matrizen können als Listen von Zeilen eingegeben werden.

```
M = matrix ( [[1,2], [3,4]])
M
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

alternativ als Liste mit Formatanweisung

```
matrix(2,2 ,[ 1,2,3,4])
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Matrizen können auch formale Einträge (oder Elementen anderer Ringe) haben, allerdings müssen alle Einträge aus dem gleichen Ring kommen.

```
A = matrix (2,2,[1,x,0,2])
A
```

$$\begin{pmatrix} 1 & x \\ 0 & 2 \end{pmatrix}$$

NB: sage beginnt bei 0 zu zählen!

```
A[0,0]
```

$$1$$

```
A[0,1]
```

x

```
parent(A)
```

$\mathrm{Mat}_{2\times 2}(\mathrm{SR})$

Strukturierte Matrizen können sehr einfach mit List Comprehensions erzeugt werden.

```
var('y')
matrix( [[x^i*y^j for j in range(5)] for i in range(5)] )
```

$$\begin{pmatrix} 1 & y & y^2 & y^3 & y^4 \\ x & xy & xy^2 & xy^3 & xy^4 \\ x^2 & x^2y & x^2y^2 & x^2y^3 & x^2y^4 \\ x^3 & x^3y & x^3y^2 & x^3y^3 & x^3y^4 \\ x^4 & x^4y & x^4y^2 & x^4y^3 & x^4y^4 \end{pmatrix}$$

oder mittels einer Funktion

```
n = 6
H  = matrix (n,n, lambda i,j: 1/(i+j+1))
H
```

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \end{pmatrix}$$

Die üblichen Matrixoperationen

```
A+M
```

$$\begin{pmatrix} 2 & x+2 \\ 3 & 6 \end{pmatrix}$$

```
A*M
```

$$\begin{pmatrix} 3x+1 & 4x+2 \\ 6 & 8 \end{pmatrix}$$

Ein paar wichtige Matrizen.

```
identity_matrix(5)
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```
zero_matrix(2,5)
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Die in der Numerik wichtigen dünnbesetzten (engl. *sparse*) Matrizen kann man platzsparend mit dictionaries definieren.

```
B = matrix (30, 30, {(0,0):1, (3,4):1})
B
```

$$\begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

```
parent(B)
```

$\mathrm{Mat}_{30\times 30}(\mathbf{Z})$

```
B[0,0]
```

1

```
B[0,1]
```

0

Intern werden Matrizen als Listen von Zeilen gespeichert.

```
A[0]
```

$(1,\ x)$

```
A.row(0)
```

$(1,\ x)$

```
A.column(0)
```

$(1,\ 0)$

Eine Matrix kann nachträglich geändert werden.

```
M[1,1]=42
M
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 42 \end{pmatrix}$$

Zuweisung fremder Elemente ist nicht möglich.

```
M[0,1] = x
```

```
Traceback (click to the left of this block for traceback)
...
TypeError: unable to convert x to an integer
```

Dazu muß M zuerst in den höheren Bereich hochgezogen werden.

```
M1 = matrix(SR, M)
M1
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 42 \end{pmatrix}$$

```
M1[0,1] = x
M1
```

$$\begin{pmatrix} 1 & x \\ 3 & 42 \end{pmatrix}$$

Matlab-Notation

```
A[:,1]
```

$$\begin{pmatrix} x \\ 2 \end{pmatrix}$$

```
A[1,:]
```

$$\begin{pmatrix} 0 & 2 \end{pmatrix}$$

Noch einmal die Hilbertmatrix.

```
n = 6
H  = matrix (n,n, lambda i,j: 1/(i+j+1))
H
```

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} \\ \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} & \frac{1}{10} & \frac{1}{11} \end{pmatrix}$$

```
parent(H)
```

$$\mathrm{Mat}_{6\times 6}(\mathbf{Q})$$

Wir lösen eine lineare Gleichung mit rechter Seite b. Zunächst als einspaltige Matrix.

```
b = matrix(n,1, lambda i,j:1/(i+1)^2)
b
```

$$\begin{pmatrix} 1 \\ \frac{1}{4} \\ \frac{1}{9} \\ \frac{1}{16} \\ \frac{1}{25} \\ \frac{1}{36} \end{pmatrix}$$

Mehrere Möglichkeiten.

```
H^-1*b
```

$$\begin{pmatrix} \frac{71}{15} \\ -35 \\ 140 \\ -280 \\ \frac{525}{2} \\ -\frac{462}{5} \end{pmatrix}$$

```
H\b
```

$$\begin{pmatrix} \frac{71}{15} \\ -35 \\ 140 \\ -280 \\ \frac{525}{2} \\ -\frac{462}{5} \end{pmatrix}$$

```
H.inverse()*b
```

$$\begin{pmatrix} \frac{71}{15} \\ -35 \\ 140 \\ -280 \\ \frac{525}{2} \\ -\frac{462}{5} \end{pmatrix}$$

```
H.solve_right(b)
```

$$\begin{pmatrix} \frac{71}{15} \\ -35 \\ 140 \\ -280 \\ \frac{525}{2} \\ -\frac{462}{5} \end{pmatrix}$$

Oder als Vektor.

```
b = vector ( [ 1/i^2 for i in [1..n] ])
b
```

$$\left(1, \frac{1}{4}, \frac{1}{9}, \frac{1}{16}, \frac{1}{25}, \frac{1}{36}\right)$$

```
H^-1*b
```

$$\left(\frac{71}{15}, -35, 140, -280, \frac{525}{2}, -\frac{462}{5}\right)$$

```
H.solve_right(b)
```

$$\left(\frac{71}{15}, -35, 140, -280, \frac{525}{2}, -\frac{462}{5}\right)$$

```
H\b
```

$$\left(\frac{71}{15}, -35, 140, -280, \frac{525}{2}, -\frac{462}{5}\right)$$

Vorsicht bei singulären Matrizen.

```
A = matrix( [[1,2,3], [2,4,5], [0,0,1]] )
A
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 0 & 0 & 1 \end{pmatrix}$$

```
A.rank()
```

$$2$$

Es werden nicht alle Lösungen geliefert!

```
A \ vector([0,0,0])
```

$$(0, 0, 0)$$

Um alle Lösungen zu erhalten, muß man den Kern der Matrix kennen.

```
A.right_kernel()
```

$$\mathrm{RowSpan}_{\mathbf{Z}}\begin{pmatrix} 2 & -1 & 0 \end{pmatrix}$$

Das ist ein Vektorraum. (In diesem Fall nur ein Modul, weil die ganzen Zahlen keinen Körper bilden).

Lineare Algebra: Vektorräume

```
V = QQ^5
V
```

$\mathbf{Q}^5$

Erzeugen von Elementen.

```
v1 = V([1,1,1,0,0])
v2 = V([1,-1,1,0,0])
v3 = V([1,0,1,0,0])
```

```
v1
```

(1, 1, 1, 0, 0)

```
parent(v1)
```

$\mathbf{Q}^5$

Der von den Vektoren erzeugte Unterraum. Es wird sofort eine Basis gebildet.

```
U = V.subspace([v1,v2,v3])
U
```

$\mathrm{RowSpan}_{\mathbf{Q}} \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$

```
U.basis()
```

[(1, 0, 1, 0, 0), (0, 1, 0, 0, 0)]

Feststellen, ob ein Vektor im Unterraum liegt.

```
v4 = V([0,0,0,1,0])
v4 in U
```

False

```
v1 in U
```

True

Bestimmung der Koordinaten bezüglich der Basis von U.

```
U.coordinates(v1)
```

$[1, 1]$

Manipulation von Unterräumen (Summe, Durchschnitt, Vergleich).

```
W = V.subspace([V(v4)])
W
```

$\mathrm{RowSpan}_{\mathbf{Q}}\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \end{pmatrix}$

```
U <= W
```

```
__main__:1: DeprecationWarning: The default order on free mod
has changed. The old ordering is in
sage.modules.free_module.EchelonMatrixKey
See http://trac.sagemath.org/23878 for details.
```

False

```
U+W
```

$\mathrm{RowSpan}_{\mathbf{Q}}\begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$

```
U <= U+W
```

True

```
U.intersection(W)
```

$\mathrm{RowSpan}_{\mathbf{Q}}()$

Wir haben oben den Kern einer Matrix betrachtet.

```
A.right_kernel()
```

$\mathrm{RowSpan}_{\mathbf{Z}}\begin{pmatrix} 2 & -1 & 0 \end{pmatrix}$

Allerdings müssen wir die Matrix zuerst in einen Körper verpflanzen.

```
A1 = matrix(QQ, A)
parent(A1)
```

$\mathrm{Mat}_{3\times 3}(\mathbf{Q})$

```
A1.right_kernel()
```

$$\mathrm{RowSpan}_{\mathbf{Q}}\begin{pmatrix} 1 & -\frac{1}{2} & 0 \end{pmatrix}$$

Der Bildraum.

```
A1.image()
```

$$\mathrm{RowSpan}_{\mathbf{Q}}\begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

In diesem Fall sind Kern und Bildraum komplementär.

```
A1.right_kernel() + A1.image()
```

$$\mathrm{RowSpan}_{\mathbf{Q}}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Ein Anwendungsbeispiel

Als Anwendungsbeispiel untersuchen wir die lineare Unabhängigkeit von Polynomen. Dazu müssen wir sie erst in entsprechende Koordinatenvektoren umwandeln.

```
Qx = QQ[x]
x = Qx(x)
```

```
pp =[1+x+x^2, 1-x+x^2, x^3-x, 2+x^3]
pp
```

$$\left[x^2 + x + 1, x^2 - x + 1, x^3 - x, x^3 + 2\right]$$

Dazu werden am besten die Koeffizientenvektoren extrahiert.

```
p0=pp[0]
p0
```

$$x^2 + x + 1$$

Die TAB-Taste zeigt zwei mögliche Kandidaten.

```
p0.coefficients()
```

$$[1, 1, 1]$$

`.coefficients()` zeigt nur die nichtverschwindenden Koeffizienten.

```
[p.coefficients() for p in pp]
```

$$[[1, 1, 1], [1, -1, 1], [-1, 1], [2, 1]]$$

`.coeffs()` füllt mit 0 auf, allerdings sind die Dimensionen trotzdem verschieden.

```
kv0=[p.coefficients(sparse=False) for p in pp]
kv0
```

$$[[1, 1, 1], [1, -1, 1], [0, -1, 0, 1], [2, 0, 0, 1]]$$

wir müssen die Vektoren explizit mit 0 auffüllen.

```
n=max([p.degree() for p in pp])+1
n
```

$$4$$

```
def verlaengern(v,n):
    for i in range(len(v),n):
        v.append(0)
    return v
```

```
kv = [verlaengern(v,n) for v in kv0]
kv
```

$$[[1, 1, 1, 0], [1, -1, 1, 0], [0, -1, 0, 1], [2, 0, 0, 1]]$$

Der `.append`-Befehl verändert die Listen direkt.

```
kv0
```

$$[[1, 1, 1, 0], [1, -1, 1, 0], [0, -1, 0, 1], [2, 0, 0, 1]]$$

Die Dimension der linearen Hülle der Polynome ist jetzt gleich dem Rang der Koeffizientenmatrix.

```
rank (matrix(kv0))
```

$$4$$

Das ganze als Funktion verpackt:

```
def linearunabhaengig(pp):
    n=max([p.degree() for p in pp])+1
```

```
    kv0=[p.coefficients(sparse=False) for p in pp]
    kv = [verlaengern(v,n) for v in kv0]
    r = rank (matrix(kv0))
    return (r == len(pp))
```

```
linearunabhaengig(pp)
```

True

Es geht auch mit Unterräumen.

```
Qx = QQ[x]
x=Qx(x)
```

```
pp
```

$[x^2 + x + 1, x^2 - x + 1, x^3 - x, x^3 + 2]$

```
kv0 = [p.coefficients(sparse=False) for p in pp]
kv0
```

$[[1, 1, 1], [1, -1, 1], [0, -1, 0, 1], [2, 0, 0, 1]]$

```
kv = [verlaengern(v,n) for v in kv0]
kv
```

$[[1, 1, 1, 0], [1, -1, 1, 0], [0, -1, 0, 1], [2, 0, 0, 1]]$

```
V = QQ^n
V
```

$\mathbf{Q}^4$

```
U = V.subspace(kv)
U
```

$$\mathrm{RowSpan}_{\mathbf{Q}} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
U.dimension()
```

4

als Funktion verpackt.

```
def linearunabhaengig2(pp):
    kv0 = [p.coeffs() for p in pp]
    n=max([p.degree() for p in pp])+1
    V=QQ^n
    U = V.subspace( [verlaengern(v, n) for v in kv0] )
    return(U.dimension() == len(pp))
```

```
linearunabhaengig2(pp)
```

```
__main__:1: DeprecationWarning: The use of coeffs() is now
deprecated in favor of coefficients(sparse=False).
See http://trac.sagemath.org/17518 for details.
```

True

Generatoren

Ein *Generator* ist ein Speziallfall eines Iterators. Wichtige Anwendungen sind: 'lazy evaluation', Schleifen, elegantes Aufsammeln von Zwischenergebnissen und insbesondere rekursive Erzeugung mathematischer Objekte.

'lazy evaluation'

Praktisch gesehen ist ein Generator eine Art Liste, deren Elemente spontan bei Bedarf erzeugt werden und dann verfallen. Einfache Generatoren kann man ähnlich wie Listen erzeugen, mit runden anstatt eckigen Klammern.

```
g = (1..4)
g
```

```
<generator object at 0x7fa787ba7e90>
```

Der Generator wird duch die Methode **.next()** in Bewegung versetzt. Er gibt daraufhin einen Wert aus und bleibt bis zum nächsten Aufruf von **.next()** stehen (mehr dazu unten). Von außen betrachtet verhält er sich wie ein Kaugummiautomat, wobei **.next()** dem Einwurf einer Münze entspricht.

```
g.next()
```

1

```
g.next()
```

2

```
g.next()
```

3

```
g.next()
```

```
4
```

Wenn alle Werte ausgegeben sind, bricht der Generator ab.

```
g.next()
```

```
Traceback (click to the left of this block for traceback)
...
StopIteration
```

So wie ein Kaugummi nicht von alleine in den Automaten zurückkehrt, sondern letzterer neu befüllt werden muß, kann ein Generator kann nur einmal von vorne nach hinten durchlaufen werden und nicht in einen vorhergehenden Zustand zurückversetzt werden. Dafür muß ein neuer Generator initialisiert werden.

```
g = (1..4)
```

```
g.next()
```

```
1
```

Man kann auch alle Kaugummis auf einen Schlag herausholen durch direke Umwandlung in eine Liste:

```
list(g)
```

```
[2, 3, 4]
```

```
g.next()
```

```
Traceback (click to the left of this block for traceback)
...
StopIteration
```

Schleifen

Generatoren spielen eine wichtige Rolle in Schleifen, insbesondere, wenn dabei sonst eine lange Liste erzeugt werden müßte.

```
g=(1..4)
for i in g:
    print i
```

```
1
2
3
4
```

Am Ende der Schleife ist der Generator leer.

```
g.next()
```

```
Traceback (click to the left of this block for traceback)
...
StopIteration
```

Ein Generator kann auch unendlich groß sein. In diesem Fall ist es nicht ratsam, ihn in eine Liste umzuwandeln, weil sage nicht von vornherein erkennen kann, ob ein Generator nach endlich vielen Schritten abbricht oder nicht. (Dieses Frage ist nach A.Turing algorithmisch nicht entscheidbar).

```
nn = (1..)
```

```
nn.next()
```

```
1
```

```
nn.next()
```

```
2
```

```
for i in nn:
    print i
    if i > 10:
        break
```

```
3
4
5
6
7
8
9
10
11
```

```
nn.next()
```

```
12
```

Auch die rationalen Zahlen sind abzählbar.

```
for q in QQ:
    print q
```

```
WARNING: Output truncated!
full_output.txt
```

```
0
1
-1
1/2
-1/2
2
-2
1/3
-1/3
3
-3
2/3
-2/3
3/2
-3/2
1/4
-1/4
4
-4


...

541/355
-541/355
356/541
-356/541
Traceback (click to the left of this block for traceback)
...
__SAGE__
full_output.txt
```

Neue Iteratoren aus vorhandenen können mit *List Comprehension* erzeugt werden.

```
g=(1..4)
```

```
g2 = (i^2 for i in g)
g
```

```
<generator object at 0x7fa787dca050>
```

dabei läuft im Hintergrund der ursprüngliche Generator mit:

```
g.next()
```

```
1
```

```
g2.next()
```

```
4
```

```
g.next()
```

```
3
```

```
g2.next()
```

```
16
```

```
g.next()
```

```
Traceback (click to the left of this block for traceback)
...
StopIteration
```

```
g2.next()
```

```
Traceback (click to the left of this block for traceback)
...
StopIteration
```

ansonsten läßt sich ein Generator wie eine Liste behandeln.

```
reduce(operator.add, (1..5))
```

```
15
```

Generatoren Programmieren

Ein Generator kann wie eine Funktion programmiert werden, indem anstelle von **return** der Befehl **yield** gesetzt wird. Letzterer hat zwei Funktionen:

- Anhalten der Funktion
- Zurückgeben eines Werts.

Dabei wird der aktuelle Zustand festgehalten und beim nächsten Aufruf von `.next()` an dieser Stelle fortgesetzt bis zum nächsten `yield` oder bis zum Ende des Codes.

```
def abc():
    yield 'a'
    yield 'b'
    yield 'c'
```

Dabei ist zu beachten, daß **abc()** nicht der Generator an sich ist, sondern eine Funktion, die bei jedem Aufruf neuen Generator erzeugt und initialisiert.

```
g = abc()
```

```
g
```

```
<generator object abc at 0x7fa787bb9aa0>
```

Der Aufruf von **.next()** bewirkt, daß der Generator gestartet wird und bis zum ersten **yield** läuft. Der entsprechende Wert wird zurückgegeben und der Generator hält an.

```
g.next()
```

```
a
```

Beim zweiten Aufruf macht er an der gleichen Stelle weiter, bis er auf das nächste **yield** trifft.

```
g.next()
```

```
b
```

usw.

```
g.next()
```

```
c
```

Bis das Ende des Codes erreicht wird.

```
g.next()
```

```
Traceback (click to the left of this block for traceback)
...
StopIteration
```

Will man einen Generator unter gewissen Bedingungen vorzeitig abbrechen, wird an der entsprechenden Stelle ein **return** eingefügt.

```
def abc(cnicht):
    yield 'a'
    yield 'b'
    if cnicht:
        return
    yield 'c'
```

Ist die Bedingung erfüllt, bricht der Generator an der entsprechenden Stelle ab.

```
g = abc(True)
```

```
g.next()
```

```
a
```

```
g.next()
```

```
b
```

```
g.next()
```

```
Traceback (click to the left of this block for traceback)
...
StopIteration
```

Wenn nicht, wird **return** übersprungen.

```
g = abc(False)
```

```
g.next()
```

```
a
```

```
g.next()
```

```
b
```

```
g.next()
```

```
c
```

```
g.next()
```

```
Traceback (click to the left of this block for traceback)
...
StopIteration
```

Noch einmal kurz

```
[t for t in abc(True)]
```

```
[a, b]
```

```
[t for t in abc(False)]
```

```
[a, b, c]
```

Aufsammeln von Ergebnissen

Generatoren sind auch nützlich, um im Verlauf von Rechnungen Zwischenergebnisse auszuwerfen, ohne dabei explizit eine Liste mitzuführen. Wir betrachten noch einmal das Primzahlzwillingsproblem aus der Übung. **p** durchläuft alle Primzahlen kleiner als die obere

Schranke **n** und jedes Mal, wenn ein Zwilling auftaucht, wird das Paar ausgeworfen.

```
def primzw(n=oo):
    p=3
    while p<n:
        q = next_prime(p)
        if q == p+2:
            yield [p,q]
        p=q
```

Der Vorteil ist Übersichtlichkeit und geringstmöglicher Speicherbedarf (allerdings auf Kosten der Laufzeit).

```
list(primzw(100))
```

$[[3, 5], [5, 7], [11, 13], [17, 19], [29, 31], [41, 43], [59, 61], [71, 73]]$

Dabei kann im Prinzip die Anzahl der Zwillinge offen gelassen werden:

```
pp = primzw(oo)
```

Um zum Beispiel die ersten 10 Paare zu erzeugen, schreiben wir einen zweiten Generator um den ersten herum, der die Paare mitzählt:

```
def firstn(n,g):
    for i in range(n):
        yield g.next()
```

Und wir bekommen das gewünschte Resultat ohne darüber nachdenken zu müssen, wie groß wir die obere Grenze wählen müssen.

```
list(firstn(10,pp))
```

$[[3, 5], [5, 7], [11, 13], [17, 19], [29, 31], [41, 43], [59, 61], [71, 73], [101, 103], [107, 109]]$

Rekursive Generatoren

Besonders elegant werden Generatoren, wenn sie rekursiv angewandt werden. Als Beispiel erzeugen wir rekursiv alle Elemente der kartesischen Potenz einer Menge. Die kartesische Potenz kann induktiv definiert werden, was sich unmittelbar in einen Generator übersetzen läßt:

```
def cart(S,n):
    if n == 1:
        for s in S:
```

```
            yield [s]
    else:
        for s in S:
            for p in cart(S,n-1):
                yield [s]+p
```

```
list(cart([1,2],4))
```

[[1, 1, 1, 1], [1, 1, 1, 2], [1, 1, 2, 1], [1, 1, 2, 2], [1, 2, 1, 1], [1, 2, 1, 2], [1, 2, 2, 1], [1, 2, 2, 2], [2, 1, 1, 1], [2, 1,

Ganz analog kann man alle Permutationen der Elemente einer Liste erzeugen, indem jeweils ein Element an die Spitze gestellt und der Rest permutiert wird.

```
def perm(S):
    n = len(S)
    if n==0: yield []
    else:
        for i in range(n):
            for p in perm(S[:i]+S[i+1:]):
                yield [S[i]] + p
```

```
list(perm([1,2,3]))
```

[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

Einfacher zu lesen und zu Programmieren wird es mit **Set**:

```
def permset(S):
    S1=Set(S)
    n = len(S1)
    if n == 0:
        yield []
    else:
        for s in S1:
            for p in perm(S1.difference([s])):
                yield [s]+p
```

```
list(permset([1,2,3]))
```

[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

```

```