Contents

1 Sage als Taschenrechner

Man kann Sage einfach als **sehr** mächtigen Taschenrechner verwenden. Es werden dabei Zahlen mit beliebiger Genauigkeit unterstützt, ebenso Brüche und komplexe Zahlen. Für viele Berechnungen werden exakte Ergebnisse geliefert.
Um eine Rechnung auszuführen, geben wir die Rechnung in eine Zelle ein und drücken dann **Umschalt + Enter** um die Berechnung zu starten.

Sage code
```
1 + 1
```

```
2
```

Sage code
```
(2 * 2 + 5)/3
```

```
3
```

Exponentation

Sage code
```
2^10
```

```
1024
```

Langzahlarithmetik

Sage code

```
2^1000
```

```
107150860718626732094842504906000181056140481170553360744375038837035105112493612249319837\
881569585812759467291755314682518714528569231404359845775746985748039345677748242309854 2\
107460506237114187795418215304647498358194126739876755916554394607706291457119647768654 21\
676604298316526243868372056680693 76
```

Bruchrechnen

Sage code

```
1/6 + 7/12
```

```
3/4
```

Komplexe Zahlen

Sage code

```
sqrt(-4)
```

```
2*I
```

Sage code

```
I^2
```

```
-1
```

Realteil

Sage code

```
real((1 + I)/(1 - I))
```

```
0
```

Imaginärteil

Sage code

```
imag((1 + I)/(1 - I))
```

```
1
```

Exakte Auswertung mathematischer Funktionen

Sage code

```
sin(pi/2)
```

```
1
```

Sage code

```
sin(pi/3)
```

```
1/2*sqrt(3)
```

Sage code

```
sin(pi/4)
```

```
1/2*sqrt(2)
```

Um Ungeleichungen exakt auszuwerten, müssen wir sie in einen Wahrheitswert umwandeln.

Sage code

```
bool(sqrt(2) < 2 * sin(pi/3))
```

```
True
```

Gleitkommaarithmetik mit beliebiger Genauigkeit

Sage code
```
numerical_approx(pi, digits = 400)
```

```
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348\
253421170679821480865132823066470938446095505822317253594081284811174502841027019385211055\
596446229489549303819644288109756659334461284756482337867831652712019091456485669234603\
486104543266482133936072602491412737245870066063155881748815209209628292540917153643678925\
903600113305305488204665213841469519415116094330572703657595919530921861173819326117931051\
1848072453903
```

Dasselbe in objektorientierter Notation

Sage code
```
(pi + e^2).n(digits = 1000)
```

```
10.530648752520443465693070843854510697377484969926953145062072414830390202365266762012347\
310421238862755901247778761172770978448302331855606203751621526558892350406998309349274808\
079509755704996370169637846984588438743376671107308831095292457066858818372749804549942236\
496297922263889433805673555524487890454058302910103812047950796924085123780149881289704809\
093035647254337983403014542652297161592863442315991833510138597124827441396033036719792839\
928124150602143815127097154269577240182606467901802408317660896453035365228345658929992748\
792918001800937296173333692781053000583057675416277360434289533620709249060138338795181997\
973750309002758643851883702926331707287258181290037567515935461189742948294037381966476377\
551141261429810903623516013132460565864583284206738541402336151367908897279860043864889639\
891429010084124909613735172446736689987681625072748642898120795781539235186609912908539998\
013639314329067156430539000857850175309296293268346866917304865948408910465623063448527800\
727433562524079550079 1
```

Bei längeren Berechnungen ist es sinnvoll sich zuerst einen geeigneten Datentyp zu definieren.

Sage code
```
R = RealField(1000) # Praezision in Bits
```

Sage code
```
a = R(pi + 2 * cos(sqrt(2))) # ist aequivalent zu a = (pi + 2*cos(sqrt(2))).n(prec=1000)
a
```

```
3.4534800431205421853719393410966821674460639001577165347557651277858194485392799158383948\
858603787212764074422124493433029780972406979919670147290414122985134570460403690116304800\
293103154488672423994678507761079241574053184885496625498252454523664605727741827887817590\
952826950183930549966025246479740 7
```

Sage code
```
type(a)
```

```
<type 'sage.rings.real_mpfr.RealNumber'>
```

Sage code
```
parent(a)
```

```
Real Field with 1000 bits of precision
```

Achtung: Kommen Zahlen unterschiedlicher Präzision in einer Rechnung vor, dann hat das Ergebnis immer die kleinste Präzision. Konstanten müssen also manuell zu einer Zahl mit hoher Präzision konvertiert werden.

Sage code
```
parent(0.1)
```

```
Real Field with 53 bits of precision
```

Falsch:

Sage code

```
a1 = 2 * a + 0.1
a1; parent(a1)
```

```
7.00696008624108
Real Field with 53 bits of precision
```

Falsch: Hier wird mit $2a + 0.1$ mit niedriger Präzision berechnet, und danach das Ergebnis zu einer Zahl mit 1000 Bit Präzision konvertiert

Sage code

```
b = R(2 * a + 0.1)
b; parent(b)
```

```
7.006960086241083907054871815489605069160461425781250000000000000000000000000000000000000\
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000\
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000\
000000000000000000000000000000000
Real Field with 1000 bits of precision
```

Richtig:

Sage code

```
c = 2 * a + R(0.1)
c; parent(c)
```

```
7.006960086241084370743878682193364334892127800315433069511530255571638897078559831676789\
771720757442552814884424898686605956194481395983934029458082824597026914092080738023260 96\
586206308977344847989357015522158483148106369770993250996504909047329211455483655775635 19\
905653900367861099932050492959481 5
Real Field with 1000 bits of precision
```

2 Symbolisches Rechnen

Symbolische Variablen müssen deklariert werden.

Sage code

```
var('x, y, z')
```

```
(x, y, z)
```

Sage code

```
expr1 = (x + y + z)^3
expr1
```

```
(x + y + z)^3
```

Sage code

```
parent(expr1)
```

```
Symbolic Ring
```

Anzeige in mathematischer Notation

Sage code

```
view(expr1)
```

$(x+y+z)^3$

Ausmultiplizieren

Sage code

```
view(expand(expr1))
```

$x^3+3\,x^2y+3\,x^2z+3\,xy^2+6\,xyz+3\,xz^2+y^3+3\,y^2z+3\,yz^2+z^3$

Faktorisieren

Sage code

```
expr2 = (x^2 - 2 * x * y + y^2)
view(expr2)
```

$x^2-2\,xy+y^2$

Sage code

```
view(factor(expr2))
```

$(x-y)^2$

Substitution von Variablen und Termen

Sage code

```
expr3 = expr2(y = sin(z^2))
view(expr3)
```

$x^2-2\,x\sin\left(z^2\right)+\sin\left(z^2\right)^2$

Sage code

```
expr3.subs_expr(sin(z^2) == y).show()
```

$x^2-2\,xy+y^2$

Differentialrechnung

Sage code

```
diff(x * sin(x), x).show()
```

$x\cos\left(x\right)+\sin\left(x\right)$

2.1 Polynome

Wir konstruieren einen Datentyp für Polynome in der Variablen t, und Koeffizienten aus den Rationalen Zahlen ($\mathbb{Q}$).

Sage code

```
P.<t> = QQ[]
```

gen() gibt die Variable des Polynoms zurück

Sage code

```
P.gen()
```

```
t
```

Wir definieren uns einige Polynome

Sage code

```
p1 = (1/2 * t - 3/2) * (t - 1/4)
view(p1)
```

$$\frac{1}{2}t^2 - \frac{13}{8}t + \frac{3}{8}$$

roots() berechnet die Nullstellen eines Polynoms zusammen mit ihren Vielfachheiten.

Sage code

```
p1.roots()
```

```
[(3, 1), (1/4, 1)]
```

Probe durch Einsetzen.

Sage code

```
p1(3); p1(1/4)
```

```
0
0
```

degree() berechnet den Grad des Polynoms.

Sage code

```
p1.degree()
```

```
2
```

Sage code

```
p2 = (1 + t - t^2)^3
view(p2)
```

$$-t^6 + 3t^5 - 5t^3 + 3t + 1$$

Sage code

```
parent(p2)
```

```
Univariate Polynomial Ring in t over Rational Field
```

Standardmäßig berechnet roots() nur Nullstellen in der selben Grundmenge in der die Koeffizienten des Polynoms liegen. In unserem Fall $\mathbb{Q}$. Unser Polynom hat keine rationalen Nullstellen.

Sage code

```
p2.roots()
```

```
[]
```

Wir können die Grundmenge aber auch explizit angeben. Hier berechnen wir alle reellen Nullstellen.

Sage code

```
p2.roots(ring = RR)
```

```
[(-0.618031057174341, 1), (1.61804465415111, 1)]
```

Um sämtliche Nullstellen zu bekommen, brauchen wir komplexe Zahlen.

Sage code

```
p2.roots(ring = CC)
```

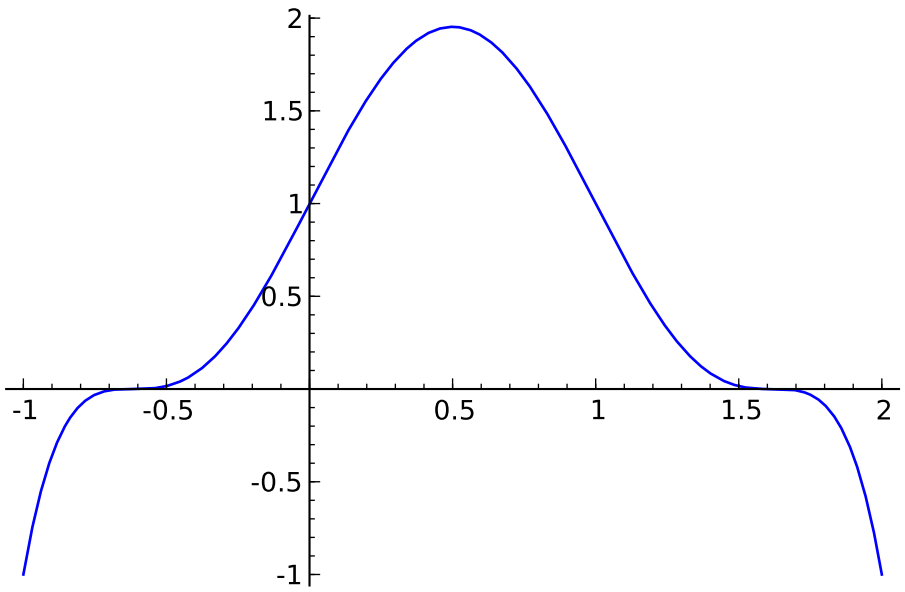
```
[(-0.618031057174341, 1), (1.61804465415111, 1), (-0.618035454537672 - 2.53887260392275e-\
6*I, 1), (-0.618035454537672 + 2.53887260392275e-6*I, 1), (1.61802865604929 - 9.236411382\
58941e-6*I, 1), (1.61802865604929 + 9.23641138258941e-6*I, 1)]
```

3 Plotten von Funktionen

Sage verfügt über vielfältige High-level Funktionen zum Plotten von Funktionen.
Wir plotten das Polynom $p_2 = -t^6 + 3t^5 - 5t^3 + 3t + 1$ im Intervall $[-1, 2]$.

Sage code

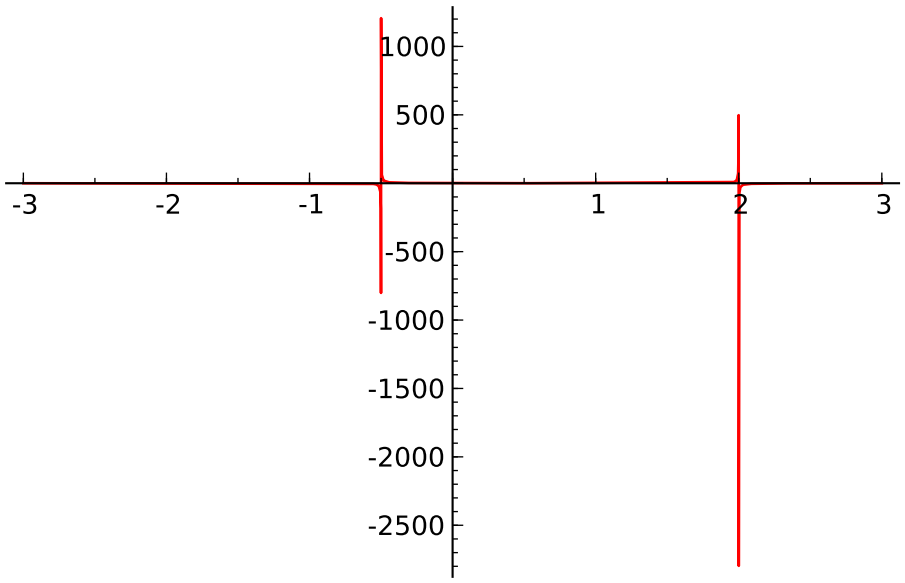
```
p = plot(p2, -1, 2).show(filename="sage0.pdf")
```



Plot einer Funktion mit Polstellen. Es sind leider keine Details sichtbar.

Sage code

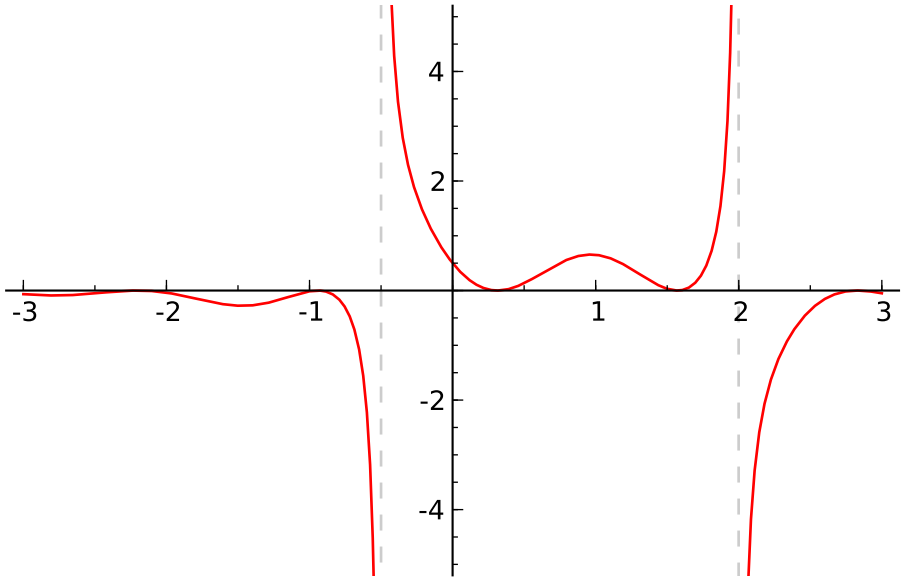
```
f(x) = (sin(5 * x) - 1)/(2 * x^2 - 3 * x - 2)
p = plot(f(x), (x, -3, 3), rgbcolor = 'red')
p.show(filename="sage0.pdf")
```



Für ein besseres Ergebnis müssen wir den den Ausschnitt explizit auswählen.

Sage code

```
plot2 = plot(f(x), (x, -3, 3), rgbcolor = 'red', detect_poles = 'show')
plot2.show(ymin = -5, ymax = 5, filename="sage0.pdf")
```



4 Interaktive Programme

Sage bietet die Möglichkeit kleine interaktive Progamme in das Notebook einzubinden (Siehe die Dokumentation zu *interact*). Weitere Beispiele für *interact* finden Sie im Sage-Wiki.
Das folgende Beispiel visualisiert die Taylorpolynome der Funktion $f(x) = \sin(x) \cdot e^{-x}$.

Sage code

```
var('x')
f(x) = sin(x) * e^(-x)
p = plot(f, -1, 5, thickness = 2)
html('<h2>Taylor Polynome</h2>')

@interact
def _(x0 = input_box(0, label = "x0="),
      order = slider(0, 12, 1, label = "Ordnung: ", default = 3)):

    ft = f.taylor(x, x0, order)
    pt = plot(ft, -1, 5, color = 'green', thickness = 2)
    dot = point((x0, f(x0)), pointsize = 80, rgbcolor = (1, 0, 0))
    html('$f(x) = %s$' % latex(f(x)))
    html('$\hat{f}(x;%s) = %s + \mathcal{O}(x^{%s})$' %
                 (latex(x0), latex(ft(x)), order + 1))
    (dot + p + pt).show(ymin = -.5, ymax = 1)
```

5 Datentypen

Die wichtigsten Datentypen in Sage/Python sind:

- Tupel 5.1

- Listen 5.2
  - Erzeugen von Listen 5.3
  - Kurznotation für arithmetische Progressionen 5.4
  - List Comprehensions 5.5
- Strings 5.6
- Dictionaries 5.7
- Mengen 5.8

5.1 Tupel

Tupel sind ähnlich wie Listen, nur können einmal erstellte Tupel nicht mehr verändert werden (*immutable*). Tupel sind besonders nützlich, um mehrere Werte gleichzeitig von einer Funktion zurückzugeben.

Sage code
```
a = 1, 2, 3
a
```

```
(1, 2, 3)
```

Sage code
```
a[0]; a[1]; a[2]
```

```
1
2
3
```

Die Funktion xgcd(a, b) berechnet den kleinsten gemeinsamen Teiler von a und b mit Hilfe des erweiterten Euclidschen Algrithmus und gibt das Ergebnis als Tupel von drei Zahlen zurück.

Sage code
```
t = xgcd(14, 25)
t
```

```
(1, 9, -5)
```

Man kann die Elemente des Tupels auch direkt einzelnen Variablen zuweisen (Tupel unpacking).

Sage code
```
a1, a2, a3 = xgcd(14, 25)
a2
```

```
9
```

Simultanes vertauschen zweier Variablen

Sage code
```
t1 = 10
t2 = cos(x)
t1; t2
```

```
10
cos(x)
```

Sage code
```
t1, t2 = t2, t1

t1; t2
```

```
cos(x)
10
```

5.2 Listen

Listen sind wahrscheinlich der wichtigste Datentyp in Sage/Python. Der wichtigste Unterschied zu Tupeln ist dass Listen veränderbar (mutable) sind. Sie können Elemente zu Listen hinzufügen, löschen, verändern. Das ist bei Tupeln nicht möglich.
Einige Operationen auf Listen:

- [] erzeugt die leere Liste
- len(L) bestimmt die Länge der Liste
- $L[k]$ greift auf das Element an der Stelle k zu. Allerdings beginnt die Nummerierung bei 0. Die gültigen Indizes für eine Liste der Länge n sind also die Werte $0, \ldots, n-1$.
- $L[-k]$ greift auf das k-te Element der Liste gezählt von hinten zu. $-n, \ldots, -1$, (das entspricht $(n) - n, \ldots, (n) - 1$) sind hier die gültigen Werte.
- $L[i:k]$ greift auf den Bereich $L[i], \ldots, L[k-1]$ zu.
- L.append(elem) fügt elem ans Ende der Liste L hinzu.
- L1+L2, die Listen L1 und L2 werden aneinandergehängt.

Sage code

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
L
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Sage code

```
len(L)
```

```
10
```

Sage code

```
print L[0] # erstes Element
print L[1] # zweites Element
print L[-1] # letztes Element
print L[2:5] # ein Teil der Liste
```

```
1
2
10
[3, 4, 5]
```

Hinzufügen von Elementen

Sage code

```
L.append(11)
print "Liste:", L
print "Anzahl der Elemente:", len(L)
```

```
Liste: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
Anzahl der Elemente: 11
```

Ersetzen von Elementen und Teillisten

Sage code
```
L[0] = "Hallo"
L
```

```
['Hallo', 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Sage code
```
L[1:3] = 1/2
L
```

```
['Hallo', 1/2, 4, 5, 6, 7, 8, 9, 10, 11]
```

Sage code
```
L[2] = [pi^2/6, e]
L
```

```
['Hallo', 1/2, [1/6*pi^2, e], 5, 6, 7, 8, 9, 10, 11]
```

Sehr nützlich ist die Funktion zip, die ein Tupel von Listen in eine Liste von Tupeln verwandelt.

Sage code
```
zip([1, 2, 3], ['a', 'b', 'c'], [sin, cos, tan])
```

```
[(1, 'a', sin), (2, 'b', cos), (3, 'c', tan)]
```

5.3 Erzeugen von Listen

- range(n) erzeugt die Liste $[0, 1, \ldots, n-1]$.
- range(i, j) erzeugt die Liste $[i, i+1, \ldots, j-1]$.
- range(i, j, step) erzeugt die Liste von i bis j (nicht inkludiert) mit Schrittweite step.

Sage code
```
range(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sage code
```
range(0, 101, 20)
```

```
[0, 20, 40, 60, 80, 100]
```

Sage code
```
range(10, 0, -1)
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

srange ist ähnlich wie range, aber für Fließkommazahlen

Sage code
```
srange(0, 1, 0.1, include_endpoint = True)
```

```
[0.000000000000000, 0.100000000000000, 0.200000000000000, 0.300000000000000, 0.4000000000\
00000, 0.500000000000000, 0.600000000000000, 0.700000000000000, 0.800000000000000, 0.9000\
00000000000, 1.00000000000000]
```

Sage code
```
R = RealField(prec = 200)
srange(R(0), R(0.5), R(0.1))
```

```
[0.00000000000000000000000000000000000000000000000000000000000000, 0.1000000000000000000000\
00000000000000000000000000000000000000000, 0.2000000000000000000000000000000000000000000000000000000000\
0000000, 0.30000000000000000000000000000000000000000000000000000000000000, 0.400000000000\
00000000000000000000000000000000000000000000]
```

5.4 Kurznotation für arithmetische Progressionen

Achtung: Diese Notation ist Sage spezifisch und funktioniert nicht in Standard Python.

Sage code

```
[1..100]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 2\
5, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47\
, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,\
 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, \
92, 93, 94, 95, 96, 97, 98, 99, 100]
```

Sage code

```
[1,3..10]
```

```
[1, 3, 5, 7, 9]
```

Sage code

```
[0,0.2..1]
```

```
[0.000000000000000, 0.200000000000000, 0.400000000000000, 0.600000000000000, 0.8000000000\
00000, 1.00000000000000]
```

5.5 List Comprehensions

List Comprehensions orientieren sich an mathematischer Mengenschreibweise, und sind eine Elegante Art um komplexe Listen zu generieren.

Sage code

```
[x^2 for x in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Sage code

```
var('x')
[sin(x).diff(i) for i in [0..4]]
```

```
[sin(x), cos(x), -sin(x), -cos(x), sin(x)]
```

Sage code

```
[x^2 for x in [1..20] if x % 2 == 0]
```

```
[4, 16, 36, 64, 100, 144, 196, 256, 324, 400]
```

Sage code

```
[(x,y) for x in [-3..3] for y in [-2..2] if -1 <= x-y <= 1]
```

```
[(-3, -2), (-2, -2), (-2, -1), (-1, -2), (-1, -1), (-1, 0), (0, -1), (0, 0), (0, 1), (1, \
0), (1, 1), (1, 2), (2, 1), (2, 2), (3, 2)]
```

5.6 Strings

Strings sind im wesentlichen Listen von Buchstaben, allerdings sind Strings aus Performencegründen **nicht mutable**, genauso wie Tupel.

Sage code

```
s1 = "computer"
s2 = "mathematik"
s3 = "informatik"
```

Sage code

```
s4 = s1.capitalize() + s2
s4
```

```
'Computermathematik'
```

Sage code

```
s = s4 + " (%s)" % s3.capitalize()
s
```

```
'Computermathematik (Informatik)'
```

Sage code

```
len(s)
```

```
31
```

Sage code

```
s[8:18].capitalize()
```

```
'Mathematik'
```

Sage code

```
s.split()
```

```
['Computermathematik', '(Informatik)']
```

Sage code

```
s.upper()
```

```
'COMPUTERMATHEMATIK (INFORMATIK)'
```

Sage code

```
suchstring = "Informatik"

istart=s.find(suchstring)
istart
```

```
20
```

Sage code

```
s[istart:istart + len(suchstring)]
```

```
'Informatik'
```

5.7 Dictionaries

Ein Dictionary speichert Zuordnungen von je einem Wert zu einem Schlüsselwert
Einige nützliche Funktionen:

- {}, erzeugt ein leeres Dictionary
- d[s], gibt das Element mit Schlüssel zurück
- keys(), gibt die Liste der Schlüssel zurück
- values(), die Liste der Werte
- items(), erzeugt eine Liste von allen (Schlüssel, Wert) Paaren
- haskey(s), gibt zurück, ob der Schlüssel s vorhanden ist

Sage code

```
huss = {'name': 'Wilfried Huss', 'office': 'C305', 'email': 'huss@finanz.math.tugraz.at'}
huss
```

```
{'name': 'Wilfried Huss', 'office': 'C305', 'email': 'huss@finanz.math.tugraz.at'}
```

Sage code

```
huss['name']
```

```
'Wilfried Huss'
```

Sage code

```
huss.keys()
```

```
['name', 'office', 'email']
```

Sage code

```
huss.values()
```

```
['Wilfried Huss', 'C305', 'huss@finanz.math.tugraz.at']
```

Sage code

```
huss.items()
```

```
[('name', 'Wilfried Huss'), ('office', 'C305'), ('email', 'huss@finanz.math.tugraz.at')]
\
```

Sage code

```
huss.has_key('name'), huss.has_key('nachname')
```

```
(True, False)
```

Sage code

```
for (key, value) in huss.iteritems():
  print key, ":", value
```

```
name : Wilfried Huss
office : C305
email : huss@finanz.math.tugraz.at
```

Sage code

```
primzahlen = {}
p = 1;
for i in [1..30]:
  p = next_prime(p)
  primzahlen[p] = i

primzahlen
```

```
{2: 1, 3: 2, 5: 3, 7: 4, 11: 5, 13: 6, 17: 7, 19: 8, 23: 9, 29: 10, 31: 11, 37: 12, 41: 1\
3, 43: 14, 47: 15, 53: 16, 59: 17, 61: 18, 67: 19, 71: 20, 73: 21, 79: 22, 83: 23, 89: 24\
, 97: 25, 101: 26, 103: 27, 107: 28, 109: 29, 113: 30}
```

Sage code

```
p = 109
print "%d ist die %d. Primzahl" % (p, primzahlen[p])
```

```
109 ist die 29. Primzahl
```

5.8 Mengen

Sage code

```
s1 = set([1..10])
s2 = set([5..15])
s3 = set([1, 1, 1, 1])
s1; s2; s3
```

```
set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
set([5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
set([1]\
)
```

Sage code

```
s1.union(s2)
```

```
set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
```

Sage code

```
s1.intersection(s2)
```

```
set([5, 6, 7, 8, 9, 10])
```

Sage code

```
s1.symmetric_difference(s2)
```

```
set([1, 2, 3, 4, 11, 12, 13, 14, 15])
```

6 Kontrollstrukturen

Python die Programmiersprache von Sage verwendet Einrückungen zur Definition von Blöcken.

6.1 If-Statement

Sage code

```
a = -4
if a > 0:
  print a, "ist positiv"
elif a < 0:
  print a, "ist negativ"
else:
  print a, "ist null"
```

```
-4 ist negativ
```

6.2 For-Schleife

Sage code

```
for i in [1,2,3,4]:
  j = i^2
  print j
```

```
1
4
9
16
```

6.3 While-Schleife

Sage code

```
i = 1
while i < 20:
  if i.is_prime():
    print "%2d ist eine Primzahl" % i
  i += 1
```

```
2 ist eine Primzahl
 3 ist eine Primzahl
 5 ist eine Primzahl
 7 ist eine Primzahl
11 ist\
 eine Primzahl
13 ist eine Primzahl
17 ist eine Primzahl
19 ist eine Primzahl
```

7 Definieren von Funktionen

Sage code

```
def absolutbetrag(x):
  if x < 0:
    return -x
  else:
    return x
```

Sage code

```
absolutbetrag(-3); absolutbetrag(6)
```

```
3
6
```

Selbstdefinierte Funktionen funktionieren automatisch für Typen, die alle in der Funktion verwendeten Methoden und Operationen unterstützen.
In unserem Beispiel:

- Vergleichsoperator ¡
- Negation -

Sage code

```
R = RealField(prec = 200)
absolutbetrag(R.random_element(-10, -5))
```

```
9.6862679206977625062752945168460557973263756378085563603966
```