Sage 1: Zahlen

Vorsicht: bei den untenstehenden Beispielen wird nur jeweils das Ergebnis der letzten Zeile einer Box tatsächlich ausgegeben! Zum Experimentieren Befehle jeweils in eigene Boxen kopieren oder Box mit der Tastenkombination STRG plus ; in mehrere Boxen zerteilen und einzeln ausführen. STRG plus BACKSPACE verbindet zwei Boxen wieder.
Um eigene Worksheets mit Text und Beschreibungen zu ergänzen, im Sage Notebook einfach oben rechts auf "Edit" klicken. HTML kann zur Formatierung verwendet werden, ebenso LaTeX für Formeln (Mathe-Umgebung wie gewohnt verwenden). Die Sage-Boxen sind in der Text-Ansicht mit geschwungenen Klammern begrenzt.

Zahlen

Ganzzahlen

Sage kann mit beliebig großen ganzen Zahlen umgehen. Die Datentyp dazu heißt "Integer" (Beschreibung "Integer Ring").

```
a = 1234 + 2^1000
print type(a)
print parent(a)
```

```
<type 'sage.rings.integer.Integer'>
Integer Ring
```

Die Funktionen "type" und "parent" geben für beliebige Variablen deren internen Datentyp an. Neben dem Datentyp kann man auch testen, ob eine gewisse Variable x (unabhängig vom Datentyp) eine ganze Zahl im mathematischen Sinne ist, d.h. $x \in \mathbb{Z}$:

```
x = 6/3
w = 6.0/3.0
y = 1/3
z = sqrt(4)
x in ZZ; w in ZZ; y in ZZ; z in ZZ
```

```
True
True
False
True
```

Einige praktische Funktionen im Zusammenhang mit Primzahlen und Teilbarkeit:

```
a = 23452345417985
print factor(a)
print is_prime(a)
print next_prime(a)
print abs(a - next_prime(a))
print gcd(a, x)
print divisors(a)
```

```
5 * 11 * 415253 * 1026859
False
23452345417999
14
1
[1, 5, 11, 55, 415253, 1026859, 2076265, 4567783, 5134295, 11295449,
22838915, 56477245, 426406280327, 2132031401635, 4690469083597,
23452345417985]
```

Einige Funktionen erlauben sowohl die Schreibweise var.fun() als auch fun(var), zum Beispiel:

```
factor(a) == a.factor()
```

```
True
```

Um die Faktoren weiter verwenden zu können, wandelt man das Ergebnis am besten in eine

Liste um (die Liste enthält Tupel der Form (Primzahl, Vielfachheit)):

```
list(factor(a))
```

```
[(5, 1), (11, 1), (415253, 1), (1026859, 1)]
```

Ziffern (binär, dezimal, ...) einer Zahl:

```
print a.digits()
print len(a.digits())
print sum(a.digits())
print a.digits(base=2)
```

```
[5, 8, 9, 7, 1, 4, 5, 4, 3, 2, 5, 4, 3, 2]
14
62
[1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1,
0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```

Wenn nicht genau sicher ist, wie eine Funktion heißt, gibt es mit der Tabulator-Taste eine Autovervollständigung bzw. eine Liste passender Funktionen (z.B. `div`+TAB, `a.`+TAB für eine Liste der möglichen Funktionen von Variable a).
Die Dokumentation zu einer Funktion (kurze Beschreibung, mögliche Parameter, Beispiele) erhält man mit ?:

```
gcd?
```

File: /usr/local/sage-6.3/local/lib/python2.7/site-packages/sage/rings/arith.py

Type: <type 'function'>

Definition: gcd(a, b=None, **kwargs)

Docstring:

The greatest common divisor of a and b, or if a is a list and b is omitted the greatest common divisor of all elements of a.

INPUT:

- `a,b` - two elements of a ring with gcd or
- `a` - a list or tuple of elements of a ring with gcd

Additional keyword arguments are passed to the respectively called methods.

OUTPUT:

The given elements are first coerced into a common parent. Then, their greatest common divisor *in that common parent* is returned.

EXAMPLES:

```
sage: GCD(97,100)
1
sage: GCD(97*10^15, 19^20*97^2)
97
sage: GCD(2/3, 4/5)
2/15
sage: GCD([2,4,6,8])
2
sage: GCD(srange(0,10000,10))  # fast  !!
10
```

Note that to take the gcd of n elements for $n \neq 2$ you must put the elements into a list by enclosing them in `[..]`. Before #4988 the following wrongly returned 3 since the third parameter was just ignored:

```
sage: gcd(3,6,2)
Traceback (click to the left of this block for traceback)
...
```

Für Modulo-Rechnung kann man in Sage natürlich den normalen Modulo-Operator verwenden:

```
5 % 3
-5 % 3
```

```
1
```

Möchte man längere Rechnungen modulo einer fixen Zahl ausführen, definiert man sich am besten den dazugehörigen "Modulo-Rechnungs-Ring", zum Beispiel für Rechnung modulo 7:

```
Z7 = Integers(7)
a = Z7(4)
-a; a^(-1); a * a^(-1); 5*a; parent(a)
```

```
3
2
1
6
Ring of integers modulo 7
```

Z7 und a bringen auch jede Menge weitere nützliche Funktionen mit - einfach ausprobieren:

```
list(Z7); Z7.random_element()
```

```
[0, 1, 2, 3, 4, 5, 6]
4
```

Brüche und exakte Ausdrücke

Brüche von Ganzzahlen werden weder abgerundet (wie Ganzzahldivision in C) noch mit Gleitkommazahlen approximiert, sondern exakt gespeichert. Das gleiche gilt auch für viele Funktionsauswertungen:

```
3/5
sin(pi/3)
```

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

```
parent(3/5); parent(sin(pi/3)); parent(6/2)
```

```
Rational Field
Symbolic Ring
Rational Field
```

Rechnen mit exakten Ausdrücken ergibt wieder exakte Ausdrücke:

```
a = sin(pi/3)/sqrt(3)
a; parent(a); a in QQ
```

```
1/2
Symbolic Ring
True
```

Auch hier kann man prüfen, ob ein Ausdruck beispielsweise eine rationale Zahl ($x \in \mathbb{Q}$) oder zumindest eine reelle Zahl ($x \in \mathbb{R}$) ist:

```
sin(pi/3) in QQ
sqrt(3) in QQ
sin(pi/3)/sqrt(3) in QQ
pi in QQ
pi in RR
```

```
True
```

Kommazahlen

Direkt als Kommazahlen angegebene Werte werden standardmäßig als Gleitkommazahlen mit 53 Bits Genauigkeit gespeichert:

```
1.0/6
parent(1.0/6)
```

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

Mithilfe der Funktion n (Abkürzung für numerical_approx) können exakte Ausdrücke durch

Gleitkommazahlen approximiert werden:

```
c = sin(pi/3)
n(c)
c.n()
```

```
0.866025403784439
```

Dabei können Genauigkeit (in Bits, mit Parameter prec) bzw. Anzahl der Dezimalstellen (Parameter digits) festgelegt werden:

```
print n(1/6, prec=1000)
print n(1/6, digits=1000)
```

```
0.16666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
666666666666666666666666666667
0.16666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666667
```

Falls man wiederholt dieselbe Genauigkeit benötigt, kann man sich auch eine Typ dafür definieren (hier R mit 1000 Bits Präzision) und den wiederholt verwenden:

```
R = RealField(1000)
c = R(sin(pi/3))
b = R(1/3)
parent(c * b)
```

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

Bei Rechnungen mit mehreren verschieden genauen Werten hat das Ergebnis die geringste in der Eingabe vorkommende Präzision:

```
ungenau = n(1/10)
genau = R(1/15)
eins = genau/ungenau
parent(eins)
```

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

```
a = (1/3).n(prec=13)
b = (1/3).n(prec=100)

parent(a*b)
```

```
Real Field with 13 bits of precision
```

Nachträgliches Konvertieren zu einer besseren Präzision hilft natürlich nicht, um ein bereits ungenaues Ergebnis (hier 53 Bits Präzision) zu verbessern:

```
a = R(1.0/3.0)
parent(a); a
```

```
Real Field with 1000 bits of precision
0.33333333333333331482961625624739099293947219848632812500000000000\
00000000000000000000000000000000000000000000000000000000000000000000\
```

```
00000000000000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000000000000000000000000\
0000000000000000000000000000
```

Will man eine Kommazahl wieder zu einer Ganzzahl runden, gibt es mehrere Möglichkeiten: normal kaufmännisch runden (round), aufrunden (ceil) oder abrunden (floor):

```
number = 3456.7 / 376
number; round(number); ceil(number); floor(number)
```

```
9.19335106382979
9
10
9
```

Komplexe Zahlen

Für die Darstellung von komplexen Zahlen $z \in \mathbb{C}$ sind die Konstanten I und i auf die imaginäre Einheit vordefiniert; in Python (und notfalls auch in Sage) kann auch die Notation mit j verwendet werden:

```
print sqrt(-1)
z = 3 + 4*i
z2 = 3 + 4j
print z
print z2
parent(z); parent(z2)
```

```
I
4*I + 3
3.00000000000000 + 4.00000000000000*I
Symbolic Ring
Complex Field with 53 bits of precision
```

```
z3 = z2 / (4 + 2*I)
z3 in CC
```

```
True
```

Um separat auf Imaginärteil, Realteil, Absolutbetrag usw. einer komplexen Zahl zugreifen zu können, gibt es diverse Hilfsfunktionen:

```
z3; imag(z3); real(z3); abs(z3)
```

```
1.00000000000000 + 0.500000000000000*I
0.500000000000000
1.00000000000000
1.11803398874989
```

Betrag (abs) und Winkel (arg) für Polarkoordinaten:

```
print abs(z)
print arg(z)
print arg(z).n()
```

```
5
arctan(4/3)
0.927295218001612
```

Vorsicht, man kann I und i (genau wie andere Konstanten wie pi, ...) auch versehentlich überschreiben:

```
I = 5
3 + 4*I
```

```
23
```

Wahrheitswerte (Bool)

Vergleiche und wahr-falsch-Fragen haben als Ergebnis-Datentyp Wahrheitswerte (Bool):

```
parent(5 in ZZ)
```

```
parent(3 < 5)
parent(4 * 5 == 20)
```

```
<type 'bool'>
```

Boolesche Werte und Bedingungen kann man mit "and, or, not" (entspricht in C &&, ||, !) verknüpfen:

```
5 in ZZ and not 3 < 5
```

```
False
```

Anders als in C sind auch mehrere Vergleiche hintereinander möglich:

```
-5 < -3 < -1
```

```
True
```

Bei Vergleichen mit exakten Ausdrücken ist es teilweise notwendig, das Ergebnis der Gleichung explizit in einen Wahrheitswert zu konvertieren:

```
sin(pi/3) == sqrt(3)/2
bool(sin(pi/3) == sqrt(3)/2)
```

```
True
```

```
```