Sage 1: Einführung, Programmieren, Zahlen

Grundlagen

Worksheets & Variablen

Einfache Ausgaben mit Worksheet-Boxen

Dieses Dokument ist ein Sage-Worksheet. In die Eingabeboxen werden Rechnungen/Befehle eingegeben, durch drücken auf "evaluate" oder strg+enter ausgeführt werden. Das Ergebnis wird unter der Box angezeigt.
Um erklärenden Text zwischen den Eingabeboxen hinzuzufügen, kann unter dem "Edit"-Button HTML-Code eingefügt werden. Mathematische Formeln im erklärenden Text können in LaTeX-Syntax eingegeben werden, zum Beispiel: $\int_0^{\pi/2} \cos x \, \mathrm{d}x = 1$.
Einfache Taschenrechner-artige Eingaben:

```
5+4
```

```
9
```

Bei mehreren Eingaben werden alle Werte berechnet, aber nur das Ergebnis der letzten Rechnung wird angezeigt:

```
5*3
5-4
```

```
1
```

Um Ergebnisse weiterverwenden zu können, kann man es einer Variable zuweisen (hier mit dem Namen "a"). Da das Ergebnis einer Zuweisung standardmäßig nicht anzeigt wird, schreiben wir noch eine Zeile, in der das Ergebnis von "a" (also sein Wert) angezeigt wird:

```
a = 5*2
a
```

```
10
```

Anders als z.B. in C muss man den Typ der Variablen nicht explizit angeben, sondern er ergibt sich aus dem Wert dessen, was zugewiesen wird. Im obigen Beispiel ist int*int=int, also ist a ein Integer. Man kann den Wert von a jetzt auch mit Werten eines anderen Types überschreiben; anders als in C ist der Typ von a nicht statisch, sondern dynamisch. Nach der folgenden Zuweisung ist der Typ von a eine Kommazahl (float):

```
a = 5.0/2.0
a
```

```
2.50000000000000
```

Und jetzt ein String:

```
a = "asdf"
a
```

```
'asdf'
```

Neben der normalen Ausgabe (im gleichen Format wie die Eingabe) kann Sage Ergebnisse auch in LaTeX-Formatierung ausgeben (mit "show"). Ebenso kann man sich den LaTeX-Quellcode dazu anzeigen lassen (mit "latex"). Analog gibt es Darstellungen für HTML usw.

```
show(1/10)
latex(1/10+2/10)
```

$$\frac{1}{10}$$

```
\frac{3}{10}
```

Sage vs. Python

Neben den Grundrechenarten funktionieren auch viele weitere Operatoren, die man aus C und Python gewohnt ist, etwa:

```
5%2
```

```
1
```

```
5<<2
```

```
20
```

Die Datentypen und Operatoren in Sage sind aber etwas "mathematischer" als in C und Python. Der Datentyp des folgenden Ausdrucks beispielsweise ist "rationale Zahl" (Bruch), anstatt beispielsweise Integer mit Wert 0 (wie in C) oder Gleitkommazahl. Der Wert wird also exakt gespeichert und nicht durch eine Gleitkommazahl approximiert. In Binärdarstellung hat 1/10 nämlich keine exakte, endliche Darstellung wie 0.1 in dezimal, sondern unendlich viele, periodische Nachkommastellen: 0.00011001100110011... Eine Gleitkommadarstellung wuerde diese Kommastellen irgendwo abschneiden, und das Ergebnis wäre nur mehr gerundet. In Sage hingegen wird der Wert als "Bruch mit Nenner 10 und Zähler 1" gespeichert, und es wird damit exakt weitergerechnet. Ähnliches gilt auch für Wurzeln usw., sofern man sie nicht explizit in floats umwandelt (z.B. durch Multiplikation mit einem Float).

```
1/10
```

```
1/10
```

Wie oben im Bruchbeispiel schon gesehen, verhalten sich einige Operatoren anders, "mathematischer". Das liegt daran, dass eingegebene Zahlen automatisch einen Sage-Zahlen-Datentyp bekommen statt einen Python-Zahlen-Datentyp (z.B. `Integer` statt `int`), um beispielsweise eine bessere Genauigkeit zu erreichen, und eben auch die Operatoren passender belegen zu können.
Ein weiteres Beispiel ist der Zirkumflex, der in Sage das tut, was man als Mathematiker erwartet, nämlich exponentieren:

```
5^2
```

```
25
```

Um die C- (und Python-)Verhaltensweise von / und ^ zu bekommen (nämlich Ganzzahldivision und bitweises XOR), gibt es separate Operatoren:

```
5//2
```

```
2
```

```
5^^2
```

```
7
```

Es funktioniert auch der Python-Exponential-Operator:

```
5**2
```

```
25
```

Die kombinierten Zuweisungsoperatoren +=, -= usw. aus C funktionieren auch (aber fuehren jeweils die Sage-Entsprechung aus, etwa bei /)

```
a = 5 + (3 * 4)
a /= 5
show(a)
```

$$\frac{17}{5}$$

Ein String.

```
a="abc"
a
```

```
'abc'
```

"Addition" von Strings bedeutet Verkettung.

```
a+"de"
```

```
'abcde'
```

Neben den einfachen Datentypen sind Listen ein sehr wichtiger Typ in Sage (und Python). Listen werden durch Angeben der Listenelemente zwischen eckigen Klammern [] erstellt. Die Einträge müssen nicht denselben Typ haben. Man kann Listen auch verschachteln. Um Speicherangelegenheiten braucht man sich dabei als Programmierer überhaupt nicht zu kümmern (anders als bei Arrays o.ä. in C):

```
b = [5, 4/3, 23, "string", [2,3,4]]
b
```

```
[5, 4/3, 23, 'string', [2, 3, 4]]
```

```
l1=[1,2]
l2=[]
print l1, l2
```

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

Element an Liste anhöngen mit append() und Konkatenation mit "+"

```
l1.append(8)
l2.append(3)
l3=l1+l2
print l1, l3
```

```
[1, 2, 8] [1, 2, 8, 3]
```

Zugreifen auf einzelne Elemente der Liste. Index startet mit 0!

```
l1[1]
```

```
2
```

Teillisten

```
print l1[1:]
```

```
[2, 8]
```

```
l4=[0,1,2,3,4,5]
print l4[1:3]
```

```
[1, 2]
```

Der Integer-Typ ist nicht auf 32-Bit-Werte eingeschränkt, sondern kann mit beliebig großen Zahlen umgehen:

```
2^1000
```

```
10715086071862673209484250490600018105614048117055336074437503883703\
51051124936122493198378815695858127594672917553146825187145285692314\
04359845775746985748039345677748242309854210746050623711418779541821\
53046474983581941267398767559165543946077062914571196477686542167660\
42983165262438683720566806937 6
```

Bei Kommazahlen kann man die Präzision (Anzahl der Binär-Stellen/Bits) auch frei wählen. Der Befehl RealField erstellt quasi den dazugehörigen Datentyp (Parameter ist die Anzahl der Bits für die Rechen-Genauigkeit). Um Zahlen in dieser Präzision zu berechnen, kann man R() zum Umwandeln verwenden. Vorsicht: Bei Rechnungen wird die Präzision des jeweils ungenauesten Wertes übernommen. Im folgenden Beispiel findet nur die erste Rechung tatsächlich mit 1000 Bits Präzision statt; die zweite rechnet zuerst in Standard-Präzision und konvertiert erst danach zu 1000 Bit durch Auffüllen mit 0; die dritte übernimmt bei der Division die Standard-Präzision des ungenaueren Operanden:

```
R = RealField(1000)
genau = R(5.0)/R(3.0)
ungenau1 = R(5.0/3.0)
ungenau2 = R(5.0)/3.0
genau, ungenau1, ungenau2
```

```
(1.66666666666666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666667,
1.66666666666666740681534975010436028242111206054687500000000000000\
00000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000, 1.66666666666667)
```

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` bzw. `parent` geben für beliebige Variablen deren internen Python- bzw. Sage-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)
```

```
0.166666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666666666666666666666666666666666666666666666\
666666666666666666666666666667
0.166666666666666666666666666666666666666666666666666666666666666666\
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
```

```
c.prec()
```

```
1000
```

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\
0000000000000000000000000000000000000000000000000000000000000000000\
0000000000000000000000000000000000000000000000000000000000000000000\
0000000000000000000000000000000000000000000000000000000000000000000\
00000000000000000000000000
```

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
```

```
0.1*10 == 1.0
```

```
True
```

Programmieren in Sage

Grundsätzlich basiert Sage auf Python, und verwendet daher dieselbe Syntax für Schleifen, if-Abfragen, Funktionen und so weiter.
Vorsicht: Sage verwendet Python 2, nicht Python 3, es gibt daher einige Unterschiede (z.B. print-Statement).
Zusätzlich zu den vielen Bibliotheken für Mathematik bringt Sage auch einige Syntax-Erweiterungen und -Modifikationen mit (z.B. mathematische Operatoren, vereinfachte Range-Listen).
Die folgenden Inhalte sind großteils eine Wiederholung aus Programmieren 0:

Kontrollstrukturen

If-Statement

	If	If-Else	If-Elif-Else
In C:	`if (condition) { statements }`	`if (condition) { statements } else { statements }`	—
In Sage/Python:	`if condition: statements`	`if condition: statements else: statements`	`if condition: statements elif condition: statements ... elif condition: statements else: statements`

Beispiele

In C:

```
float x = 355.0/113.0;
if (x >= 3 && x <= 5)
{
  printf("%d is between 3 and 5\n", x);
}
```

In Python/Sage:

```
x = pi
if x >= 3 and x <= 5:
  print x, "is between 3 and 5"
```

```
pi is between 3 and 5
```

oder auch:

```
if 3 <= x <= 5:
  print "{x} is between 3 and 5".format(x=x)
```

```
pi is between 3 and 5
```

Einrückung statt Klammern!
Einrückungslevel (Anzahl der Tabs/Spaces) wird durch erste Zeile nach dem if festgelegt und muss für die folgenden Zeilen innerhalb dieses if beibehalten werden.

Richtig:

```
if 3 <= x <= 5:
  sq = x^2
  print "{x} is between 9 and 25".format(x=sq)
```

```
pi^2 is between 9 and 25
```

```
if 3 <= x <= 5:
        sq = x^2
        print "{x} is between 9 and 25".format(x=sq)
```

```
pi^2 is between 9 and 25
```

Falsch:

```
x = 19
if 3 <= x <= 5:
        xsq = x^2
  print "{x} is between 9 and 25".format(x=xsq)
```

```
Traceback (click to the left of this block for traceback)
...
IndentationError: unindent does not match any outer indentation
level
```

else und **elif** für alternative Fälle:

```
x=pi
if 3 <= x < 5:
  print "x is in [3,5)"
elif 5 <= x < 7:
  print "x is in [5,7)"
elif 7 <= x < 9:
  print "x is in [7,9)"
else:
  print "x is in (-oo,3) or in [9,+oo)"
```

```
x is in [3,5)
```

While-Statement

	While	While-Else
In C:	`while (condition) { statements }`	—
In Sage/Python:	`while condition: statements`	`while condition: statements else: statements`

Neben der normalen **while**-Schleife unterstützt Python auch eine **while-else**-Schleife: Der **while**-Teil wird wiederholt, bis die *condition* falsch wird; danach wird einmal der **else**-Block ausgeführt. Die Ausführung findet nur statt, wenn die Schleife "regulär" beendet wird (d.h. sobald die Bedingung falsch wird), aber nicht bei vorzeitigem Abbruch der Schleife (durch **return**, **break** o.ä.).

For-Statement

	For	For-Else
In C:	`for (init; condition; update) { statements }`	—
In Sage/Python:	`for element in sequence: statements`	`for element in sequence: statements else: statements`

Im Unterschied zu C ist die **for**-Schleife in Python/Sage nicht definiert durch Abbruchbedingung, Startwerte und Updatefunktion, sondern iteriert immer über alle Elemente einer *sequence*. Eine *sequence* ist ein aufzählbarer (iterierbarer) Datentyp, beispielsweise eine Liste, ein String, ein Tuple, ein Generator, ...
Wie bei der **while**-Schleife wird der **else**-Block der **for**-Schleife dann ausgeführt, wenn die Schleife am Ende der Liste angekommen ist; nicht jedoch, wenn die Schleife vorzeitig (z.B. durch **break**) beendet wurde.

In C:

```
int i;
for (i = 0; i < 10; i++)
{
  printf("%d^2 = %d\n", i, i*i);
}
```

In Python/Sage wird stattdessen eine Liste der Werte angegeben, für die die Schleife ausgeführt werden soll:

```
for i in [0,1,2,3,4,5,6,7,8,9]:
  print i, "^2 = ", i^2
```

```
0 ^2 =  0
1 ^2 =  1
2 ^2 =  4
3 ^2 =  9
4 ^2 =  16
5 ^2 =  25
6 ^2 =  36
7 ^2 =  49
8 ^2 =  64
9 ^2 =  81
```

```
for letter in "alternativlos":
    if letter in "aeiou":
        print 3 * letter,
    else:
        print letter,
```

```
aaa l t eee r n aaa t iii v l ooo s
```

Natürlich braucht man diese Sequence nicht immer explizit selbst händisch erzeugen, sondern viele eingebaute Funktionen liefern eine Liste oder andere solche Sequence als Ergebnis zurück:
In Sage/Python:

```
for i in range(10):
  print i, "^2 = ", i^2
```

```
0 ^2 =  0
1 ^2 =  1
2 ^2 =  4
3 ^2 =  9
4 ^2 =  16
5 ^2 =  25
6 ^2 =  36
7 ^2 =  49
8 ^2 =  64
9 ^2 =  81
```

In Sage:

```
for i in [0..9]:
  print i, "^2 = ", i^2
```

```
0 ^2 =  0
1 ^2 =  1
2 ^2 =  4
3 ^2 =  9
4 ^2 =  16
5 ^2 =  25
6 ^2 =  36
7 ^2 =  49
8 ^2 =  64
9 ^2 =  81
```

Oder mit anderer Schrittweite

```
for i in [0,2..9]:
  print i, "^2 = ", i^2
```

```
0 ^2 =  0
2 ^2 =  4
4 ^2 =  16
6 ^2 =  36
8 ^2 =  64
```

Achtung: `range` erzeugt den Python-Integertyp. Manche Funktionen funktionieren damit nicht!

```
for i in range(3,6):
    print parent(i)
    i.is_prime()
```

```
<type 'int'>
Traceback (click to the left of this block for traceback)
...
```

```
AttributeError: 'int' object has no attribute 'is_prime'
```

Für solche Zwecke besser `srange` verwenden.

```
for i in srange(3,6):
    print parent(i)
    i.is_prime()
```

```
Integer Ring
True
Integer Ring
False
Integer Ring
True
```

Im Prinzip kann über alle endlichen Kollektionen iteriert werden. Mehr dazu später unter dem Stichwort "generator".

```
for k in Integers(7):
    print k
```

```
0
1
2
3
4
5
6
```

Was ist mit Switch-Case, Goto, ...?

Switch-case kann man beispielsweise mit **elif** nachbauen.
Die wenigen erträglichen Verwendungen von goto werden eigentlich abgedeckt durch

- **return**: zur aufrufenden Funktion zurückkehren
- **break**: Schleife vorzeitig beenden
- **continue**: direkt zum nächsten Schleifendurchlauf
- **raise**: Exception (Fehler) melden
- **try/except**: Exception (Fehler) behandeln

Funktionen

Funktionen definieren und verwenden

In C:

```
int plus(int a, int b)
{
  return a + b;
}
```

In Sage/Python ist es nicht notwendig, Datentypen (insb. Typ des Rückgabewerts) anzugeben:

```
def plus(a, b):
    return a + b
```

Diese Funktion funktioniert also für alle Datentypen, die den +-Operator unterstützen, auch Strings, Listen etc:

```
print plus(4, 6)
```

```
print plus("hello", "world")
print plus(pi/2, 1/2)
```

Wenn für Parameter Default-Werte angegeben werden, müssen beim Aufruf nicht alle Parameter angegeben werden:

```
def diff(a, b=1):
    return a - b
```

```
diff(5)
```

```
4
```

```
diff(2,5)
```

```
-3
```

```
diff(b=2, a=5)
```

```
3
```

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.

```
```