Sage 0: Einführung

Basics

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'
```

Die Datentypen und Operatoren in Sage sind etwas "mathematischer" als in C. 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
```

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"). Ebenso gibt es Darstellungen für HTML usw.

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

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

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

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

```
5%2
```

```
1
```

```
5<<2
```

```
20
```

Wie oben im Bruchbeispiel schon gesehen, verhalten sich einige Operatoren aber anders, "mathematischer". 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)
```

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

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]]
```

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

```
2^1000
```

```
107150860718626732094842504906000181056140481170553360744375038837\
```

```
5105112493612249319837881569585812759467291755314682518714528569231 4\
0435984577574698574803934567774824230985421074605062371141877954182 1\
5304647498358194126739876755916554394607706291457119647768654216766 0\
4298316526243868372056680693 76
```

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
```

```
(1.6666666666666666666666666666666666666666666666666666666666666666\
6666666666666666666666666666666666666666666666666666666666666666666666\
6666666666666666666666666666666666666666666666666666666666666666666666\
6666666666666666666666666666666666666666666666666666666666666666666666\
66666666666666666666666666667,
1.6666666666666674068153497501043602824211120605468750000000000000\
0000000000000000000000000000000000000000000000000000000000000000000000\
0000000000000000000000000000000000000000000000000000000000000000000000\
0000000000000000000000000000000000000000000000000000000000000000000000\
0000000000000000000000000, 1.66666666666667)
```

Neben simplen, einzeiligen Rechnungen können auch Kontrollstrukturen, ähnlich wie aus C bekannt, verwendet werden - mehr dazu gleich :)

```
if a > 3:
    print "groesser als 3:", a
    print "und so eine Art printf: {eins} {z} {d}".format(eins=4, z=7, d=a)
    show(a)
else:
    print "kleiner als 3"
    show(a)
```

```
groesser als 3: 17/5
und so eine Art printf: 4 7 17/5
```

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

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
```

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)
```

```
10
helloworld
1/2*pi + 1/2
```

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

```
def inc(a, b=1):
    return a + b
```

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

```
7
```