3.Vorlesung 13.12.2024

Inhaltsverzeichnis

1. Listen für Fortgeschrittene (Teil 2)
2. Weitere interessante Datentypen
3. Formales Rechnen
4. Gleichungen
5. Polynome
6. Gleichungen in mehreren Variablen
7. Zeitmessung

Listen für Fortgeschrittene: Teil 2

Die Operatoren **+** und * sind für Listen definiert und erlauben das Aneinanderketten sowie Wiederholen von Listen.

In [1]:
```
l = [1,..,10]
l
```

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

In [2]:
```
l.append(11)
```

Die Liste wird dabei verändert.

In [3]:
```
l
```

Out[3]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [4]:
```
l+[4, 8, 15, 16, 23, 42]
```

Out[4]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 4, 8, 15, 16, 23, 42]
```

In [5]:
```
l
```

Out[5]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [6]:
```
l*2
```

Out[6]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

In [7]:
```
l
```

Out[7]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Wie bereits aus Matlab bekannt, kann über Doppelpunkt-Syntax auf Teillisten zugegriffen werden. Hierbei ist zu beachten, dass die obere Schranke (anders als in Matlab) nicht mehr zum Indexbereich gehört.

In [8]:
```
print(l[2:7])
print(l[2:10:2])
```

```
[3, 4, 5, 6, 7]
[3, 5, 7, 9]
```

Ebenfalls erlaubt uns die Doppelpunkt-Syntax, bestimmte Teile einer Liste zu überschreiben.

In [9]:
```
[42]*4
```

Out[9]:
```
[42, 42, 42, 42]
```

In [10]:
```
l[2:10:2] = [42]*4
l
```

Out[10]:
```
[1, 2, 42, 4, 42, 6, 42, 8, 42, 10, 11]
```

map und **filter**

Mittels **map** und **filter** lassen sich aus bestehenden Listen neue Listen erzeugen.

map(*funktion*, *liste*) bildet die Liste der Abbilder unter der gegebenen Funktion, also `map(f, [x0,x1,x2]) = [f(x0), f(x1), f(x2)]`.

In [11]:
```
l = srange(0,10)
l
```

Out[11]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [12]:
```
l2 = map(factorial, l)
l2
```

Out[12]:
```
<map object at 0x7fd2ae23d000>
```

In [13]:
```
list(l2)
```

Out[13]:
```
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

Die ursprüngliche Liste bleibt hierbei unverändert.

In [14]:
```
l
```

Out[14]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

map-Objekte können nur einmal durchlaufen werden!

In [15]:
```
list(l2)
```

Out[15]:
```
[]
```

filter(***funktion, liste***) bildet die Liste jener Elemente, für die die gegebene Funktion *True* ergibt.</p>

In [16]:
```
l3 = filter(is_prime, l)
l3
```

Out[16]:
```
<filter object at 0x7fd2b0c3c640>
```

In [17]:
```
list(l3)
```

Out[17]:
```
[2, 3, 5, 7]
```

Die ursprüngliche Liste bleibt hierbei unverändert.

In [18]:
```
l
```

Out[18]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Auch filter-Objekte können nur einmal durchlaufen werden!

In [19]:
```
l3 = filter(is_prime, l)
l3
```

Out[19]:
```
<filter object at 0x7fd2ae23c3a0>
```

In [20]:
```
for i in l3:
    print(i)
```

```
2
3
5
7
```

In [21]:
```
list(l3)
```

Out[21]:
```
[]
```

List comprehensions

So genannte **list comprehensions** sind eine Kurzschreibweise, die das schnelle Erzeugen einer Liste aus einer anderen Liste erlauben.
Die Syntax hier lautet **[*Element* for *Laufvariable* in *Liste* if *Bedingung*]**, wobei die Bedingung *optional* ist.

In [22]:
```
[n^2 for n in range(10)]
```

Out[22]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Hierbei ist "n" die Laufvariable, "n^2" das Element der neuen Liste, "range(0,10)" die ursprüngliche Liste. Das ganze kann auch an Bedingungen gerknüpft werden:

In [23]:
```
squares = [n*n for n in range(0,10) if 2.divides(n)]
squares
```

Out[23]:
```
[0, 4, 16, 36, 64]
```

reduce

reduce "verschmilzt" alle Elemente einer Liste von links beginnend gemäß einer angegebenen Funktion.

Zunächst werden die ersten zwei Elemente verschmolzen, dann das Ergebnis mit dem dritten Element, dann das Ergebnis mit den vierten usw. bis alle Elemente verbraucht sind.

reduce$(x_1,x_2,x_3,...,x_n) = f(f(...f(f(x_1,x_2),x_3),...),x_n)$

In [24]:
```
L = range(10)
L
```

Out[24]:
```
range(0, 10)
```

In [25]:
```
reduce(operator.add, L)      # dieser Ausdruck entspricht ((((0+1)+2)+3)+
```

Out[25]:
```
45
```

In [26]:
```
reduce(max,L)
```

Out[26]:
```
9
```

Weitere interessante Datentypen

set

Der Datentyp **set** entspricht dem mathematischen Konzept der *Menge*, also einer ungeordneten Ansammlung an verschiedenen Objekten.

Zur expliziten Umwandlung kann einfach der Datentyp wie der Name einer Funktion verwendet werden.

In [27]:
```
L = list(range(0,5))*2
parent(L),L
```

Out[27]:
```
(<class 'list'>, [0, 1, 2, 3, 4, 0, 1, 2, 3, 4])
```

In [28]:
```
S = set(L)
parent(S),S
```

Out[28]:
```
(<class 'set'>, {0, 1, 2, 3, 4})
```

In [29]:
```
L2 = list(S)
parent(L2),L2
```

Out[29]:
```
(<class 'list'>, [0, 1, 2, 3, 4])
```

In [30]:
```
S2 = {42,43,44}
```

In [31]:
```
S2
```

Out[31]:
```
{42, 43, 44}
```

In [32]:
```
parent(S2)
```

Out[32]:
```
<class 'set'>
```

In [33]:
```
S+S2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Cell In [33], line 1
----> 1 S+S2

TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

In [34]:
```
S.union(S2)
```

Out[34]:
```
{0, 1, 2, 3, 4, 42, 43, 44}
```

Tupel

Ein **Tupel** ist eine unveränderliche Aufreihung einer festen Anzahl an Objekten. Tupel können direkt über mehrere Variablen ausgebreitet werden.

Hierdurch bieten sich Tupel perfekt zur Rückgabe mehrerer Werte aus einer Funktion an.

In [35]:
```
T = (4, 8, 15)
one,two,three = T
print(one)
print(two)
print(three)
```

```
4
8
15
```

Tupel können nicht verändert werden

In [36]:
```
T[1]
```

Out[36]:
```
8
```

In [37]:
```
T[1]=9
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Cell In [37], line 1
----> 1 T[Integer(1)]=Integer(9)

TypeError: 'tuple' object does not support item assignment
```

ähnliches geht auch mit Listen:

In [38]:
```
[a,b,c]=[1,2,3]
```

In [39]:
```
a
```

Out[39]:
```
1
```

Das ist praktisch beim Vertauschen von Variablen.

In [40]:
```
a=1
b=2
a,b = b,a
```

In [41]:
```
a
```

Out[41]:
```
2
```

In [42]:
```
b
```

Out[42]:
```
1
```

oder beim Durchlaufen von Listen

In [43]:
```
L = [(i,i*i,i*i*i) for i in [16, 23, 42]]
L
```

Out[43]:
```
[(16, 256, 4096), (23, 529, 12167), (42, 1764, 74088)]
```

In [44]:
```
for n, nsquare, ncube in L:
    print(n,ncube-nsquare)
```

```
16 3840
23 11638
42 72324
```

Die herkömmliche Methode ist unübersichtlich

In [45]:
```
for l in L:
    print(l[0],l[2]-l[1])
```

```
16 3840
23 11638
42 72324
```

Dictionary

Das **dictionary** ist eine Zuordnung von *Schlüsseln* zu *Werten*. Es verhält sich ähnlich einer Liste, mit dem Unterschied, dass - im Gegensatz zu den ganzzahligen *Indizes* einer Liste - (fast) jedes beliebige Objekt als *Schlüssel* für einen Eintrag verwendet werden kann.

Dictionaries sehen auf den ersten Blick wie Mengen von Paaren aus (geschwungene Klammern {}):

In [51]:
```
D = { 'a': 1, 25: 15, ZZ: QQ }
print(D)
print(D['a'])
D['z'] = SR
print(D)
```

```
{'a': 1, 25: 15, Integer Ring: Rational Field}
1
{'a': 1, 25: 15, Integer Ring: Rational Field, 'z': Symbolic Ring}
```

Sie haben aber einen eigenen Typ

In [52]:
```
parent(D)
```

Out[52]:
```
<class 'dict'>
```

In [53]:
```
D[1]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call las
Cell In [53], line 1
----> 1 D[Integer(1)]

KeyError: 1
```

In [54]:
```
D[1] = 8
```

In [55]:
```
D
```

Out[55]:
```
{'a': 1, 25: 15, Integer Ring: Rational Field, 'z': Symbolic Ring, 1: 8}
```

Wie auch Listen sind **set**, **tuple** und **dictionary** Beispiele für *iterierbare Objekte*, die mittels einer **for**-Schleife durchlaufen werden können.

In [56]:
```
S1 = set([1,2,3])
S2 = set([3,4,8,2])
for v in (S2-S1):
    print(v)
```

```
8
4
```

Rechnen mit formalen Ausdrücken

Um mit formalen Ausdrücken zu rechnen, muß man entsprechende Variablennamen deklarieren, ausgenommen die Variable x

In [57]:
```
x
```

Out[57]:
```
x
```

In [58]:
```
parent(x)
```

Out[58]:
```
Symbolic Ring
```

In [59]:
```
x^2+4
```

Out[59]:
```
x^2 + 4
```

Formale Ausdrücke können natürlich auch Variablen zugeordnet werden.

In [60]:
```
a=x^3
a
```

Out[60]:
```
x^3
```

Man mache sich den prinzipiellen Unterschied zwischen einer "Variable" im Sinn der Informatik (hier z.B. die Variable a) und der symbolischen "Variable" x klar: a bezeichnet einen Platz im Speicher des Computers, in dem ein gewisser Wert abgelegt ist, und dieser Wert hat einen Typ. Dagegen ist x hier eine Variable im Sinne der Analysis, d.h., ein Platzhalter, für den verschiedene Werte eingesetzt werden *können*.

Nicht die Variable a hat einen festen Typ, sondern ihr Wert; mit anderen Worten, die Typen sind *dynamisch*.

In [61]:
```
parent(a)
```

Out[61]:
```
Symbolic Ring
```

In [62]:
```
a=1
a
```

Out[62]:
```
1
```

In [63]:
```
parent(a)
```

Out[63]:
```
Integer Ring
```

alle formalen Symbole außer x müssen als solche deklariert werden.

In [64]:
```
y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call las
Cell In [64], line 1
----> 1 y

NameError: name 'y' is not defined
```

In [65]:
```
var('y')
y
```

Out[65]:
```
y
```

In [66]:
```
parent(y)
```

Out[66]:
```
Symbolic Ring
```

das kann auch gruppenweise passieren.

In [67]:
```
var('u,v')
```

Out[67]:
```
(u, v)
```

In [68]:
```
parent(u)
```

Out[68]:
```
Symbolic Ring
```

Formale Ausdrücke werden vorerst nicht ausmultipliziert

Löschen mit `reset`

In [69]:
```
reset('y')
```

In [70]:
```
y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call las
Cell In [70], line 1
----> 1 y

NameError: name 'y' is not defined
```

In [71]:
```
var('y')
```

Out[71]:
```
y
```

In [72]:
```
f=(x+y)^2
f
```

Out[72]:
```
(x + y)^2
```

das muß explizit angefordert werden.

In [73]:
```
g=expand(f)
g
```

Out[73]:
```
x^2 + 2*x*y + y^2
```

alternativ kann die Methode `expand` direkt angefordert werden.

In [74]:
```
f.expand()
```

Out[74]:
```
x^2 + 2*x*y + y^2
```

dabei wird der ursprüngliche Ausdruck nicht verändert

In [75]:
```
f
```

Out[75]:
```
(x + y)^2
```

Die inverse Operation zum ausmultiplizieren ist **factor**. Zum Vergleich: Wenn man **factor** auf eine ganze Zahl anwendet, wird trotz gleichen Namens der Funktion eine ganz andere Methode aufgerufen:

In [76]:
```
factor(g)
```

Out[76]:
```
(x + y)^2
```

In [77]:
```
factor(442)
```

Out[77]:
```
2 * 13 * 17
```

- Mit formalen Ausdrücken können alle herkömmlichen symbolischen Operationen durchgeführt werden. Es können alle herkömmlichen Ausdrücke differenziert werden.

In [78]:
```
diff(sin(x),x)
```

Out[78]:
```
cos(x)
```

Integration ist bekanntlich nicht immer möglich. Für sogenannte elementare Funktionen (Polynome, rationale Funktionen, exp, log, sin, cos und alles, was daraus erzeugt werden kann, sowie algebraische Funktionen können mit dem Risch-Algorithmus (1967) gelöst werden. Wenn eine gegebene elementare Funktion eine elementare Stammfunktion hat, kann sie berechnet werden, wenn die Stammfunktion nicht elementar ist, dann erkennt es der Algorithmus und liefert eine entsprechende Fehlermeldung.

In [79]:
```
integrate(sin(x),x)
```

Out[79]:
```
-cos(x)
```

In [80]:
```
integrate(sin(x),x,0,1)
```

Out[80]:
```
-cos(1) + 1
```

In [83]:
```
integrate(sin(exp(sqrt(x))),x)
```

Out[83]:
```
integrate(sin(e^sqrt(x)), x)
```

In [84]:
```
show(_)
```

Einige nicht-elementare Funktionen können dennoch mit heuristischen Methoden berechnet werden.

In [85]:
```
integrate(exp(x^2),x)
```

Out[85]:
```
-1/2*I*sqrt(pi)*erf(I*x)
```

In [86]:
```
erf?
```

Allerdings begibt man sich in Teufels Küche: Während der Risch-Algorithmus bewiesenermaßen korrekt ist, können heuristische Methoden alle möglichen Fehler liefern. Darüberhinaus sind gewisse Fragen formal *nicht entscheidbar*, d.h., es gibt keinen Algorithmus, der feststellen kann, ob ein formaler Ausdruck 0 ergibt, wenn neben elementaren Funktionen auch z.B. die Betragsfunktion erlaubt wird (Satz von Richardson, 1968).

In sage version 8 ergab das folgende Integral den **falschen** Wert -1.

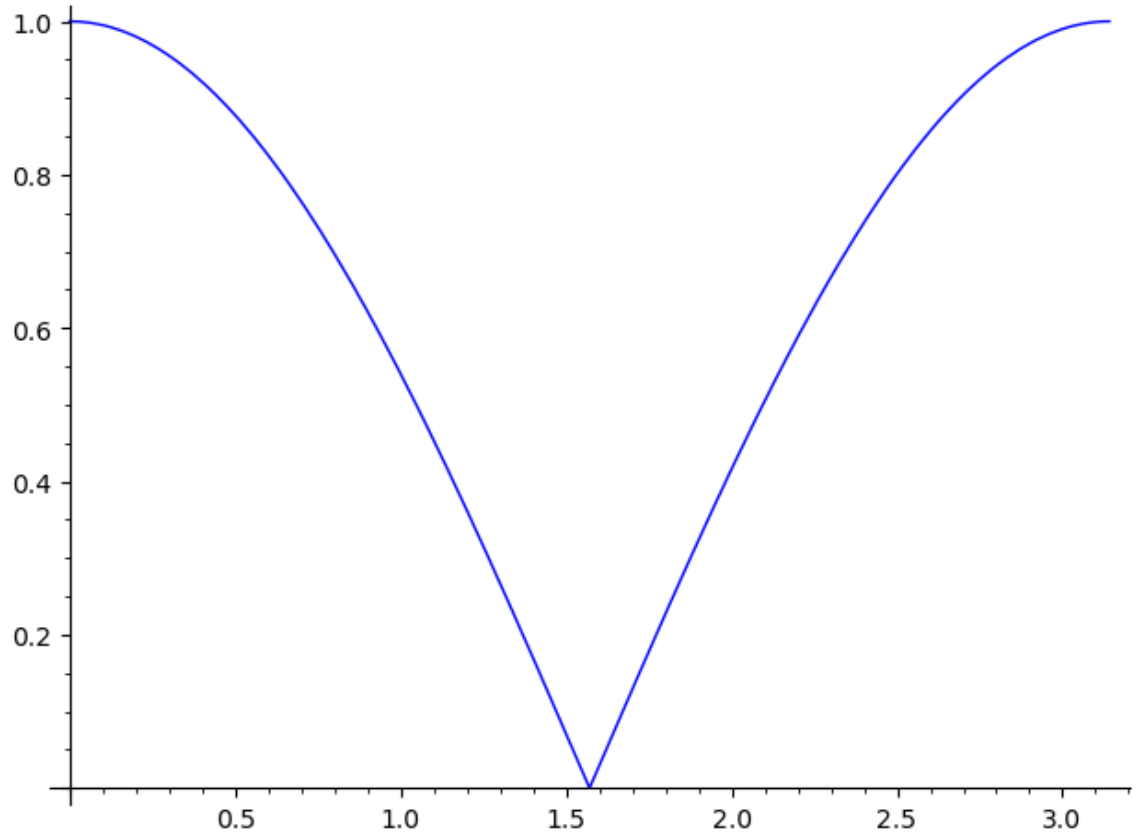
In [88]:
```
integrate(abs(cos(x)),x,0,pi)
```

```
Warning, integration of abs or sign assumes constant sign by intervals (c
orrect if the argument is real):
Check [abs(cos(sageVARx))]
Discontinuities at zeroes of cos(sageVARx) were not checked
```

Out[88]:
```
2
```

In [89]:
```
plot(abs(cos(x)),x,0,pi)
```

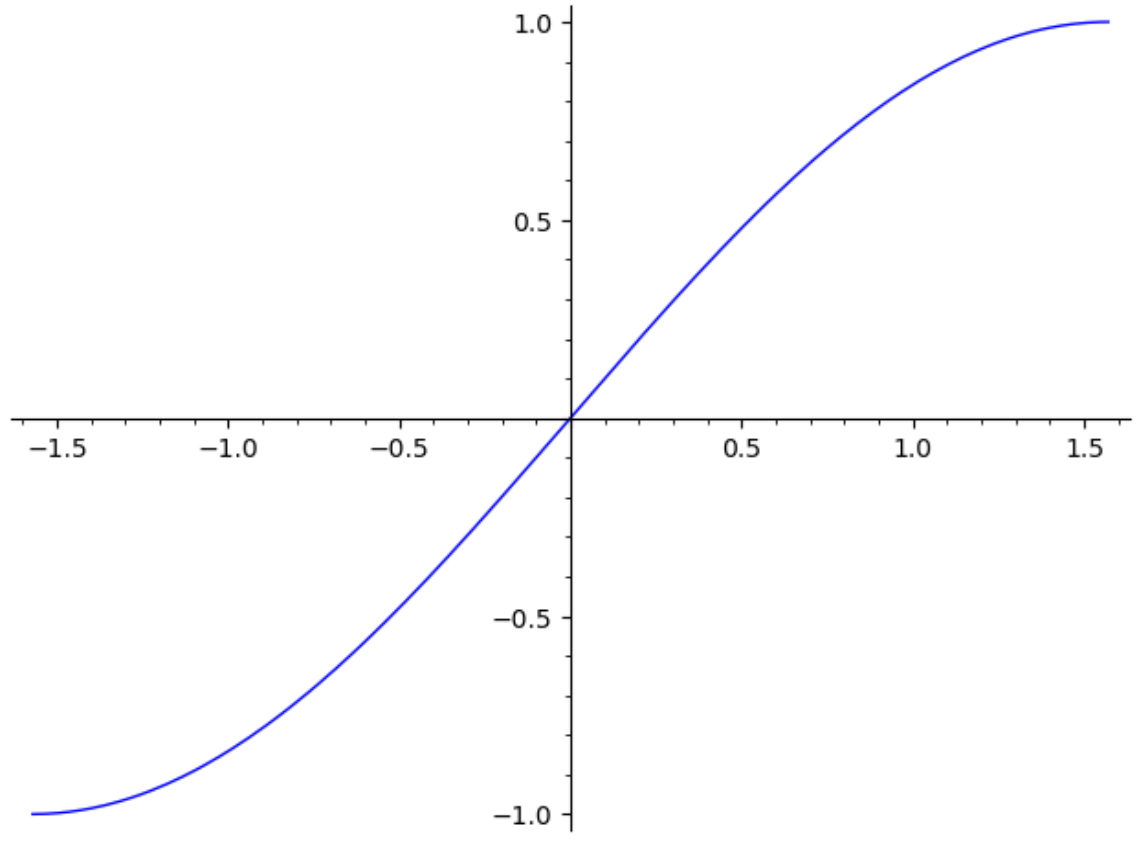
Out[89]:



Ganz konsistent ist es immer noch nicht.

In [90]:
```
plot(sin(x),x,-pi/2,pi/2)
```

Out[90]:



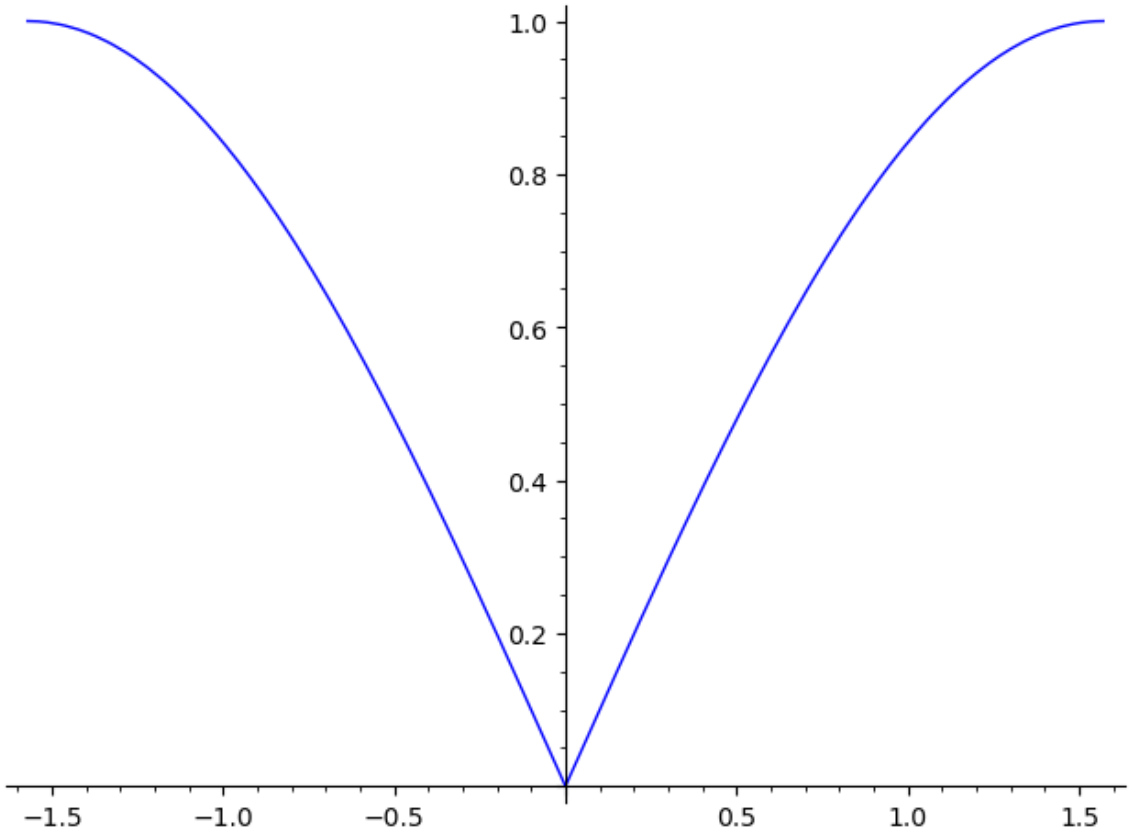
In [91]:
```
integrate(sin(x),x,-pi/2,pi/2)
```

Out[91]:
```
0
```

$|\sin(x)|=\sqrt{\sin(x)^2}$?

In [92]:
```
plot(sqrt(sin(x)^2),x,-pi/2,pi/2)
```

Out[92]:



In [93]:
```
integrate(sqrt(sin(x)^2),x,-pi/2,pi/2)
```

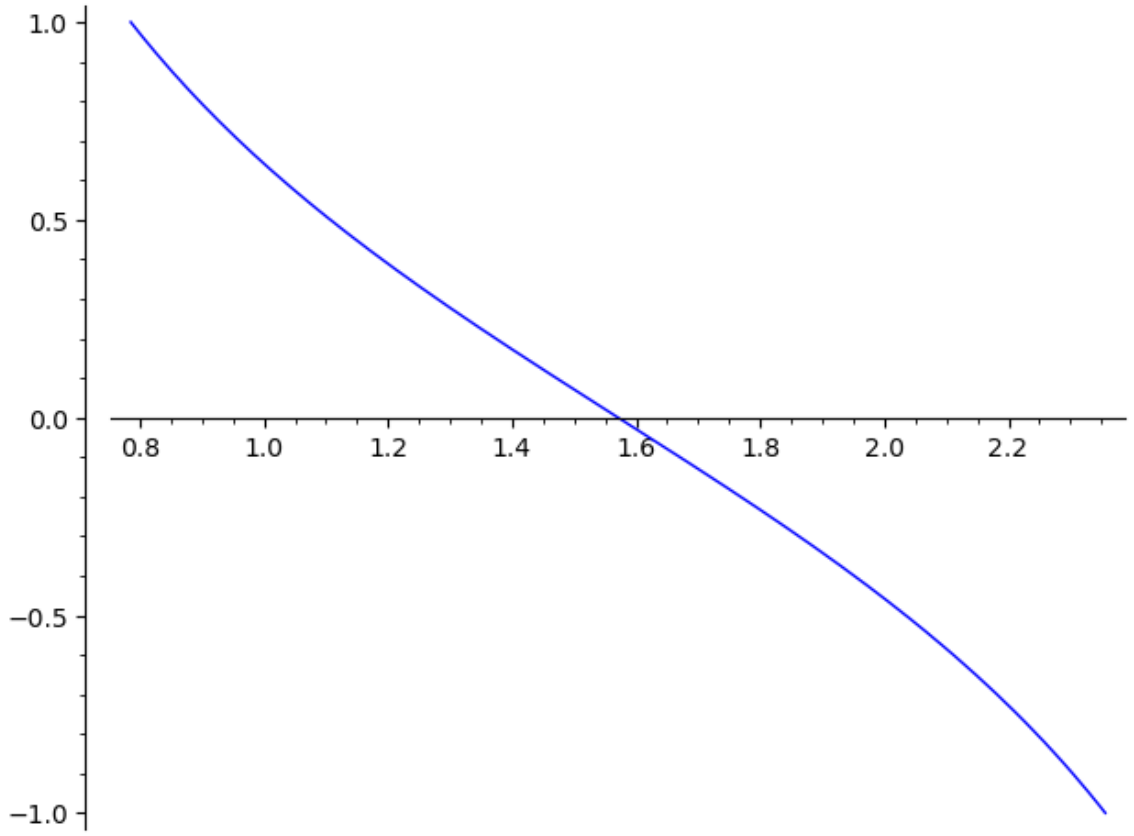
```
Warning, integration of abs or sign assumes constant sign by intervals (c
orrect if the argument is real):
Check [abs(sin(sageVARx))]
Discontinuities at zeroes of sin(sageVARx) were not checked
```

Out[93]:
2

Das gleiche mit $\cot(x)$:

In [94]:
```
plot(cot(x),x,pi/4,3*pi/4)
```

Out[94]:



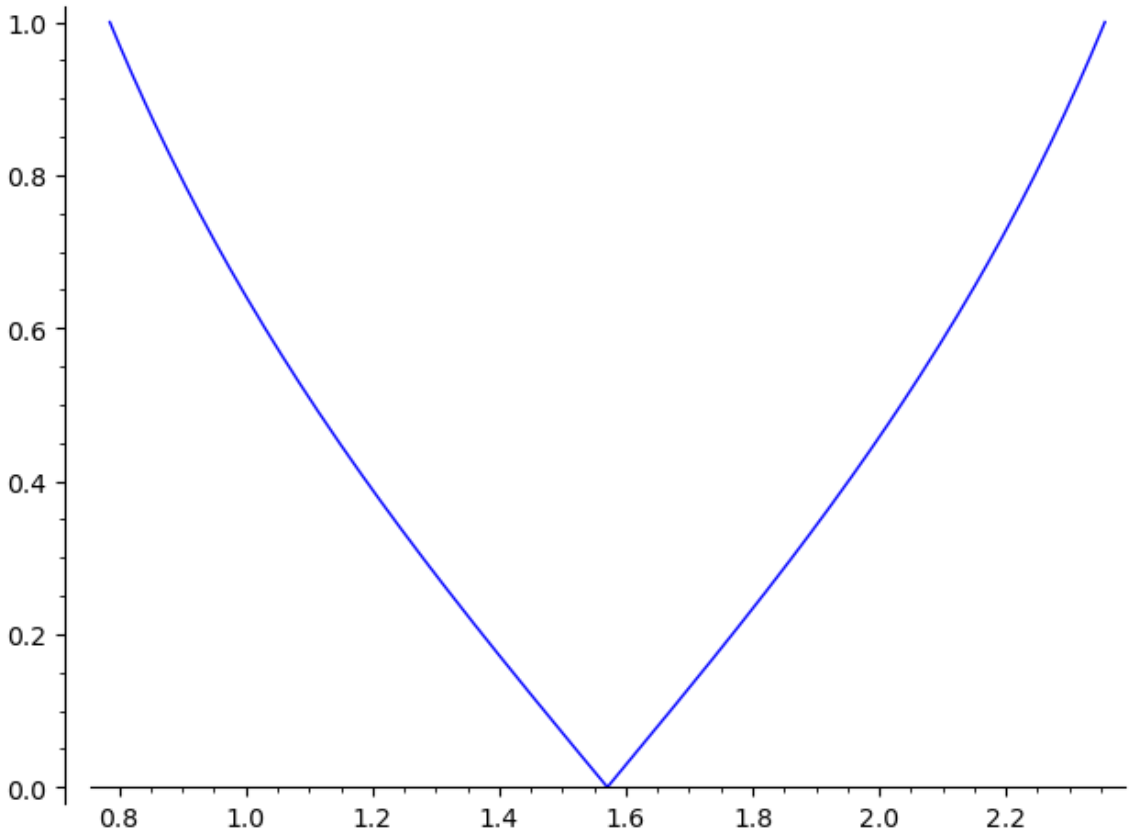
In [95]:
```
integrate(cot(x),x,pi/4,3*pi/4)
```

Out[95]:
```
0
```

$|\cot(x)|=\sqrt{\cot(x)^2}$?

In [96]:
```
plot(sqrt(cot(x)^2),x,pi/4,3*pi/4)
```

Out[96]:



In [97]:
```
integrate(sqrt(cot(x)^2),x,pi/4,3*pi/4)
```

Out[97]:
```
0
```

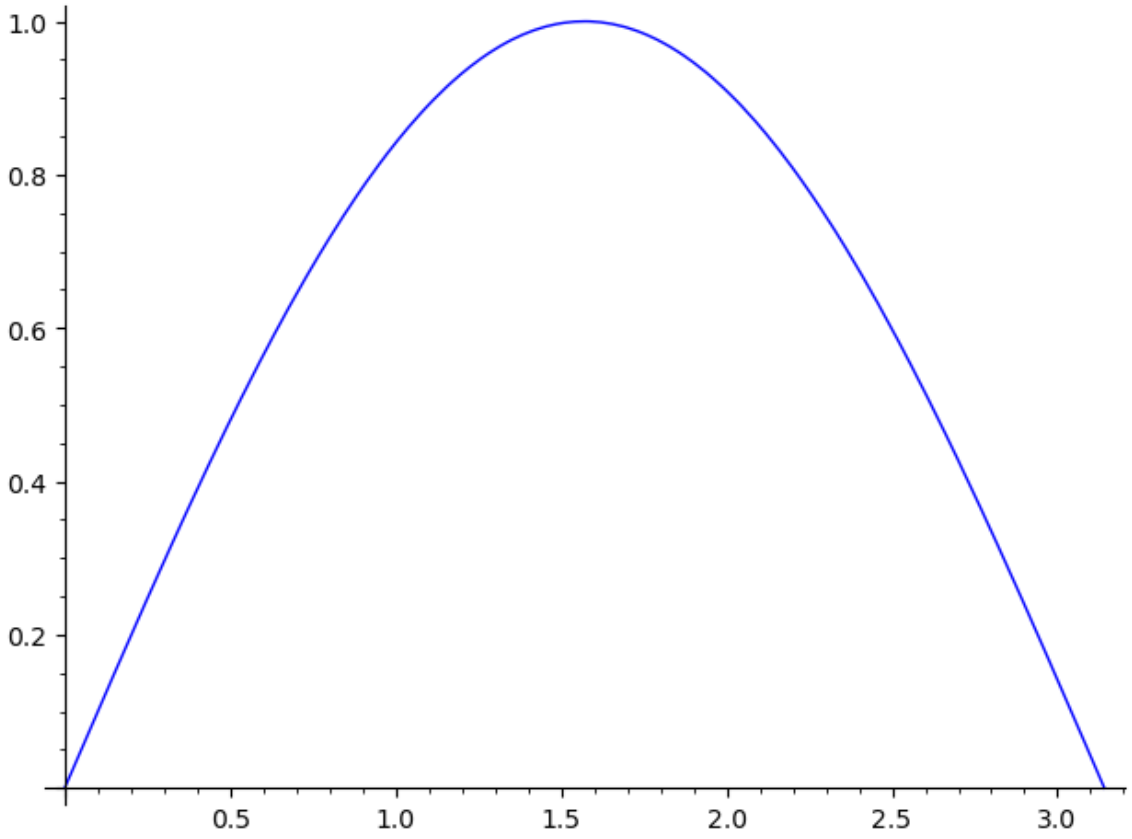
Alle bisher verwendeten Variablen:

In [98]:
```
show_identifiers()
```

Out[98]:
```
['D',
 'In',
 'L',
 'L2',
 'Out',
 'S',
 'S1',
 'S2',
 'T',
 'a',
 'b',
 'c',
 'checkbox',
 'color_selector',
 'exit',
 'f',
 'g',
 'get_ipython',
 'i',
 'input_box',
 'input_grid',
 'interact',
 'l',
 'l2',
 'l3',
 'n',
 'ncube',
 'nsquare',
 'one',
 'quit',
 'range_slider',
 'selector',
 'slider',
 'squares',
 'text_control',
 'three',
 'two',
 'u',
 'v',
 'y']
```

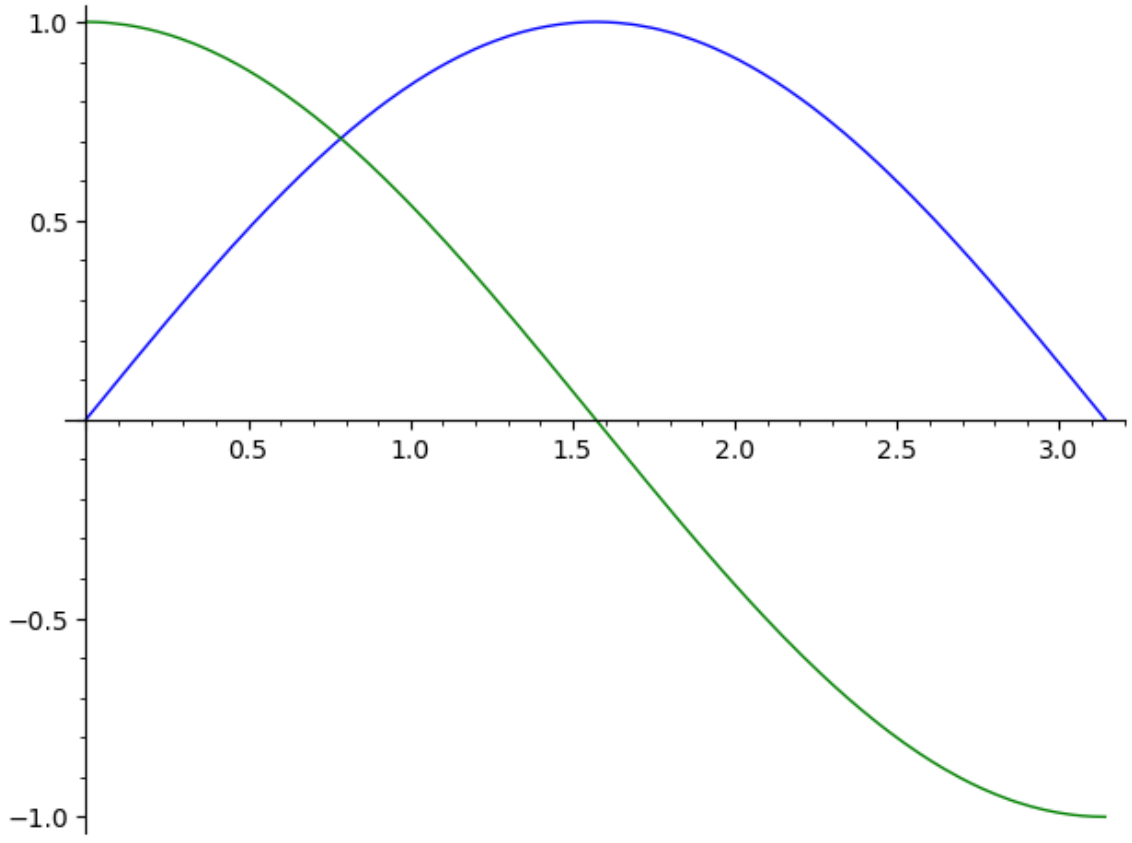
Einfache Grafik

In [99]:
```
G = plot(sin(x),0,pi)
G.show()
```



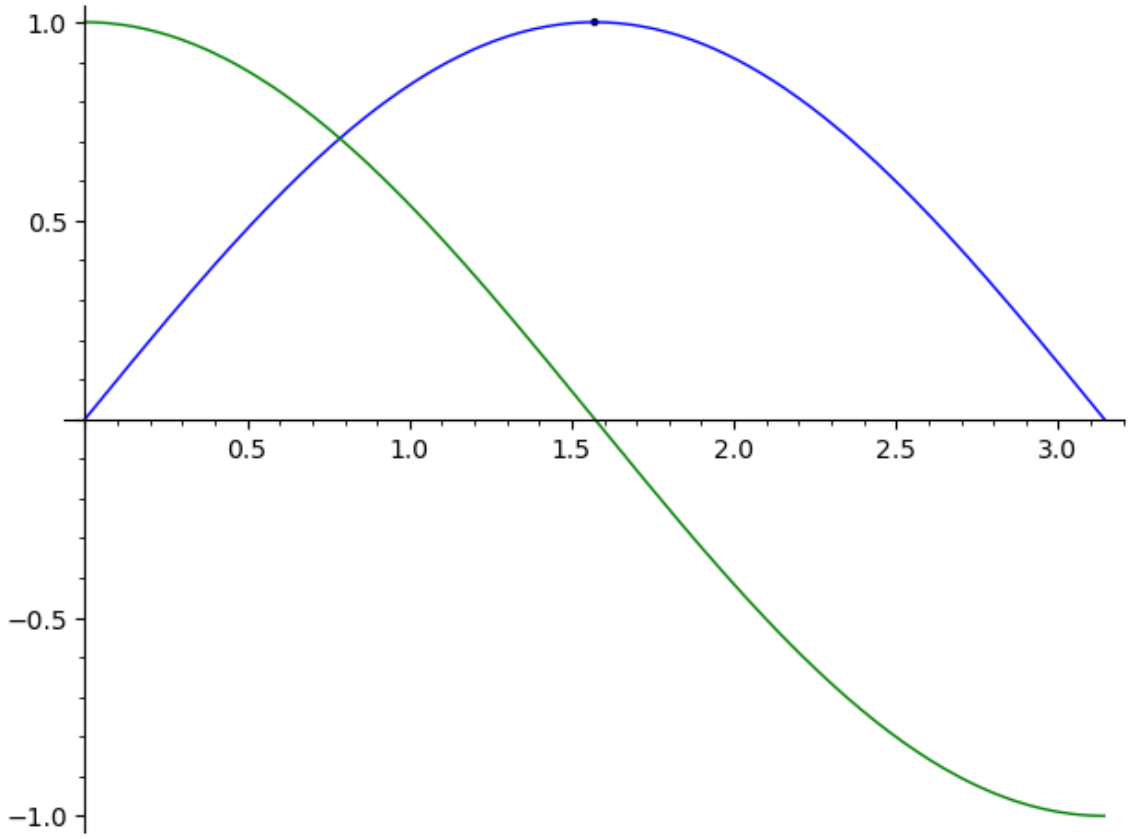
In [100…

```
G += plot(cos(x),0,pi,color='green')
G.show()
```



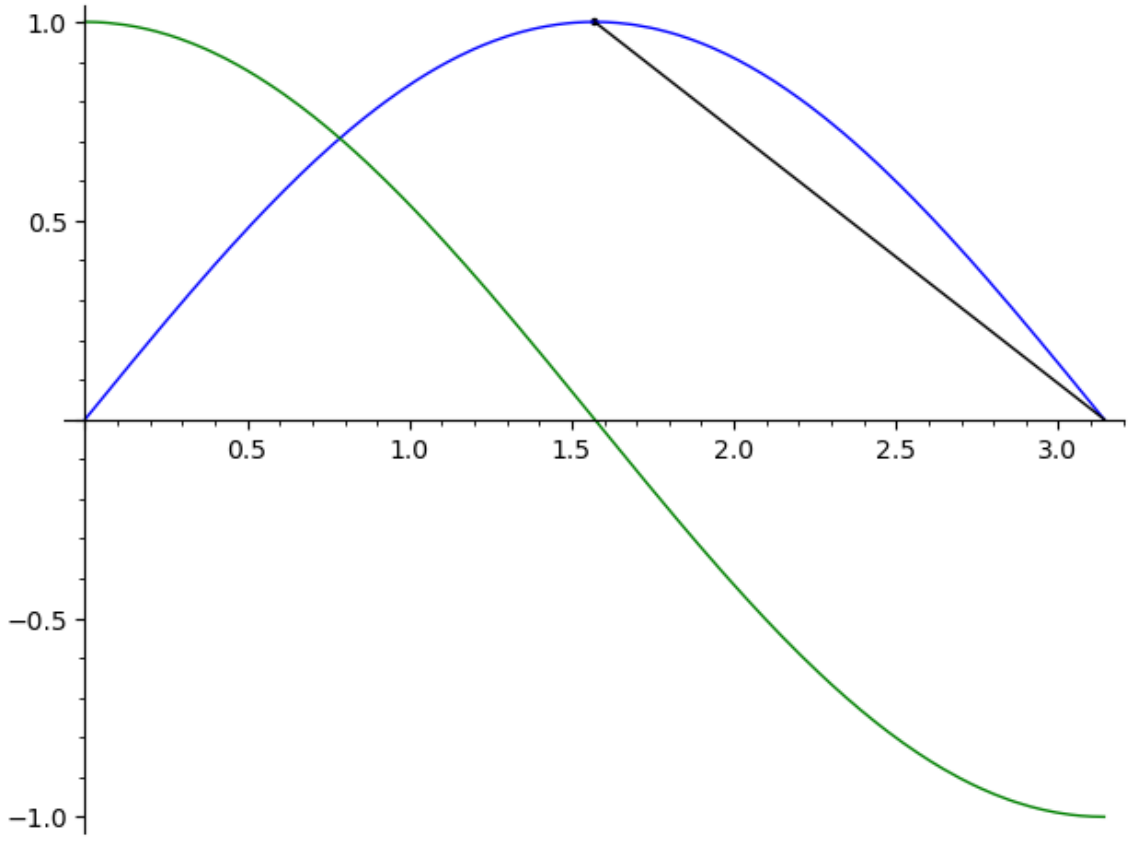
In [101…

```
G += point( (pi/2,1), color='black')
G.show()
```



In [102…

```
G += line([(pi/2,1),(pi,0)], color='black')
G.show()
```



Lösung von Gleichungen

Polynomgleichungen können bis zum Grad 4 explizit gelöst werden.

In [103…
```
solve(x^2-2 == 0, x)
```

Out[103]:
```
[x == -sqrt(2), x == sqrt(2)]
```

Allerdings sind die sogenannten *Formeln von Cardano* recht unhandlich:

In [104…
```
solve(x^3-x+2==0, x)
```

Out[104]:
```
[x == -1/2*(1/9*sqrt(26)*sqrt(3) - 1)^(1/3)*(I*sqrt(3) + 1) - 1/6*(-I*sq
rt(3) + 1)/(1/9*sqrt(26)*sqrt(3) - 1)^(1/3), x == -1/2*(1/9*sqrt(26)*sqr
t(3) - 1)^(1/3)*(-I*sqrt(3) + 1) - 1/6*(I*sqrt(3) + 1)/(1/9*sqrt(26)*sqr
t(3) - 1)^(1/3), x == (1/9*sqrt(26)*sqrt(3) - 1)^(1/3) + 1/3/(1/9*sqrt(2
6)*sqrt(3) - 1)^(1/3)]
```

In [105…
```
show(_[0])
```

In [108…
```
solve(x^4-x+2==0, x)
```

Out[108]:
```
[x == -1/2*sqrt(1/3)*sqrt((3*(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(2/3) +
8)/(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(1/3)) - 1/2*sqrt(-(1/18*I*sqrt(202
1)*sqrt(3) + 1/2)^(1/3) - 6*sqrt(1/3)/sqrt((3*(1/18*I*sqrt(2021)*sqrt(3)
+ 1/2)^(2/3) + 8)/(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(1/3)) - 8/3/(1/18*I
*sqrt(2021)*sqrt(3) + 1/2)^(1/3)), x == -1/2*sqrt(1/3)*sqrt((3*(1/18*I*s
qrt(2021)*sqrt(3) + 1/2)^(2/3) + 8)/(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^
(1/3)) + 1/2*sqrt(-(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(1/3) - 6*sqrt
(1/3)/sqrt((3*(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(2/3) + 8)/(1/18*I*sqrt
(2021)*sqrt(3) + 1/2)^(1/3)) - 8/3/(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^
(1/3)), x == 1/2*sqrt(1/3)*sqrt((3*(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^
(2/3) + 8)/(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(1/3)) - 1/2*sqrt(-(1/18*I*
sqrt(2021)*sqrt(3) + 1/2)^(1/3) + 6*sqrt(1/3)/sqrt((3*(1/18*I*sqrt(202
1)*sqrt(3) + 1/2)^(2/3) + 8)/(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(1/3)) -
8/3/(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(1/3)), x == 1/2*sqrt(1/3)*sqrt((3
*(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(2/3) + 8)/(1/18*I*sqrt(2021)*sqrt(3)
+ 1/2)^(1/3)) + 1/2*sqrt(-(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(1/3) + 6*sq
rt(1/3)/sqrt((3*(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^(2/3) + 8)/(1/18*I*sqr
t(2021)*sqrt(3) + 1/2)^(1/3)) - 8/3/(1/18*I*sqrt(2021)*sqrt(3) + 1/2)^
(1/3))]
```

In [109…
```
show(_[0])
```

Für Polynome von Grad größer 5 gibt es keine Lösungsformeln (**Satz von Abel-Ruffini**).

In [110…
```
solve(x^5-x+2==0, x)
```

Out[110]:
```
[0 == x^5 - x + 2]
```

Gleichungen mit transzendenten Funktionen können nur in wenigen Spezialfällen explizit gelöst werden.

In [111…
```
solve(sin(x)==0, x)
```

Out[111]:
```
[x == 0]
```

Wenn alle Lösungen gewünscht sind, muß das explizit angefordert werden.

In [112…
```
solve(sin(x)==0, x, to_poly_solve='force')
```

Out[112]:
```
[x == pi*z13380]
```

z8308 ist eine automatisch erzeugte formale Variable. Das Ergebnis ist zu interpretieren als Menge aller ganzzahligen Vielfachen von π.

Alternativ findet folgender Trick alle Lösungen.

In [113…
```
solve([sin(x)==0,sin(x)==0], [x])
```

Out[113]:
```
[[x == pi*z13419]]
```

Einfache Ungleichungen können ebenfalls gelöst werden.

In [114…
```
solve(1/(x-1)<8, x)
```

Out[114]:
```
[[x < 1], [x > (9/8)]]
```

In [115…
```
solve(abs(x)<1, x)
```

Out[115]:
```
#0: solve_rat_ineq(ineq=abs(_SAGE_VAR_x) < 1)
 [[-1 < x, x < 1]]
```

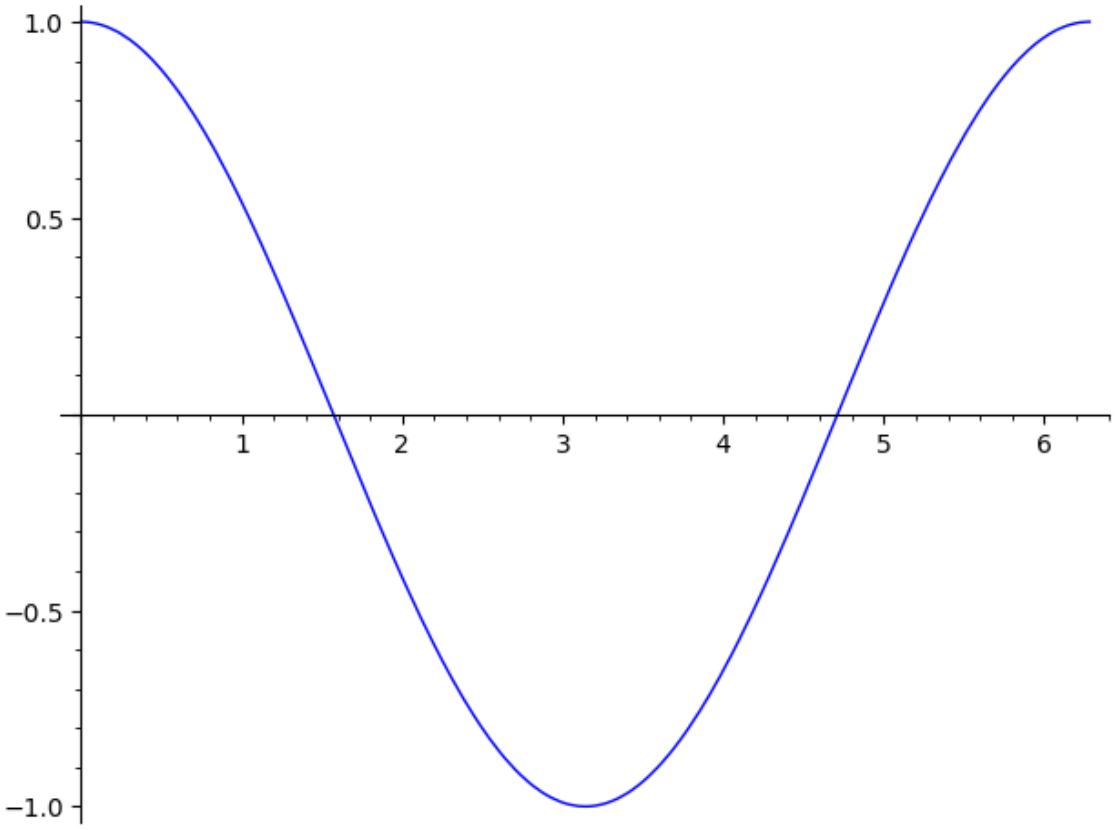
Der Algorithmus stößt aber rasch an Grenzen.

In [116…
```
solve(cos(x)<1/2, x)
```

Out[116]:
```
#0: solve_rat_ineq(ineq=cos(_SAGE_VAR_x) < 1/2)
 [[-2*cos(x) + 1 > 0]]
```

In [117…
```
plot(cos(x),x,0,2*pi)
```

Out[117]:



In [118…]:
```
solve(cos(x)<1/2, x,to_poly_solve='force')
```

Out[118]:
```
#0: solve_rat_ineq(ineq=cos(_SAGE_VAR_x) < 1/2)
 [[-2*cos(x) + 1 > 0]]
```

Polynome

Wenn man von vornherein weiß, daß man es nur mit Polynomen zu tun hat, ist es effizienter, in einem Polynomring zu arbeiten. Wir betrachten den Ring der Polynome in der einer Variable über dem Körper der rationalen Zahlen

In [119…]:
```
QQ
```

Out[119]:
```
Rational Field
```

In [120…]:
```
Px = QQ[x]
Px
```

Out[120]:
```
Univariate Polynomial Ring in x over Rational Field
```

Allerdings weiß das Symbol x nichts von diesem Ring

In [121…]:
```
parent(x)
```

Out[121]:
```
Symbolic Ring
```

Dazu muß es explizit umgewandelt werden.

In [122…
```
x = Px(x)
parent(x)
```

Out[122]:
```
Univariate Polynomial Ring in x over Rational Field
```

Achtung, die Variable x (im Sinn der Informatik) und das Symbol x sind nicht das gleiche. Im vorherigen Ausdruck ist x nach wie vor Element des `Symbolic Ring`

In [123…
```
f
```

Out[123]:
```
(x + y)^2
```

In [124…
```
parent(f)
```

Out[124]:
```
Symbolic Ring
```

Gegebenenfalls wird x automatisch in den `Symbolic Ring` zurückgeholt.

In [125…
```
cos(x)
```

Out[125]:
```
cos(x)
```

In [126…
```
parent(_)
```

Out[126]:
```
Symbolic Ring
```

Der *Zeiger* x bleibt davon aber unberührt.

In [127…
```
parent(x)
```

Out[127]:
```
Univariate Polynomial Ring in x over Rational Field
```

Man kann die obigen Operationen (Erzeugung des Polynomrings und Zuweisung der Variablen) auch in einem Schritt durchführen

In [128…
```
Py.<y> = QQ[]
Py
```

Out[128]:
```
Univariate Polynomial Ring in y over Rational Field
```

In [129…
```
parent(y)
```

Out[129]:
```
Univariate Polynomial Ring in y over Rational Field
```

Das gleiche in mehreren Variablen

In [130…
```
P2.<u,v> = QQ[]
P2
```

Out[130]:
```
Multivariate Polynomial Ring in u, v over Rational Field
```

In [131…
```
parent(u)
```

Out[131]:
```
Multivariate Polynomial Ring in u, v over Rational Field
```

In [132…]
```
parent(v)
```

Out[132]:
```
Multivariate Polynomial Ring in u, v over Rational Field
```

Dabei sind die Ringe strikt getrennt.

In [133…]
```
u+x
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Cell In [133], line 1
----> 1 u+x

File /usr/lib/python3/dist-packages/sage/structure/element.pyx:1233, in s
e.structure.element.Element.__add__ (build/cythonized/sage/structure/elem
t.c:11188)()
   1231 # Left and right are Sage elements => use coercion model
   1232 if BOTH_ARE_ELEMENT(cl):
-> 1233     return coercion_model.bin_op(left, right, add)
   1234
   1235 cdef long value

File /usr/lib/python3/dist-packages/sage/structure/coerce.pyx:1248, in sa
.structure.coerce.CoercionModel.bin_op (build/cythonized/sage/structure/c
rce.c:11885)()
   1246     # We should really include the underlying error.
   1247     # This causes so much headache.
-> 1248     raise bin_op_exception(op, x, y)
   1249
   1250 cpdef canonical_coercion(self, x, y):

TypeError: unsupported operand parent(s) for +: 'Multivariate Polynomial
ng in u, v over Rational Field' and 'Univariate Polynomial Ring in x over
ational Field'
```

Im Ring der Polynome sind immer alle Nullstellen berechenbar.

In [134…]
```
p= x^2-3
p
```

Out[134]:
```
x^2 - 3
```

In [135…]
```
parent(p)
```

Out[135]:
```
Univariate Polynomial Ring in x over Rational Field
```

Allerdings nur im zugrundeliegenden Ring.

In [136…]
```
p.roots()
```

Out[136]:
```
[]
```

Wenn in anderen Ringen gesucht werden soll, muß dies explizit angegeben werden.

In [137…]
```
p.roots(ring=RR)
```

Out[137]:
```
[(-1.73205080756888, 1), (1.73205080756888, 1)]
```

Eine Zahl heißt **algebraisch** wenn sie Nullstelle eines Polynoms mit rationalen Koeffizienten ist. Die algebraischen Zahlen bilden einen Körper und können beliebig genau approximiert werden, das wird von sage durch das Fragezeichen am Ende der Dezimalentwicklung angedeutet.

In [138…]
```
p.roots(ring=QQbar)
```

Out[138]:
```
[(-1.732050807568878?, 1), (1.732050807568878?, 1)]
```

Das Polynom kann auch in den `Symbolic Ring` eingebettet werden.

In [139…]
```
pe = SR(p)
pe
```

Out[139]:
```
x^2 - 3
```

In [140…]
```
parent(pe)
```

Out[140]:
```
Symbolic Ring
```

jetzt können die dortigen Methoden angewendet werden.

In [141…]
```
solve(pe==0,x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
Cell In [141], line 1
----> 1 solve(pe==Integer(0),x)

File /usr/lib/python3/dist-packages/sage/symbolic/relation.py:1057, in so
e(f, *args, **kwds)
   1055     x = vars[0]
   1056 elif not isinstance(x, Expression):
-> 1057     raise TypeError("%s is not a valid variable." % repr(x))
   1059 if isinstance(f, (list, tuple)) and len(f) == 1:
   1060     # f is a list with a single element
   1061     if isinstance(f[0], Expression):

TypeError: x is not a valid variable.
```

In [142…]
```
parent(x)
```

Out[142]:
```
Univariate Polynomial Ring in x over Rational Field
```

In [143…]
```
restore('x')
```

In [144…]
```
solve(pe==0,x)
```

Out[144]:
```
[x == -sqrt(3), x == sqrt(3)]
```

Gleichungen in mehreren Variablen

Die Lösungsmenge wird als Liste zurückgegeben.

In [145…
```
var('a,b')
sol=solve([3*a^2+b^2==1, a+b==1],[a,b])
sol
```

Out[145]:
```
[[a == (1/2), b == (1/2)], [a == 0, b == 1]]
```

Die Lösungen können daraus auf verschiedene Arten extrahiert werden.

In [146…
```
sol[0]
```

Out[146]:
```
[a == (1/2), b == (1/2)]
```

Durch Substitution

In [147…
```
exp(a).subs(sol[0])
```

Out[147]:
```
e^(1/2)
```

dabei werden alle Vorkommen der Variablen ersetzt.

In [148…
```
(exp(a)+exp(b)).subs(sol[1])
```

Out[148]:
```
e + 1
```

Oder mit der Methode `rhs` (=right hand side):

In [149…
```
sol[0][0].rhs()
```

Out[149]:
```
1/2
```

In [150…
```
exp(sol[0][0].rhs())
```

Out[150]:
```
e^(1/2)
```

In [151…
```
exp(x).subs(x=sol[0][0].rhs())
```

Out[151]:
```
e^(1/2)
```

oder als dictionary

In [152…
```
var('a,b')
sol=solve([3*a^2+b^2==1, a+b==1],[a,b])
sol
```

Out[152]:
```
[[a == (1/2), b == (1/2)], [a == 0, b == 1]]
```

In [153…
```
var('a,b')
sol=solve([3*a^2+b^2==1, a+b==1],[a,b],solution_dict=True)
sol
```

Out[153]:
```
[{a: 1/2, b: 1/2}, {a: 0, b: 1}]
```

In [154…
```
sol[0][a]
```

Out[154]:
```
1/2
```

In [155…
```
sol[1][b]
```

Out[155]:
```
1
```

Zeitmessung

Es laufen intern zwei Uhren: die *cputime* mißt die Zeit, die im Prozessor verbracht wird, die *walltime* mißt die Zeit, die insgesamt vergeht.

In [156…
```
cputime()
```

Out[156]:
```
6.694081
```

In [157…
```
walltime()
```

Out[157]:
```
1734097347.4772635
```

Durch Auswerten der Differenz kann die für einen Vorgang verbrauchte Zeit gemessen werden.

In [158…
```
t=cputime()
factor(2^67-1)
cputime()-t
```

Out[158]:
```
0.006450000000000067
```

Eine genaueren Wert erhält man durch Mittelung über mehrere Versuche. Dabei muß der auszuführende Befehl als String übergeben werden.

In [159…
```
timeit?
```

In [160…
```
timeit( 'factor(2^67-1)' )
```

Out[160]:
```
625 loops, best of 3: 730 µs per loop
```

In [161…
```
t=cputime()
for k in [2^10..2^13]:
    factor(k)
cputime(t)
```

Out[161]:
```
0.019796999999998732
```

Zeilenumbrüche können mit \n angegeben werden.

In [162…
```
timeit( 'for k in [2^10..2^13]:\n     factor(k)')
```

Out[162]:
```
25 loops, best of 3: 17.9 ms per loop
```

In [163…
```
print("a\nb")
```

```
a
b
```

In [ ]: