2025_11_21_sage_1

November 21, 2025

Christopher Frei

frei@math.tugraz.at

Steyrergasse 30, 2. Stock

Wo bekomme ich Sage?

- Online am Jupyter-Server der Uni Graz: https://imsc.uni-graz.at/jupyter/ (Login via SSO über Uni Graz Online account)

Lokale Installation auf Ihrem eigenen Rechner:

- unter Linux: wahrscheinlich in der Paketverwaltung Ihrer Distribution vorhanden
- Herunterladen auf www.sagemath.org (den dortigen Installationsanweisungen folgen)

Kommerzielle Onlineversion:

- www.cocalc.com

Die für die Übungen relevante Version ist Sage 10.7.

Was ist Sage?

Ein Computeralgebrasystem basierend auf der Programmiersprache Python. Also im Wesentlichen Python mit vielen zusätzlichen Mathematikfunktionen. Es kennt viele mathematische Objekte und Strukturen und kann mit diesen numerisch und exakt rechnen. Sage ist Open-Source. Viele seiner Funktionen sind aus anderen Open-Source-Mathematikprogrammen übernommen (z.B. Maxima, GAP).

Wir verwenden Sage über das Jupyter-Notebookinterface, das neben Python auch andere Programmiersprachen unterstützt. Man kann seinen Code hier auf mehrere Zellen aufteilen und diese getrennt voneinander ausführen.

Einfache Rechnungen

Zelle ausführen: Shift + Enter

[1]:
```
1+1
```

[1]:
```
2
```

[2]:
```
2*3
```

[2]:
```
6
```

[3]:
```
2/3
```

[3]: 2/3

[4]:
```
2/3+1/6
```

[4]: 5/6

[5]:
```
_
```

[5]: 5/6

[6]:
```
_^2
```

[6]: 25/36

[7]:
```
14//3     # Ganzzahlige Division
```

[7]: 4

[8]:
```
14 % 3    # Rest bei Division
```

[8]: 2

[9]:
```
14.quo_rem(3)    # Division mit Rest als Sage-Methode
```

[9]: (4, 2)

[10]:
```
(14/3).n()     # Numerisch auswerten
```

[10]: 4.66666666666667

Variablen

In ausgeführten Zellen belegte Variablen bleiben im Speicher und können auch in späteren Zellen verwendet werden. Relevant ist die Reihenfolge, in der die Zellen ausgeführt werden, nicht ihre Reihenfolge im Notebook.

Damit hier keine Fehlermeldung kommt, muß die übernächste Zelle vor der nächsten ausgeführt werden.

[13]:
```
a^3
```

[13]: 64

[12]:
```
a=4        # Hier wird kein Output angezeigt. Will man den Wert von a nach der␣
↪Zuweisung sehen, muss man das extra angeben.
```

[14]:
```
a=1/2
a
```

[14]:
```
1/2
```

[15]:
```
a=1/2; a
```

[15]:
```
1/2
```

[16]:
```
show(a)    # schönere Ausgabe. Sie können den Bruch rechts anklicken und auch
 ↪den LaTeX-Code erhalten.
```

$$\frac{1}{2}$$

[17]:
```
b=sqrt(2)  # Irrationale Zahlen werden auch als symbolische Ausdrücke
 ↪gespeichert.
b
```

[17]:
```
sqrt(2)
```

[18]:
```
show(b)
```

$$\sqrt{2}$$

[19]:
```
b.n()       # numerische Auswertung wie zuvor
```

[19]:
```
1.41421356237310
```

[20]:
```
b^2
```

[20]:
```
2
```

Klassen und Objekte

Python, und daher auch Sage, sind objektorientiert: alles ist ein Objekt, und jedes Objekt gehört einer Klasse an. Die Klasse bestimmt, was man mit einem Objekt machen kann. Sie stellt Methoden bereit, das sind Funktionen, die mit ihren Objekten ausgeführt werden können.

[21]:
```
a
```

[21]:
```
1/2
```

[22]:
```
type(a)     # die Klasse des in der Variable a gespeicherten Objekts.
```

[22]:
```
<class 'sage.rings.rational.Rational'>
```

In sage definierte mathematische Objekte sind üblicherweise Elemente gewisser mathematischer Struktren. Diese sind ähnlich zu den Klassen in Python, aber anschaulicher.

[23]:
```
parent(a)   # die Struktur, deren Element a ist; hier ist a eine rationale
 ↪Zahl, und daher Element der Struktur Rational Field
```

[23]:
```
Rational Field
```

[24]:
```
show(parent(a))
```

Q

[25]:
```
Rational Field        # Die Ausgabe der vorletzten Berechnung, "Rational␣
↪Field", ist nur eine Beschreibung des Objekts parent(a). Solche␣
↪Beschreibungen sind nicht dazu gedacht, auf die Objekte zuzugreifen, daher␣
↪kommt hier eine Fehlermeldung
```

```
  Cell In [25], line 1
    Rational Field        # Die Ausgabe der vorletzten Berechnung, "Rational␣
↪Field", ist nur eine Beschreibung des Objekts parent(a). Solche Beschreibunge␣
↪sind nicht dazu gedacht, auf die Objekte zuzugreifen, daher kommt hier eine␣
↪Fehlermeldung
              ^
SyntaxError: invalid syntax
```

[26]:
```
f=parent(a)    # Hier wird parent(a) in einer neuen Variable f gespeichert und␣
↪wir können damit weiterarbeiten
f
```

[26]:
```
Rational Field
```

[27]:
```
type(f)
```

[27]:
```
<class 'sage.rings.rational_field.RationalField_with_category'>
```

Viele mathematische Objekte werden als symbolische Ausdrücke gespeichert.

[28]:
```
b=sqrt(2)
type(b)
```

[28]:
```
<class 'sage.symbolic.expression.Expression'>
```

[29]:
```
parent(b)
```

[29]:
```
Symbolic Ring
```

[30]:
```
b^2
```

[30]:
```
2
```

[31]:
```
parent(_)        # das Ergebnis 2 wird hier immer noch als symbolischer␣
↪Ausdruck gespeichert, also nicht automatisch wieder als ganze Zahl gesehen
```

[31]:
```
Symbolic Ring
```

[32]:
```
parent(2)        # ganze Zahlen sind Elemente der Struktur Integer Ring
```

[32]: Integer Ring

Im nächsten Beispiel wird die Zahl 2, ein Element des Integer Ring, mit der Zahl 1/2, ein Element des Rational Field, multipliziert. Dazu wird intern zuerst 2 als Element des Rational Field betrachtet, und dann die für Rational Field definierte Multiplikation verwendet. Solche impliziten Typumwandlungen geschehen oft automatisch, wenn es eine kanonische Umwandlung gibt.

[21]:
```
2*(1/2)
```

[21]: 1

[22]:
```
type(_)     # Das Ergebnis, 1, wurde nicht automatisch wieder zu einem Element␣
 ↪des Integer Ring umgewandelt, auch wenn das hier möglich wäre.
```

[22]: <class 'sage.rings.rational.Rational'>

[23]:
```
ZZ(2*(1/2))     # Hier geben wir durch ZZ(...) explizit an, dass das Ergebnis␣
 ↪als Element des Integer Ring gesehen werden soll.
```

[23]: 1

[24]:
```
type(_)
```

[24]: <class 'sage.rings.integer.Integer'>

[25]:
```
ZZ(1/2)     # Natürlich kann nicht jede rationale Zahl in eine ganze Zahl␣
 ↪umgewandelt werden.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 1
----> 1 ZZ(Integer(1)/Integer(2))     # Natürlich kann nicht jede rationale Zahl␣
 ↪in eine ganze Zahl umgewandelt werden.

File /opt/sagemath/sage-10.5/src/sage/structure/parent.pyx:901, in sage.
 ↪structure.parent.Parent.__call__()
    899 if mor is not None:
    900     if no_extra_args:
--> 901         return mor._call_(x)
    902     else:
    903         return mor._call_with_args(x, args, kwds)

File /opt/sagemath/sage-10.5/src/sage/rings/rational.pyx:4208, in sage.rings.
 ↪rational.Q_to_Z._call_()
   4206 """
   4207 if not mpz_cmp_si(mpq_denref((<Rational>x).value), 1) == 0:
-> 4208     raise TypeError("no conversion of this rational to integer")
   4209 cdef Integer n = Integer.__new__(Integer)
```

```
   4210 n.set_from_mpz(mpq_numref((<Rational>x).value))

TypeError: no conversion of this rational to integer
```

[26]:
```
sqrt(-1)
```

[26]:
```
I
```

[27]:
```
I    # Die Variable I ist als Wert imaginäre Einheit vordefniert
```

[27]:
```
I
```

[28]:
```
I^2
```

[28]:
```
-1
```

[29]:
```
I=5    # I wird nicht vor Umdefinitionen geschützt!
```

[30]:
```
sqrt(-1)   # Hier ist der Output trotzdem noch I, denn es handelt sich nur um␣
↪eine Beschreibung des Wertes von sqrt(-1). Das ist unabhängig vom Wert der␣
↪Variable I, den wir oben geändert haben.
```

[30]:
```
I
```

[31]:
```
I^2
```

[31]:
```
25
```

[32]:
```
restore('I')     # So kann man geänderte vordefinierte Variablen wieder auf␣
↪ihren ursprünglichen Wert zurücksetzen.
```

[33]:
```
I^2
```

[33]:
```
-1
```

[34]:
```
pi     # Die Variable pi ist als die Kreiszahl vordefiniert
```

[34]:
```
pi
```

[35]:
```
parent(pi)
```

[35]:
```
Symbolic Ring
```

[36]:
```
pi.n(digits=5000)    # numerische Auswertung muß explizit gefordert werden.␣
↪Hier: auf 5000 Ziffern genau
```

[36]:
```
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745
```

028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526193118817101000313783875288658753320838142061717766914730359825349042875546873115956286388235378759375195778185778053217122680661300192787661119590921642019893809525720106548586327886593615338182796823030195203530185296899577362259941389124972177528347913151557485724245415069595082953311686172785588907509838175463746493931925506040092770167113900984882401285836160356370766010471018194295559619894676783744944825537977472684710404753464620804668425906949129331367702898915210475216205696602405803815019351125338243003558764024749647326391419927260426992279678235478163600934172164121992458631503028618297455570674983850549458858692699569092721079750930295532116534498720275596023648066549911988183479775356636980742654252786255181841757467289097777279380008164706001614524919217321721477235014144197356854816136115735255213347574184946843852332390739414333454776241686251898356948556209921922218427255025425688767179049460165346680498862723279178608578438382796797668145410095388378636095068006422512520511739298489608412848862694560424196528502221066118630674427862203919494504712371378696095636437191728746776465757396241389086583264599581339047802759009946576407895126946839835259570982582262052248940772671947826848260147699090264013639443745530506820349625245174939965143142980919065925093722169646151570985838741059788595977297549893016175392846813826868386894277415599185592524595395943104997252468084598727364469584865383673622262609912460805124388439045124413654976278079771569143599770012961608944169486855584840635422072225828488648158456028506016842739452267467678895252138522549954666727823986456596116354886230577456498035593634568174324112515076069479451096596094025228879710893145669136867228748940560101503308617928680920874760917824938589009714909675985261365549781893129784821682998948722658804857564014270477555132379641451523746234364542858444795265867821051141354735739523113427166102135969536231442952484937187110145765403590279934403742007310578539062198387447808478489683321445713868751943506430218453191048481005370614680674919278191197939952061419663428754440643745123718192179998391015919561814675142691239748940907186494231961567945208095146550225231603881930142093762137855956638937787083039069792077346722182562599661501421503068038447734549202605414665925201497442850732518666002132434088190710486331734649651453905796268561005508106658796998163574736384052571459102897064140110971206280439039759515677157700420337869936007230558763176359421873125147120532928191826186125867321579198414848829164470609575270695722091756711672291098169091528017350671274858322287183520935396572512108357915136988209144421006751033467110314126711136990865851639831501970165151168517143765761835155650884909989859982387345528331635507647918535893226185489632132933089857064204675259070915481416549859461637180270981994309924488957571282890592323326097299712084433573265489382391193259746366730583604142813883032038249037589852437441702913276561809377344403070746921120191302033038019762110110044929321516084244485963766983895228684783123552658213144957685726243

```
3441893039686426243410773226978028073189154411010446823252716201052652272111660396666557309254711055785376346682065310989652691862056476931257058635662018558100729360659876486117910453348850346113657686753249441668039626579787718556084552965412665408530614344431858676975145661406800700237877659134401712749470420562230538994561314071127000407854733269939081454664645880797270826683063432858785698305235808933065757406795457163775254202114955761581400250126228594130216471550979259230990796547376125517656751357517829666454779174501129961489030463994713296210734043751895735961458901938971311179042978285647503203198691514028708085990480109412147221317947647772622414254854540332157185306142288137585043063321751829798662237172159160771669254748738986654949450114654062843366393790039769265672146385306736096571209180763832716641627488880078692560290228472104031721186082041900042296617119637792133757511495950156604963186294726547364252308177036751590673502350728354056704038674351362222477158915049530984448933309634087807693259939780541934144737744184263129860809988687413260472
```

[37]:
```
# Sei o ein Objekt der Klasse k, und f eine Methode dieser Klasse. Um f für das␣
↪Objekt o auszuführen, gibt es zwei äquivalente Möglichkeiten:
#
#   k.f(o)      ....  Sollte man nicht verwenden, da unnötig kompliziert
#   o.f()       ....  zu bevorzugen
```

[38]:
```
type(pi)
```

[38]:
```
<class 'sage.symbolic.expression.Expression'>
```

[39]:
```
Expression.n(pi,digits=40)        # so nicht machen!
```

[39]:
```
3.1415926535897932384626433832795028841971
```

[40]:
```
pi.n(digits=40)                   # hier wird genau dieselbe Funktion␣
↪Expression.n() auf pi angewandt wie oben (mit zweitem Argument digits=40).␣
↪Der Aufruf ist so übersichtlicher.
```

[40]:
```
3.1415926535897932384626433832795028841971
```

[41]:
```
numerical_approx(pi, digits=40)              # hier wird die globale Funktion␣
↪numerical_approx() auf pi angewandt (wieder mit zweitem Argument digits=40).␣
↪Intern führt das zur selben Funktion wie pi.n().
```

[41]:
```
3.1415926535897932384626433832795028841971
```

Methoden und Funktionen mit demselben Namen machen nicht immer dasselbe, selbst wenn der Output gleich aussieht.

[42]:
```
mod(42,9)        # Globale Funktion mod(), liefert die Restklasse von 42 modulo␣
↪9.
```

[42]:
```
6
```

[43]:
```
parent(_)
```

[43]:
```
Ring of integers modulo 9
```

[44]:
```
42.mod(9)        # Die Funktion Integer.mod(), liefert einen Repräsentanten in␣
↪ZZ der Restklasse von 42 modulo 9.
```

[44]:
```
6
```

[45]:
```
parent(_)
```

[45]:
```
Integer Ring
```

Dokumentation und Hilfe

Auto-Verfollständigung von Methodennamen mit TAB

Methodenname ohne Klammern mit ? liefert Dokumentation

Methodenname ohne Klammern mit ?? liefert Dokumentation + Quellcode

[ ]:
```
pi.    # Druck auf Tabulatortaste liefert hier Liste aller Methoden der Klasse␣
↪von pi, also von Expression.
```

[ ]:
```
pi.n   # Druck auf TAB hier liefert alle Methoden, die min n beginnen
```

[66]:
```
pi.n?
```

```
Docstring:
   Alias for "numerical_approx()".

   EXAMPLES:

      sage: (2/3).n()
      0.666666666666667
Init docstring: Initialize self.  See help(type(self)) for accurate signature.
File:           /opt/sagemath/sage-10.1/src/sage/structure/element.pyx
Type:           builtin_function_or_method
```

[67]:
```
pi.numerical_approx?
```

```
Docstring:
   Return a numerical approximation of "self" with "prec" bits (or
   decimal "digits") of precision.

   No guarantee is made about the accuracy of the result.

   INPUT:

   * "prec" -- precision in bits
```

```
    * "digits" -- precision in decimal digits (only used if "prec" is
      not given)

    * "algorithm" -- which algorithm to use to compute this
      approximation

    If neither "prec" nor "digits" is given, the default precision is
    53 bits (roughly 16 digits).

    EXAMPLES:

       sage: sin(x).subs(x=5).n()
       -0.958924274663138
       sage: sin(x).subs(x=5).n(100)
       -0.95892427466313846889315440616
       sage: sin(x).subs(x=5).n(digits=50)
       -0.95892427466313846889315440615599397335246154396460
       sage: zeta(x).subs(x=2).numerical_approx(digits=50)
       1.6449340668482264364724151666460251892189499012068

       sage: cos(3).numerical_approx(200)
       -0.98999249660044545727157279473126130239367909661558832881409
       sage: numerical_approx(cos(3),200)
       -0.98999249660044545727157279473126130239367909661558832881409
       sage: numerical_approx(cos(3), digits=10)
       -0.9899924966
       sage: (i + 1).numerical_approx(32)
       1.00000000 + 1.00000000*I
       sage: (pi + e + sqrt(2)).numerical_approx(100)
       7.2740880444219335226246195788
Init docstring: Initialize self.  See help(type(self)) for accurate signature.
File:           /opt/sagemath/sage-10.1/src/sage/symbolic/expression.pyx
Type:           builtin_function_or_method
```

[68]:
```
pi.numerical_approx??
```

```
Docstring:
    Return a numerical approximation of "self" with "prec" bits (or
    decimal "digits") of precision.

    No guarantee is made about the accuracy of the result.

    INPUT:

    * "prec" -- precision in bits
```

```
    * "digits" -- precision in decimal digits (only used if "prec" is
      not given)

    * "algorithm" -- which algorithm to use to compute this
      approximation

    If neither "prec" nor "digits" is given, the default precision is
    53 bits (roughly 16 digits).

    EXAMPLES:

       sage: sin(x).subs(x=5).n()
       -0.958924274663138
       sage: sin(x).subs(x=5).n(100)
       -0.95892427466313846889315440616
       sage: sin(x).subs(x=5).n(digits=50)
       -0.95892427466313846889315440615599397335246154396460
       sage: zeta(x).subs(x=2).numerical_approx(digits=50)
       1.6449340668482264364724151666460251892189499012068

       sage: cos(3).numerical_approx(200)
       -0.98999249660044545727157279473126130239367909661558832881409
       sage: numerical_approx(cos(3),200)
       -0.98999249660044545727157279473126130239367909661558832881409
       sage: numerical_approx(cos(3), digits=10)
       -0.9899924966
       sage: (i + 1).numerical_approx(32)
       1.00000000 + 1.00000000*I
       sage: (pi + e + sqrt(2)).numerical_approx(100)
       7.2740880444219335226246195788
Source:
    def numerical_approx(self, prec=None, digits=None, algorithm=None):
        """
        Return a numerical approximation of ``self`` with ``prec`` bits
        (or decimal ``digits``) of precision.

        No guarantee is made about the accuracy of the result.

        INPUT:

        - ``prec`` -- precision in bits

        - ``digits`` -- precision in decimal digits (only used if
          ``prec`` is not given)

        - ``algorithm`` -- which algorithm to use to compute this
          approximation
```

```
If neither ``prec`` nor ``digits`` is given, the default
precision is 53 bits (roughly 16 digits).

EXAMPLES::

    sage: sin(x).subs(x=5).n()
    -0.958924274663138
    sage: sin(x).subs(x=5).n(100)
    -0.95892427466313846889315440616
    sage: sin(x).subs(x=5).n(digits=50)
    -0.95892427466313846889315440615599397335246154396460
    sage: zeta(x).subs(x=2).numerical_approx(digits=50)
    1.6449340668482264364724151666460251892189499012068

    sage: cos(3).numerical_approx(200)
    -0.98999249660044545727157279473126130239367909661558832881409
    sage: numerical_approx(cos(3),200)
    -0.98999249660044545727157279473126130239367909661558832881409
    sage: numerical_approx(cos(3), digits=10)
    -0.9899924966
    sage: (i + 1).numerical_approx(32)
    1.00000000 + 1.00000000*I
    sage: (pi + e + sqrt(2)).numerical_approx(100)
    7.2740880444219335226246195788

TESTS:

We test the evaluation of different infinities available in Pynac::

    sage: t = x - oo; t
    -Infinity
    sage: t.n()
    -infinity
    sage: t = x + oo; t
    +Infinity
    sage: t.n()
    +infinity
    sage: t = x - unsigned_infinity; t
    Infinity
    sage: t.n()
    Traceback (most recent call last):
    …
    ValueError: can only convert signed infinity to RR

Some expressions cannot be evaluated numerically::

    sage: n(sin(x))
    Traceback (most recent call last):
```

```
    …
    TypeError: cannot evaluate symbolic expression numerically
    sage: a = var('a')
    sage: (x^2 + 2*x + 2).subs(x=a).n()
    Traceback (most recent call last):
    …
    TypeError: cannot evaluate symbolic expression numerically

Make sure we've rounded up log(10,2) enough to guarantee
sufficient precision (:trac:`10164`)::

    sage: ks = 4*10**5, 10**6
    sage: all(len(str(e.n(digits=k)))-1 >= k for k in ks)
    True

Symbolic sums with definite endpoints are expanded (:trac:`9424`)::

    sage: (k,n) = var('k,n')
    sage: f(n) = sum(abs(-k*k+n),k,1,n)
    sage: ex = f(n=8); ex
    sum(abs(-k^2 + 8), k, 1, 8)
    sage: ex.n()
    162.000000000000
    sage: (ex+1).n()
    163.000000000000

Check if :trac:`24418` is fixed::

    sage: numerical_approx(2^(450232897/4888643760))
    1.06591892580915
"""
if prec is None:
    prec = digits_to_bits(digits)

from sage.symbolic.expression_conversions import ExpressionTreeWalker

class DefiniteSumExpander(ExpressionTreeWalker):
    def composition(self, ex, operator):
        if hasattr(operator, 'name') and operator.name() == 'sum' and (
                is_a_numeric((<Expression>ex.operands()[2])._gobj)
                and is_a_numeric((<Expression>ex.operands()[3])._gobj)):
            from sage.calculus.calculus import symbolic_sum
            return symbolic_sum(*(ex.operands()))
        return super().composition(ex, operator)

s = DefiniteSumExpander(self)
cdef Expression x = self._parent(s())
from sage.rings.real_mpfr import RealField
```

```
            R = RealField(prec)
            kwds = {'parent': R, 'algorithm': algorithm}
            try:
                x = x._convert(kwds)
            except TypeError:  # numerical approximation for real number failed
                pass           # try again with complex
                kwds['parent'] = R.complex_field()
                x = x._convert(kwds)

            # we have to consider constants as well, since infinity is a constant
            # in pynac
            if is_a_numeric(x._gobj):
                res = py_object_from_numeric(x._gobj)
            elif is_a_constant(x._gobj):
                res = x.pyobject()
            else:
                raise TypeError("cannot evaluate symbolic expression numerically")

            # Important -- the  we get might not be a valid output for␣
     ↪numerical_approx in
            # the case when one gets infinity.
            if isinstance(res, AnInfinity):
                return res.n(prec=prec, digits=digits)
            return res
    File:      /opt/sagemath/sage-10.1/src/sage/symbolic/expression.pyx
    Type:      builtin_function_or_method
```

Gleitkommazahlen

[47]:
```
R=RealField()       # Körper der reellen Zahlen, in Sage implementiert als␣
 ↪Gleitkommazahlen mit 53 bit (15 Dezimalstellen) Präzision
R
```

[47]:
```
Real Field with 53 bits of precision
```

[48]:
```
R(1)
```

[48]:
```
1.00000000000000
```

[49]:
```
R100=RealField(100)      # Körper der reellen Zahlen mit höherer Präzision
R100
```

[49]:
```
Real Field with 100 bits of precision
```

[50]:
```
R100(1)
```

[50]:
```
1.0000000000000000000000000000
```

[51]:
```
R100(1)/R100(3)
```

[51]:
```
0.33333333333333333333333333333
```

[52]:
```
R100(1)/3       # Die ganze Zahl 3 wird für die Operation automatisch in ein␣
↪Element von R100 umgewandelt
```

[52]:
```
0.33333333333333333333333333333
```

[55]:
```
R100(1)/R(3)     # Hier wird R100(1) automatisch in eine relle Zahl mit mit␣
↪niedrigerer Präzision umgewandelt, das Ergebnis hat auch niedrigere Präzision
```

[55]:
```
0.333333333333333
```

[56]:
```
_.str(base=2)
```

[56]:
```
'0.01010101010101010101010101010101010101010101010101010101'
```

[57]:
```
R100(R100(1)/R(3))     # Hier wird das Ergebnis auf scheinbar beliebige Art␣
↪wieder zu einer Zahl mit Präzision 100 gemacht. Wo kommen die zusätzlichen␣
↪Stellen her?
```

[57]:
```
0.33333333333333331482961625625
```

[58]:
```
_.str(base=2)       # Antwort: Intern wird alles in Binärdarstellung␣
↪gespeichert, und in dieser Darstellung wurden einfach Nullen angehängt. Auf␣
↪Dezimal umgerechnet ergibt das die obige Zahl.
```

[58]:
```
'0.01010101010101010101010101010101010101010101010101010100000000000000000000000000
0000000000000000000000000'
```

Listen

[94]:
```
l=[1,2,3,x,"abc",ZZ,2]         # Listen werden mit eckigen Klammern angegeben␣
↪und können beliebige Objekte enthalten.
l
```

[94]:
```
[1, 2, 3, x, 'abc', Integer Ring, 2]
```

[95]:
```
type(l)
```

[95]:
```
<class 'list'>
```

[96]:
```
l[1]       # Zugriff auf Elemente via Indizes. Achtung: Indizes starten bei 0
```

[96]:
```
2
```

[97]:
```
l[0]
```

[97]: 1

[98]:
```
l[-1]     # das letzte Element
```

[98]: 2

[99]:
```
l[-2]     # das vorletzte Element
```

[99]: Integer Ring

[100]:
```
len(l)     # Länge der Liste
```

[100]: 7

[101]:
```
l.append(5)     # Ein Element am Ende anhängen. Hier wird die Liste direkt␣
↪modifiziert, und nicht etwa eine neue Kopie mit dem zusätzlichen Element␣
↪erstellt!
```

[102]:
```
l
```

[102]: [1, 2, 3, x, 'abc', Integer Ring, 2, 5]

[103]:
```
del l[2]       # Das Element an dritter Stelle (Index 2) löschen. Auch hier␣
↪wird die Liste direkt modifiziert.
```

[104]:
```
l
```

[104]: [1, 2, x, 'abc', Integer Ring, 2, 5]

[105]:
```
l.remove(x)     # das Element x aus der Liste löschen
```

[106]:
```
l
```

[106]: [1, 2, 'abc', Integer Ring, 2, 5]

[107]:
```
l.remove(2)      # nur das erste passende Element wird gelöscht, die zweite 2␣
↪bleibt in der Liste.
```

[108]:
```
l
```

[108]: [1, 'abc', Integer Ring, 2, 5]

[109]:
```
l.insert(1,"u")
```

[110]:
```
l
```

[110]: [1, 'u', 'abc', Integer Ring, 2, 5]

Komplexe und ganze Zahlen

[111]:
```
a=pi+I
a
```

[111]: pi + I

[112]:
```
type(a)
```

[112]: <class 'sage.symbolic.expression.Expression'>

[113]:
```
a.n()
```

[113]: 3.14159265358979 + 1.00000000000000*I

[114]:
```
CC     # bereits vordefiniert als die komplexen Zahlen mit 53 bits Präzision
```

[114]: Complex Field with 53 bits of precision

[115]:
```
RR
```

[115]: Real Field with 53 bits of precision

[116]:
```
ComplexField()      # Die Funktion ComplexField() liefert ein Objekt wie CC:␣
↪die komplexen Zahlen mit 53 bits Präzision
```

[116]: Complex Field with 53 bits of precision

[117]:
```
CC(a)
```

[117]: 3.14159265358979 + 1.00000000000000*I

[118]:
```
real(a)    # Realteil symbolisch
```

[118]: pi

[119]:
```
imag(a)    # Imaginärteil symbolisch
```

[119]: 1

[120]:
```
b=CC(a)    # wandle um zu komplexer Fließkommazahl
b
```

[120]: 3.14159265358979 + 1.00000000000000*I

[121]:
```
real(b)
```

[121]: 3.14159265358979

[122]:
```
imag(b)
```

[122]:
```
1.00000000000000
```

[123]:
```
parent(1)      # Ganze Zahlen sind in Sage Elemente der Struktur Integer Ring
```

[123]:
```
Integer Ring
```

[124]:
```
gcd(77,135)    # ggT
```

[124]:
```
1
```

[125]:
```
xgcd(77,335)    # erweiterter Euklidischer Algorithmus:  1 = ggT(77, 335) =␣
↪-87*77 + 20*335
```

[125]:
```
(1, -87, 20)
```

[126]:
```
-87*77+20*335
```

[126]:
```
1
```

[127]:
```
lcm(77,335)       # kgV
```

[127]:
```
25795
```

[128]:
```
77*335            # kgV = Produkt, da ggT=1
```

[128]:
```
25795
```

[129]:
```
factor(335)       # Zerlegung in Primfaktoren
```

[129]:
```
5 * 67
```

[130]:
```
77.is_prime()     # Primzahltest
```

[130]:
```
False
```

[131]:
```
a=77
```

Viele weitere Methoden der Klasse Integer über a.<TAB> verfügbar.

[132]:
```
a.divisors?
```

```
Docstring:
   Return the list of all positive integer divisors of this integer,
   sorted in increasing order.

   EXAMPLES:

      sage: (-3).divisors()
      [1, 3]
      sage: 6.divisors()
```

```
        [1, 2, 3, 6]
        sage: 28.divisors()
        [1, 2, 4, 7, 14, 28]
        sage: (2^5).divisors()
        [1, 2, 4, 8, 16, 32]
        sage: 100.divisors()
        [1, 2, 4, 5, 10, 20, 25, 50, 100]
        sage: 1.divisors()
        [1]
        sage: 0.divisors()
        Traceback (most recent call last):
        …
        ValueError: n must be nonzero
        sage: (2^3 * 3^2 * 17).divisors()
        [1, 2, 3, 4, 6, 8, 9, 12, 17, 18, 24, 34, 36, 51, 68, 72,
        102, 136, 153, 204, 306, 408, 612, 1224]
        sage: a = odd_part(factorial(31))
        sage: v = a.divisors()
        sage: len(v)
        172800
        sage: prod(e + 1 for p, e in factor(a))
        172800
        sage: all(t.divides(a) for t in v)
        True

        sage: n = 2^551 - 1
        sage: L = n.divisors()
        sage: len(L)
        256
        sage: L[-1] == n
        True

    Note:

      If one first computes all the divisors and then sorts it, the
      sorting step can easily dominate the runtime. Note, however, that
      (nonnegative) multiplication on the left preserves relative
      order. One can leverage this fact to keep the list in order as
      one computes it using a process similar to that of the merge sort
      algorithm.
Init docstring: Initialize self.  See help(type(self)) for accurate signature.
File:           /opt/sagemath/sage-10.5/src/sage/rings/integer.pyx
Type:           builtin_function_or_method
```

[133]:
```
a.divisors()  # Liste der Teiler von a
```

[133]:
```
[1, 7, 11, 77]
```

[134]:
```
a.next_prime()
```

[134]: 79

[135]:
```
79.next_prime()
```

[135]: 83