Symbolic Calculus

Release 10.6

The Sage Development Team

Apr 01, 2025

CONTENTS

Calculus is done using symbolic expressions which consist of symbols and numeric objects linked by operators (functions).

Note

While polynomial manipulation can be done with expressions, it is more efficient to use polynomial ring elements

CHAPTER

ONE

USING CALCULUS

CHAPTER

TWO

INTERNAL FUNCTIONALITY SUPPORTING CALCULUS

2.1 Symbolic Expressions

RELATIONAL EXPRESSIONS:

We create a relational expression:

```
sage: x = var('x')
sage: eqn = (x-1)^2 <= x^2 - 2*x + 3
sage: eqn.subs(x == 5)
16 <= 18
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eqn = (x-Integer(1))**Integer(2) <= x**Integer(2) - Integer(2)*x + Integer(3)
>>> eqn.subs(x == Integer(5))
16 <= 18
```

Notice that squaring the relation squares both sides.

```
sage: eqn^2
(x - 1)^4 <= (x^2 - 2*x + 3)^2
sage: eqn.expand()
x^2 - 2*x + 1 <= x^2 - 2*x + 3
```

```
>>> from sage.all import *
>>> eqn**Integer(2)
(x - 1)^4 <= (x^2 - 2*x + 3)^2
>>> eqn.expand()
x^2 - 2*x + 1 <= x^2 - 2*x + 3
```

This can transform a true relation into a false one:

```
sage: eqn = SR(-5) < SR(-3); eqn
-5 < -3
sage: bool(eqn)
True
sage: eqn^2
25 < 9
sage: bool(eqn^2)
False
```

```
>>> from sage.all import *
>>> eqn = SR(-Integer(5)) < SR(-Integer(3)); eqn
-5 < -3
>>> bool(eqn)
True
>>> eqn**Integer(2)
25 < 9
>>> bool(eqn**Integer(2))
False
```

We can do arithmetic with relations:

```
sage: e = x+1 <= x-2
sage: e + 2
x + 3 <= x
sage: e - 1
x <= x - 3
sage: e*(-1)
-x - 1 <= -x + 2
sage: (-2)*e
-2*x - 2 <= -2*x + 4
sage: e*5
5*x + 5 <= 5*x - 10
sage: e/5
1/5*x + 1/5 <= 1/5*x - 2/5
sage: 5/e
5/(x + 1) <= 5/(x - 2)
sage: e/(-2)
-1/2*x - 1/2 <= -1/2*x + 1
sage: -2/e
-2/(x + 1) <= -2/(x - 2)
```

```
>>> from sage.all import *
>>> e = x+Integer(1) <= x-Integer(2)
>>> e + Integer(2)
x + 3 <= x
>>> e - Integer(1)
x <= x - 3
>>> e*(-Integer(1))
-x - 1 <= -x + 2
>>> (-Integer(2))*e
-2*x - 2 <= -2*x + 4
>>> e*Integer(5)
5*x + 5 <= 5*x - 10
>>> e/Integer(5)
1/5*x + 1/5 <= 1/5*x - 2/5
>>> Integer(5)/e
5/(x + 1) <= 5/(x - 2)
>>> e/(-Integer(2))
-1/2*x - 1/2 <= -1/2*x + 1
>>> -Integer(2)/e
-2/(x + 1) <= -2/(x - 2)
```

We can even add together two relations, as long as the operators are the same:

```
sage: (x^3 + x <= x - 17)  + (-x <= x - 10)
x^3 <= 2*x - 27
```

```
>>> from sage.all import *
>>> (x**Integer(3) + x <= x - Integer(17))  + (-x <= x - Integer(10))
x^3 <= 2*x - 27
```

Here they are not:

```
sage: (x^3 + x <= x - 17)  + (-x >= x - 10)
Traceback (most recent call last):
...
TypeError: incompatible relations
```

```
>>> from sage.all import *
>>> (x**Integer(3) + x <= x - Integer(17))  + (-x >= x - Integer(10))
Traceback (most recent call last):
...
TypeError: incompatible relations
```

ARBITRARY SAGE ELEMENTS:

You can work symbolically with any Sage data type. This can lead to nonsense if the data type is strange, e.g., an element of a finite field (at present).

We mix Singular variables with symbolic variables:

```
sage: R.<u,v> = QQ[]
sage: var('a,b,c')
(a, b, c)
sage: expand((u + v + a + b + c)^2)
```

(continues on next page)

(continued from previous page)

```
a^2 + 2*a*b + b^2 + 2*a*c + 2*b*c + c^2 + 2*a*u + 2*b*u + 2*c*u + u^2 + 2*a*v + 2*b*v␣
↪+ 2*c*v + 2*u*v + v^2
```

```
>>> from sage.all import *
>>> R = QQ['u, v']; (u, v,) = R._first_ngens(2)
>>> var('a,b,c')
(a, b, c)
>>> expand((u + v + a + b + c)**Integer(2))
a^2 + 2*a*b + b^2 + 2*a*c + 2*b*c + c^2 + 2*a*u + 2*b*u + 2*c*u + u^2 + 2*a*v + 2*b*v␣
↪+ 2*c*v + 2*u*v + v^2
```

class sage.symbolic.expression.**E**

Bases: *Expression*

Dummy class to represent base of the natural logarithm.

The base of the natural logarithm `e` is not a constant in GiNaC/Sage. It is represented by `exp(1)`.

This class provides a dummy object that behaves well under addition, multiplication, etc. and on exponentiation calls the function `exp`.

EXAMPLES:

The constant defined at the top level is just `exp(1)`:

```
sage: e.operator()
exp
sage: e.operands()
[1]
```

```
>>> from sage.all import *
>>> e.operator()
exp
>>> e.operands()
[1]
```

Arithmetic works:

```
sage: e + 2
e + 2
sage: 2 + e
e + 2
sage: 2*e
2*e
sage: e*2
2*e
sage: x*e
x*e
sage: var('a,b')
(a, b)
sage: t = e^(a+b); t
e^(a + b)
sage: t.operands()
[a + b]
```

```
>>> from sage.all import *
>>> e + Integer(2)
e + 2
>>> Integer(2) + e
e + 2
>>> Integer(2)*e
2*e
>>> e*Integer(2)
2*e
>>> x*e
x*e
>>> var('a,b')
(a, b)
>>> t = e**(a+b); t
e^(a + b)
>>> t.operands()
[a + b]
```

Numeric evaluation, conversion to other systems, and pickling works as expected. Note that these are properties of the `exp()` function, not this class:

```
sage: RR(e)
2.71828182845905
sage: R = RealField(200); R
Real Field with 200 bits of precision
sage: R(e)
2.7182818284590452353602874713526624977572470936999595749670
sage: em = 1 + e^(1-e); em
e^(-e + 1) + 1
sage: R(em)
1.1793740787340171819619895873183164984596816017589156131574
sage: maxima(e).float()
2.718281828459045
sage: t = mathematica(e)                # optional - mathematica
sage: t                                 # optional - mathematica
E
sage: float(t)                          # optional - mathematica
2.718281828459045...

sage: loads(dumps(e))
e

sage: float(e)
2.718281828459045...
sage: e.__float__()
2.718281828459045...
sage: e._mpfr_(RealField(100))
2.7182818284590452353602874714
sage: e._real_double_(RDF)    # abs tol 5e-16
2.718281828459045
sage: import sympy                                                          #␣
↪needs sympy
sage: sympy.E == e  # indirect doctest                                      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sympy
True
```

```
>>> from sage.all import *
>>> RR(e)
2.71828182845905
>>> R = RealField(Integer(200)); R
Real Field with 200 bits of precision
>>> R(e)
2.7182818284590452353602874713526624977572470936999595749670
>>> em = Integer(1) + e**(Integer(1)-e); em
e^(-e + 1) + 1
>>> R(em)
1.1793740787340171819619895873183164984596816017589156131574
>>> maxima(e).float()
2.718281828459045
>>> t = mathematica(e)                # optional - mathematica
>>> t                                 # optional - mathematica
E
>>> float(t)                          # optional - mathematica
2.718281828459045...

>>> loads(dumps(e))
e

>>> float(e)
2.718281828459045...
>>> e.__float__()
2.718281828459045...
>>> e._mpfr_(RealField(Integer(100)))
2.7182818284590452353602874714
>>> e._real_double_(RDF)    # abs tol 5e-16
2.718281828459045
>>> import sympy                                                      # needs␣
↪sympy
>>> sympy.E == e  # indirect doctest                                  # needs␣
↪sympy
True
```

class sage.symbolic.expression.**Expression**

Bases: `Expression`

Nearly all expressions are created by calling new_Expression_from_*, but we need to make sure this at least does not leave self._gobj uninitialized and segfault.

Order(*hold=False*)

Return the order of the expression, as in big oh notation.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: n = var('n')
sage: t = (17*n^3).Order(); t
```

(continues on next page)

(continued from previous page)

```
Order(n^3)
sage: t.derivative(n)
Order(n^2)
```

```
>>> from sage.all import *
>>> n = var('n')
>>> t = (Integer(17)*n**Integer(3)).Order(); t
Order(n^3)
>>> t.derivative(n)
Order(n^2)
```

To prevent automatic evaluation use the `hold` argument:

```
sage: (17*n^3).Order(hold=True)
Order(17*n^3)
```

```
>>> from sage.all import *
>>> (Integer(17)*n**Integer(3)).Order(hold=True)
Order(17*n^3)
```

WZ_certificate(*n*, *k*)

Return the Wilf-Zeilberger certificate for this hypergeometric summand in `n`, `k`.

To prove the identity $\sum_k F(n,k) = \text{const}$ it suffices to show that $F(n+1,k) - F(n,k) = G(n,k+1) - G(n,k)$, with $G = RF$ and R the WZ certificate.

EXAMPLES:

To show that $\sum_k \binom{n}{k} = 2^n$ do:

```
sage: _ = var('k n')
sage: F(n,k) = binomial(n,k) / 2^n
sage: c = F(n,k).WZ_certificate(n,k); c
1/2*k/(k - n - 1)
sage: G(n,k) = c * F(n,k); G
(n, k) |--> 1/2*k*binomial(n, k)/(2^n*(k - n - 1))
sage: (F(n+1,k) - F(n,k) - G(n,k+1) + G(n,k)).simplify_full()
0
```

```
>>> from sage.all import *
>>> _ = var('k n')
>>> __tmp__=var("n,k"); F = symbolic_expression(binomial(n,k) /␣
↪Integer(2)**n).function(n,k)
>>> c = F(n,k).WZ_certificate(n,k); c
1/2*k/(k - n - 1)
>>> __tmp__=var("n,k"); G = symbolic_expression(c * F(n,k)).function(n,k); G
(n, k) |--> 1/2*k*binomial(n, k)/(2^n*(k - n - 1))
>>> (F(n+Integer(1),k) - F(n,k) - G(n,k+Integer(1)) + G(n,k)).simplify_full()
0
```

abs(*hold=False*)

Return the absolute value of this expression.

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: (x+y).abs()
abs(x + y)
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> (x+y).abs()
abs(x + y)
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(-5).abs(hold=True)
abs(-5)
```

```
>>> from sage.all import *
>>> SR(-Integer(5)).abs(hold=True)
abs(-5)
```

To then evaluate again, we use *unhold()*:

```
sage: a = SR(-5).abs(hold=True); a.unhold()
5
```

```
>>> from sage.all import *
>>> a = SR(-Integer(5)).abs(hold=True); a.unhold()
5
```

add(*hold=False*, **args*)

Return the sum of the current expression and the given arguments.

To prevent automatic evaluation use the `hold` argument.

EXAMPLES:

```
sage: x.add(x)
2*x
sage: x.add(x, hold=True)
x + x
sage: x.add(x, (2+x), hold=True)
(x + 2) + x + x
sage: x.add(x, (2+x), x, hold=True)
(x + 2) + x + x + x
sage: x.add(x, (2+x), x, 2*x, hold=True)
(x + 2) + 2*x + x + x + x
```

```
>>> from sage.all import *
>>> x.add(x)
2*x
>>> x.add(x, hold=True)
x + x
>>> x.add(x, (Integer(2)+x), hold=True)
```

(continues on next page)

(continued from previous page)

```
(x + 2) + x + x
>>> x.add(x, (Integer(2)+x), x, hold=True)
(x + 2) + x + x + x
>>> x.add(x, (Integer(2)+x), x, Integer(2)*x, hold=True)
(x + 2) + 2*x + x + x + x
```

To then evaluate again, we use *unhold()*:

```
sage: a = x.add(x, hold=True); a.unhold()
2*x
```

```
>>> from sage.all import *
>>> a = x.add(x, hold=True); a.unhold()
2*x
```

add_to_both_sides(*x*)

Return a relation obtained by adding `x` to both sides of this relation.

EXAMPLES:

```
sage: var('x y z')
(x, y, z)
sage: eqn = x^2 + y^2 + z^2 <= 1
sage: eqn.add_to_both_sides(-z^2)
x^2 + y^2 <= -z^2 + 1
sage: eqn.add_to_both_sides(I)
x^2 + y^2 + z^2 + I <= (I + 1)
```

```
>>> from sage.all import *
>>> var('x y z')
(x, y, z)
>>> eqn = x**Integer(2) + y**Integer(2) + z**Integer(2) <= Integer(1)
>>> eqn.add_to_both_sides(-z**Integer(2))
x^2 + y^2 <= -z^2 + 1
>>> eqn.add_to_both_sides(I)
x^2 + y^2 + z^2 + I <= (I + 1)
```

arccos(*hold=False*)

Return the arc cosine of `self`.

EXAMPLES:

```
sage: x.arccos()
arccos(x)
sage: SR(1).arccos()
0
sage: SR(1/2).arccos()
1/3*pi
sage: SR(0.4).arccos()
1.15927948072741
sage: plot(lambda x: SR(x).arccos(), -1,1)                                  #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> x.arccos()
arccos(x)
>>> SR(Integer(1)).arccos()
0
>>> SR(Integer(1)/Integer(2)).arccos()
1/3*pi
>>> SR(RealNumber('0.4')).arccos()
1.15927948072741
>>> plot(lambda x: SR(x).arccos(), -Integer(1),Integer(1))                    ␣
↪                # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(1).arccos(hold=True)
arccos(1)
```

```
>>> from sage.all import *
>>> SR(Integer(1)).arccos(hold=True)
arccos(1)
```

This also works using functional notation:

```
sage: arccos(1, hold=True)
arccos(1)
sage: arccos(1)
0
```

```
>>> from sage.all import *
>>> arccos(Integer(1), hold=True)
arccos(1)
>>> arccos(Integer(1))
0
```

To then evaluate again, we use *unhold()*:

```
sage: a = SR(1).arccos(hold=True); a.unhold()
0
```

```
>>> from sage.all import *
>>> a = SR(Integer(1)).arccos(hold=True); a.unhold()
0
```

arccosh(*hold=False*)

Return the inverse hyperbolic cosine of `self`.

EXAMPLES:

```
sage: x.arccosh()
arccosh(x)
sage: SR(0).arccosh()
1/2*I*pi
```

(continues on next page)

(continued from previous page)

```
sage: SR(1/2).arccosh()
arccosh(1/2)
sage: SR(CDF(1/2)).arccosh() #  rel tol 1e-15
1.0471975511965976*I
sage: z = maxima('acosh(0.5)')
sage: z.real(), z.imag()  # abs tol 1e-15
(0.0, 1.047197551196598)
```

```
>>> from sage.all import *
>>> x.arccosh()
arccosh(x)
>>> SR(Integer(0)).arccosh()
1/2*I*pi
>>> SR(Integer(1)/Integer(2)).arccosh()
arccosh(1/2)
>>> SR(CDF(Integer(1)/Integer(2))).arccosh() #  rel tol 1e-15
1.0471975511965976*I
>>> z = maxima('acosh(0.5)')
>>> z.real(), z.imag()  # abs tol 1e-15
(0.0, 1.047197551196598)
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(-1).arccosh()
I*pi
sage: SR(-1).arccosh(hold=True)
arccosh(-1)
```

```
>>> from sage.all import *
>>> SR(-Integer(1)).arccosh()
I*pi
>>> SR(-Integer(1)).arccosh(hold=True)
arccosh(-1)
```

This also works using functional notation:

```
sage: arccosh(-1,hold=True)
arccosh(-1)
sage: arccosh(-1)
I*pi
```

```
>>> from sage.all import *
>>> arccosh(-Integer(1),hold=True)
arccosh(-1)
>>> arccosh(-Integer(1))
I*pi
```

To then evaluate again, we use *unhold()*:

```
sage: a = SR(-1).arccosh(hold=True); a.unhold()
I*pi
```

```
>>> from sage.all import *
>>> a = SR(-Integer(1)).arccosh(hold=True); a.unhold()
I*pi
```

arcsin(*hold=False*)

Return the arcsin of x, i.e., the number y between -pi and pi such that sin(y) == x.

EXAMPLES:

```
sage: x.arcsin()
arcsin(x)
sage: SR(0.5).arcsin()
1/6*pi
sage: SR(0.999).arcsin()
1.52607123962616
sage: SR(1/3).arcsin()
arcsin(1/3)
sage: SR(-1/3).arcsin()
-arcsin(1/3)
```

```
>>> from sage.all import *
>>> x.arcsin()
arcsin(x)
>>> SR(RealNumber('0.5')).arcsin()
1/6*pi
>>> SR(RealNumber('0.999')).arcsin()
1.52607123962616
>>> SR(Integer(1)/Integer(3)).arcsin()
arcsin(1/3)
>>> SR(-Integer(1)/Integer(3)).arcsin()
-arcsin(1/3)
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(0).arcsin()
0
sage: SR(0).arcsin(hold=True)
arcsin(0)
```

```
>>> from sage.all import *
>>> SR(Integer(0)).arcsin()
0
>>> SR(Integer(0)).arcsin(hold=True)
arcsin(0)
```

This also works using functional notation:

```
sage: arcsin(0,hold=True)
arcsin(0)
sage: arcsin(0)
0
```

```
>>> from sage.all import *
>>> arcsin(Integer(0),hold=True)
arcsin(0)
>>> arcsin(Integer(0))
0
```

To then evaluate again, we use `unhold()`:

```
sage: a = SR(0).arcsin(hold=True); a.unhold()
0
```

```
>>> from sage.all import *
>>> a = SR(Integer(0)).arcsin(hold=True); a.unhold()
0
```

arcsinh(*hold=False*)

Return the inverse hyperbolic sine of `self`.

EXAMPLES:

```
sage: x.arcsinh()
arcsinh(x)
sage: SR(0).arcsinh()
0
sage: SR(1).arcsinh()
arcsinh(1)
sage: SR(1.0).arcsinh()
0.881373587019543
sage: maxima('asinh(2.0)')
1.4436354751788...
```

```
>>> from sage.all import *
>>> x.arcsinh()
arcsinh(x)
>>> SR(Integer(0)).arcsinh()
0
>>> SR(Integer(1)).arcsinh()
arcsinh(1)
>>> SR(RealNumber('1.0')).arcsinh()
0.881373587019543
>>> maxima('asinh(2.0)')
1.4436354751788...
```

Sage automatically applies certain identities:

```
sage: SR(3/2).arcsinh().cosh()
1/2*sqrt(13)
```

```
>>> from sage.all import *
>>> SR(Integer(3)/Integer(2)).arcsinh().cosh()
1/2*sqrt(13)
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(-2).arcsinh()
-arcsinh(2)
sage: SR(-2).arcsinh(hold=True)
arcsinh(-2)
```

```
>>> from sage.all import *
>>> SR(-Integer(2)).arcsinh()
-arcsinh(2)
>>> SR(-Integer(2)).arcsinh(hold=True)
arcsinh(-2)
```

This also works using functional notation:

```
sage: arcsinh(-2,hold=True)
arcsinh(-2)
sage: arcsinh(-2)
-arcsinh(2)
```

```
>>> from sage.all import *
>>> arcsinh(-Integer(2),hold=True)
arcsinh(-2)
>>> arcsinh(-Integer(2))
-arcsinh(2)
```

To then evaluate again, we use *unhold()*:

```
sage: a = SR(-2).arcsinh(hold=True); a.unhold()
-arcsinh(2)
```

```
>>> from sage.all import *
>>> a = SR(-Integer(2)).arcsinh(hold=True); a.unhold()
-arcsinh(2)
```

arctan(*hold=False*)

Return the arc tangent of `self`.

EXAMPLES:

```
sage: x = var('x')
sage: x.arctan()
arctan(x)
sage: SR(1).arctan()
1/4*pi
sage: SR(1/2).arctan()
arctan(1/2)
sage: SR(0.5).arctan()
0.463647609000806
sage: plot(lambda x: SR(x).arctan(), -20,20)                              # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> x = var('x')
```

(continues on next page)

(continued from previous page)

```
>>> x.arctan()
arctan(x)
>>> SR(Integer(1)).arctan()
1/4*pi
>>> SR(Integer(1)/Integer(2)).arctan()
arctan(1/2)
>>> SR(RealNumber('0.5')).arctan()
0.463647609000806
>>> plot(lambda x: SR(x).arctan(), -Integer(20),Integer(20))              ␣
↪               # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(1).arctan(hold=True)
arctan(1)
```

```
>>> from sage.all import *
>>> SR(Integer(1)).arctan(hold=True)
arctan(1)
```

This also works using functional notation:

```
sage: arctan(1, hold=True)
arctan(1)
sage: arctan(1)
1/4*pi
```

```
>>> from sage.all import *
>>> arctan(Integer(1), hold=True)
arctan(1)
>>> arctan(Integer(1))
1/4*pi
```

To then evaluate again, we use *unhold()*:

```
sage: a = SR(1).arctan(hold=True); a.unhold()
1/4*pi
```

```
>>> from sage.all import *
>>> a = SR(Integer(1)).arctan(hold=True); a.unhold()
1/4*pi
```

arctan2(*x*, *hold=False*)

Return the inverse of the 2-variable tan function on `self` and `x`.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: x.arctan2(y)
arctan2(x, y)
```

(continues on next page)

(continued from previous page)

```
sage: SR(1/2).arctan2(1/2)
1/4*pi
sage: maxima.eval('atan2(1/2,1/2)')
'%pi/4'

sage: SR(-0.7).arctan2(SR(-0.6))
-2.27942259892257
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> x.arctan2(y)
arctan2(x, y)
>>> SR(Integer(1)/Integer(2)).arctan2(Integer(1)/Integer(2))
1/4*pi
>>> maxima.eval('atan2(1/2,1/2)')
'%pi/4'

>>> SR(-RealNumber('0.7')).arctan2(SR(-RealNumber('0.6')))
-2.27942259892257
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(1/2).arctan2(1/2, hold=True)
arctan2(1/2, 1/2)
```

```
>>> from sage.all import *
>>> SR(Integer(1)/Integer(2)).arctan2(Integer(1)/Integer(2), hold=True)
arctan2(1/2, 1/2)
```

This also works using functional notation:

```
sage: arctan2(1,2,hold=True)
arctan2(1, 2)
sage: arctan2(1,2)
arctan(1/2)
```

```
>>> from sage.all import *
>>> arctan2(Integer(1),Integer(2),hold=True)
arctan2(1, 2)
>>> arctan2(Integer(1),Integer(2))
arctan(1/2)
```

To then evaluate again, we use *unhold()*:

```
sage: a = SR(1/2).arctan2(1/2, hold=True); a.unhold()
1/4*pi
```

```
>>> from sage.all import *
>>> a = SR(Integer(1)/Integer(2)).arctan2(Integer(1)/Integer(2), hold=True);␣
→a.unhold()
1/4*pi
```

arctanh(*hold=False*)

Return the inverse hyperbolic tangent of `self`.

EXAMPLES:

```
sage: x.arctanh()
arctanh(x)
sage: SR(0).arctanh()
0
sage: SR(1/2).arctanh()
1/2*log(3)
sage: SR(0.5).arctanh()
0.549306144334055
sage: SR(0.5).arctanh().tanh()
0.500000000000000
sage: maxima('atanh(0.5)')  # abs tol 2e-16
0.5493061443340548
```

```
>>> from sage.all import *
>>> x.arctanh()
arctanh(x)
>>> SR(Integer(0)).arctanh()
0
>>> SR(Integer(1)/Integer(2)).arctanh()
1/2*log(3)
>>> SR(RealNumber('0.5')).arctanh()
0.549306144334055
>>> SR(RealNumber('0.5')).arctanh().tanh()
0.500000000000000
>>> maxima('atanh(0.5)')  # abs tol 2e-16
0.5493061443340548
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(-1/2).arctanh()
-1/2*log(3)
sage: SR(-1/2).arctanh(hold=True)
arctanh(-1/2)
```

```
>>> from sage.all import *
>>> SR(-Integer(1)/Integer(2)).arctanh()
-1/2*log(3)
>>> SR(-Integer(1)/Integer(2)).arctanh(hold=True)
arctanh(-1/2)
```

This also works using functional notation:

```
sage: arctanh(-1/2,hold=True)
arctanh(-1/2)
sage: arctanh(-1/2)
-1/2*log(3)
```

```
>>> from sage.all import *
>>> arctanh(-Integer(1)/Integer(2),hold=True)
arctanh(-1/2)
>>> arctanh(-Integer(1)/Integer(2))
-1/2*log(3)
```

To then evaluate again, we use `unhold()`:

```
sage: a = SR(-1/2).arctanh(hold=True); a.unhold()
-1/2*log(3)
```

```
>>> from sage.all import *
>>> a = SR(-Integer(1)/Integer(2)).arctanh(hold=True); a.unhold()
-1/2*log(3)
```

args()

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = x + y
sage: f.arguments()
(x, y)

sage: g = f.function(x)
sage: g.arguments()
(x,)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> f = x + y
>>> f.arguments()
(x, y)

>>> g = f.function(x)
>>> g.arguments()
(x,)
```

arguments()

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = x + y
sage: f.arguments()
(x, y)

sage: g = f.function(x)
sage: g.arguments()
(x,)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> f = x + y
```

(continues on next page)

(continued from previous page)

```
>>> f.arguments()
(x, y)

>>> g = f.function(x)
>>> g.arguments()
(x,)
```

assume()

Assume that this equation holds. This is relevant for symbolic integration, among other things.

EXAMPLES: We call the assume method to assume that $x > 2$:

```
sage: (x > 2).assume()
```

```
>>> from sage.all import *
>>> (x > Integer(2)).assume()
```

`bool` returns `True` below if the inequality is *definitely* known to be true.

```
sage: bool(x > 0)
True
sage: bool(x < 0)
False
```

```
>>> from sage.all import *
>>> bool(x > Integer(0))
True
>>> bool(x < Integer(0))
False
```

This may or may not be True, so `bool` returns False:

```
sage: bool(x > 3)
False
```

```
>>> from sage.all import *
>>> bool(x > Integer(3))
False
```

If you make inconsistent or meaningless assumptions, Sage will let you know:

```
sage: forget()
sage: assume(x<0)
sage: assume(x>0)
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
sage: assumptions()
[x < 0]
sage: forget()
```

```
>>> from sage.all import *
>>> forget()
>>> assume(x<Integer(0))
>>> assume(x>Integer(0))
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
>>> assumptions()
[x < 0]
>>> forget()
```

binomial(*k*, *hold=False*)

Return binomial coefficient "self choose k".

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: SR(5).binomial(SR(3))
10
sage: x.binomial(SR(3))
1/6*(x - 1)*(x - 2)*x
sage: x.binomial(y)
binomial(x, y)
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> SR(Integer(5)).binomial(SR(Integer(3)))
10
>>> x.binomial(SR(Integer(3)))
1/6*(x - 1)*(x - 2)*x
>>> x.binomial(y)
binomial(x, y)
```

To prevent automatic evaluation use the `hold` argument:

```
sage: x.binomial(3, hold=True)
binomial(x, 3)
sage: SR(5).binomial(3, hold=True)
binomial(5, 3)
```

```
>>> from sage.all import *
>>> x.binomial(Integer(3), hold=True)
binomial(x, 3)
>>> SR(Integer(5)).binomial(Integer(3), hold=True)
binomial(5, 3)
```

To then evaluate again, we use `unhold()`:

```
sage: a = SR(5).binomial(3, hold=True); a.unhold()
10
```

```
>>> from sage.all import *
>>> a = SR(Integer(5)).binomial(Integer(3), hold=True); a.unhold()
10
```

The `hold` parameter is also supported in functional notation:

```
sage: binomial(5,3, hold=True)
binomial(5, 3)
```

```
>>> from sage.all import *
>>> binomial(Integer(5),Integer(3), hold=True)
binomial(5, 3)
```

canonicalize_radical()

Choose a canonical branch of the given expression.

The square root, cube root, natural log, etc. functions are multi-valued. The `canonicalize_radical()` method will choose *one* of these values based on a heuristic.

For example, `sqrt(x^2)` has two values: `x`, and `-x`. The `canonicalize_radical()` function will choose *one* of them, consistently, based on the behavior of the expression as `x` tends to positive infinity. The solution chosen is the one which exhibits this same behavior. Since `sqrt(x^2)` approaches positive infinity as `x` does, the solution chosen is `x` (which also tends to positive infinity).

Warning

As shown in the examples below, a canonical form is not always returned, i.e., two mathematically identical expressions might be converted to different expressions.

Assumptions are not taken into account during the transformation. This may result in a branch choice inconsistent with your assumptions.

ALGORITHM:

This uses the Maxima `radcan()` command. From the Maxima documentation:

> Simplifies an expression, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables; that is, all functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, radcan produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by radcan to zero.
>
> For some expressions radcan is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial fraction expansions of exponents.

EXAMPLES:

`canonicalize_radical()` can perform some of the same manipulations as *log_expand()*:

```
sage: y = SR.symbol('y')
sage: f = log(x*y)
sage: f.log_expand()
log(x) + log(y)
sage: f.canonicalize_radical()
log(x) + log(y)
```

```
>>> from sage.all import *
>>> y = SR.symbol('y')
>>> f = log(x*y)
>>> f.log_expand()
log(x) + log(y)
>>> f.canonicalize_radical()
log(x) + log(y)
```

And also handles some exponential functions:

```
sage: f = (e^x-1)/(1+e^(x/2))
sage: f.canonicalize_radical()
e^(1/2*x) - 1
```

```
>>> from sage.all import *
>>> f = (e**x-Integer(1))/(Integer(1)+e**(x/Integer(2)))
>>> f.canonicalize_radical()
e^(1/2*x) - 1
```

It can also be used to change the base of a logarithm when the arguments to `log()` are positive real numbers:

```
sage: f = log(8)/log(2)
sage: f.canonicalize_radical()
3
```

```
>>> from sage.all import *
>>> f = log(Integer(8))/log(Integer(2))
>>> f.canonicalize_radical()
3
```

```
sage: a = SR.symbol('a')
sage: f = (log(x+x^2)-log(x))^a/log(1+x)^(a/2)
sage: f.canonicalize_radical()
log(x + 1)^(1/2*a)
```

```
>>> from sage.all import *
>>> a = SR.symbol('a')
>>> f = (log(x+x**Integer(2))-log(x))**a/log(Integer(1)+x)**(a/Integer(2))
>>> f.canonicalize_radical()
log(x + 1)^(1/2*a)
```

The simplest example of counter-intuitive behavior is what happens when we take the square root of a square:

```
sage: sqrt(x^2).canonicalize_radical()
x
```

```
>>> from sage.all import *
>>> sqrt(x**Integer(2)).canonicalize_radical()
x
```

If you don't want this kind of "simplification," don't use `canonicalize_radical()`.

This behavior can also be triggered when the expression under the radical is not given explicitly as a square:

```
sage: sqrt(x^2 - 2*x + 1).canonicalize_radical()
x - 1
```

```
>>> from sage.all import *
>>> sqrt(x**Integer(2) - Integer(2)*x + Integer(1)).canonicalize_radical()
x - 1
```

Another place where this can become confusing is with logarithms of complex numbers. Suppose `x` is complex with `x == r*e^(I*t)` (`r` real). Then `log(x)` is `log(r) + I*(t + 2*k*pi)` for some integer `k`.

Calling `canonicalize_radical()` will choose a branch, eliminating the solutions for all choices of `k` but one. Simplified by hand, the expression below is `(1/2)*log(2) + I*pi*k` for integer `k`. However, `canonicalize_radical()` will take each log expression, and choose one particular solution, dropping the other. When the results are subtracted, we're left with no imaginary part:

```
sage: f = (1/2)*log(2*x) + (1/2)*log(1/x)
sage: f.canonicalize_radical()
1/2*log(2)
```

```
>>> from sage.all import *
>>> f = (Integer(1)/Integer(2))*log(Integer(2)*x) + (Integer(1)/
↪Integer(2))*log(Integer(1)/x)
>>> f.canonicalize_radical()
1/2*log(2)
```

Naturally the result is wrong for some choices of `x`:

```
sage: f(x = -1)
I*pi + 1/2*log(2)
```

```
>>> from sage.all import *
>>> f(x = -Integer(1))
I*pi + 1/2*log(2)
```

The example below shows two expressions e1 and e2 which are "simplified" to different expressions, while their difference is "simplified" to zero; thus `canonicalize_radical()` does not return a canonical form:

```
sage: e1 = 1/(sqrt(5)+sqrt(2))
sage: e2 = (sqrt(5)-sqrt(2))/3
sage: e1.canonicalize_radical()
1/(sqrt(5) + sqrt(2))
sage: e2.canonicalize_radical()
1/3*sqrt(5) - 1/3*sqrt(2)
sage: (e1-e2).canonicalize_radical()
0
```

```
>>> from sage.all import *
>>> e1 = Integer(1)/(sqrt(Integer(5))+sqrt(Integer(2)))
>>> e2 = (sqrt(Integer(5))-sqrt(Integer(2)))/Integer(3)
>>> e1.canonicalize_radical()
1/(sqrt(5) + sqrt(2))
>>> e2.canonicalize_radical()
1/3*sqrt(5) - 1/3*sqrt(2)
```

(continues on next page)

(continued from previous page)

```
>>> (e1-e2).canonicalize_radical()
0
```

The issue reported in Issue #3520 is a case where `canonicalize_radical()` causes a numerical integral to be calculated incorrectly:

```
sage: f1 = sqrt(25 - x) * sqrt( 1 + 1/(4*(25-x)) )
sage: f2 = f1.canonicalize_radical()
sage: numerical_integral(f1.real(), 0, 1)[0] # abs tol 1e-10
4.974852579915647
sage: numerical_integral(f2.real(), 0, 1)[0] # abs tol 1e-10
-4.974852579915647
```

```
>>> from sage.all import *
>>> f1 = sqrt(Integer(25) - x) * sqrt( Integer(1) + Integer(1)/
↪(Integer(4)*(Integer(25)-x)) )
>>> f2 = f1.canonicalize_radical()
>>> numerical_integral(f1.real(), Integer(0), Integer(1))[Integer(0)] # abs␣
↪tol 1e-10
4.974852579915647
>>> numerical_integral(f2.real(), Integer(0), Integer(1))[Integer(0)] # abs␣
↪tol 1e-10
-4.974852579915647
```

coefficient(*s*, *n=1*)

Return the coefficient of s^n in this symbolic expression.

INPUT:

- `s` – expression
- `n` – expression, default 1

OUTPUT: a symbolic expression. The coefficient of s^n

Sometimes it may be necessary to expand or factor first, since this is not done automatically.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.collect(x)
x^3*sin(x*y) + (a + y + 1/y)*x + 2*sin(x*y)/x + 100
sage: f.coefficient(x,0)
100
sage: f.coefficient(x,-1)
2*sin(x*y)
sage: f.coefficient(x,1)
a + y + 1/y
sage: f.coefficient(x,2)
0
sage: f.coefficient(x,3)
sin(x*y)
```

(continues on next page)

(continued from previous page)

```
sage: f.coefficient(x^3)
sin(x*y)
sage: f.coefficient(sin(x*y))
x^3 + 2/x
sage: f.collect(sin(x*y))
a*x + x*y + (x^3 + 2/x)*sin(x*y) + x/y + 100

sage: var('a, x, y, z')
(a, x, y, z)
sage: f = (a*sqrt(2))*x^2 + sin(y)*x^(1/2) + z^z
sage: f.coefficient(sin(y))
sqrt(x)
sage: f.coefficient(x^2)
sqrt(2)*a
sage: f.coefficient(x^(1/2))
sin(y)
sage: f.coefficient(1)
0
sage: f.coefficient(x, 0)
z^z
```

```
>>> from sage.all import *
>>> var('x,y,a')
(x, y, a)
>>> f = Integer(100) + a*x + x**Integer(3)*sin(x*y) + x*y + x/y +␣
↪Integer(2)*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
>>> f.collect(x)
x^3*sin(x*y) + (a + y + 1/y)*x + 2*sin(x*y)/x + 100
>>> f.coefficient(x,Integer(0))
100
>>> f.coefficient(x,-Integer(1))
2*sin(x*y)
>>> f.coefficient(x,Integer(1))
a + y + 1/y
>>> f.coefficient(x,Integer(2))
0
>>> f.coefficient(x,Integer(3))
sin(x*y)
>>> f.coefficient(x**Integer(3))
sin(x*y)
>>> f.coefficient(sin(x*y))
x^3 + 2/x
>>> f.collect(sin(x*y))
a*x + x*y + (x^3 + 2/x)*sin(x*y) + x/y + 100

>>> var('a, x, y, z')
(a, x, y, z)
>>> f = (a*sqrt(Integer(2)))*x**Integer(2) + sin(y)*x**(Integer(1)/
↪Integer(2)) + z**z
>>> f.coefficient(sin(y))
sqrt(x)
```

(continues on next page)

(continued from previous page)

```
>>> f.coefficient(x**Integer(2))
sqrt(2)*a
>>> f.coefficient(x**(Integer(1)/Integer(2)))
sin(y)
>>> f.coefficient(Integer(1))
0
>>> f.coefficient(x, Integer(0))
z^z
```

Any coefficient can be queried:

```
sage: (x^2 + 3*x^pi).coefficient(x, pi)
3
sage: (2^x + 5*x^x).coefficient(x, x)
5
```

```
>>> from sage.all import *
>>> (x**Integer(2) + Integer(3)*x**pi).coefficient(x, pi)
3
>>> (Integer(2)**x + Integer(5)*x**x).coefficient(x, x)
5
```

coefficients(*x=None*, *sparse=True*)

Return the coefficients of this symbolic expression as a polynomial in x.

INPUT:

- `x` – (optional) variable

OUTPUT: depending on the value of `sparse`,

- A list of pairs `(expr, n)`, where `expr` is a symbolic expression and `n` is a power (`sparse=True`, default)
- A list of expressions where the `n`-th element is the coefficient of `x^n` when `self` is seen as polynomial in `x` (`sparse=False`).

EXAMPLES:

```
sage: var('x, y, a')
(x, y, a)
sage: p = x^3 - (x-3)*(x^2+x) + 1
sage: p.coefficients()
[[1, 0], [3, 1], [2, 2]]
sage: p.coefficients(sparse=False)
[1, 3, 2]
sage: p = x - x^3 + 5/7*x^5
sage: p.coefficients()
[[1, 1], [-1, 3], [5/7, 5]]
sage: p.coefficients(sparse=False)
[0, 1, 0, -1, 0, 5/7]
sage: p = expand((x-a*sqrt(2))^2 + x + 1); p
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
sage: p.coefficients(a)
[[x^2 + x + 1, 0], [-2*sqrt(2)*x, 1], [2, 2]]
```

(continues on next page)

(continued from previous page)

```
sage: p.coefficients(a, sparse=False)
[x^2 + x + 1, -2*sqrt(2)*x, 2]
sage: p.coefficients(x)
[[2*a^2 + 1, 0], [-2*sqrt(2)*a + 1, 1], [1, 2]]
sage: p.coefficients(x, sparse=False)
[2*a^2 + 1, -2*sqrt(2)*a + 1, 1]
```

```
>>> from sage.all import *
>>> var('x, y, a')
(x, y, a)
>>> p = x**Integer(3) - (x-Integer(3))*(x**Integer(2)+x) + Integer(1)
>>> p.coefficients()
[[1, 0], [3, 1], [2, 2]]
>>> p.coefficients(sparse=False)
[1, 3, 2]
>>> p = x - x**Integer(3) + Integer(5)/Integer(7)*x**Integer(5)
>>> p.coefficients()
[[1, 1], [-1, 3], [5/7, 5]]
>>> p.coefficients(sparse=False)
[0, 1, 0, -1, 0, 5/7]
>>> p = expand((x-a*sqrt(Integer(2)))**Integer(2) + x + Integer(1)); p
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
>>> p.coefficients(a)
[[x^2 + x + 1, 0], [-2*sqrt(2)*x, 1], [2, 2]]
>>> p.coefficients(a, sparse=False)
[x^2 + x + 1, -2*sqrt(2)*x, 2]
>>> p.coefficients(x)
[[2*a^2 + 1, 0], [-2*sqrt(2)*a + 1, 1], [1, 2]]
>>> p.coefficients(x, sparse=False)
[2*a^2 + 1, -2*sqrt(2)*a + 1, 1]
```

collect(*s*)

Collect the coefficients of `s` into a group.

INPUT:

- `s` – the symbol whose coefficients will be collected

OUTPUT:

A new expression, equivalent to the original one, with the coefficients of `s` grouped.

Note

The expression is not expanded or factored before the grouping takes place. For best results, call *expand()* on the expression before *collect()*.

EXAMPLES:

In the first term of f, x has a coefficient of $4y$. In the second term, x has a coefficient of z. Therefore, if we collect those coefficients, x will have a coefficient of $4y + z$:

```
sage: x,y,z = var('x,y,z')
sage: f = 4*x*y + x*z + 20*y^2 + 21*y*z + 4*z^2 + x^2*y^2*z^2
sage: f.collect(x)
x^2*y^2*z^2 + x*(4*y + z) + 20*y^2 + 21*y*z + 4*z^2
```

```
>>> from sage.all import *
>>> x,y,z = var('x,y,z')
>>> f = Integer(4)*x*y + x*z + Integer(20)*y**Integer(2) + Integer(21)*y*z +␣
↪Integer(4)*z**Integer(2) + x**Integer(2)*y**Integer(2)*z**Integer(2)
>>> f.collect(x)
x^2*y^2*z^2 + x*(4*y + z) + 20*y^2 + 21*y*z + 4*z^2
```

Here we do the same thing for y and z; however, note that we do not factor the y^2 and z^2 terms before collecting coefficients:

```
sage: f.collect(y)
(x^2*z^2 + 20)*y^2 + (4*x + 21*z)*y + x*z + 4*z^2
sage: f.collect(z)
(x^2*y^2 + 4)*z^2 + 4*x*y + 20*y^2 + (x + 21*y)*z
```

```
>>> from sage.all import *
>>> f.collect(y)
(x^2*z^2 + 20)*y^2 + (4*x + 21*z)*y + x*z + 4*z^2
>>> f.collect(z)
(x^2*y^2 + 4)*z^2 + 4*x*y + 20*y^2 + (x + 21*y)*z
```

The terms are collected, whether the expression is expanded or not:

```
sage: f = (x + y)*(x - z)
sage: f.collect(x)
x^2 + x*(y - z) - y*z
sage: f.expand().collect(x)
x^2 + x*(y - z) - y*z
```

```
>>> from sage.all import *
>>> f = (x + y)*(x - z)
>>> f.collect(x)
x^2 + x*(y - z) - y*z
>>> f.expand().collect(x)
x^2 + x*(y - z) - y*z
```

collect_common_factors()

This function does not perform a full factorization but only looks for factors which are already explicitly present.

Polynomials can often be brought into a more compact form by collecting common factors from the terms of sums. This is accomplished by this function.

EXAMPLES:

```
sage: var('x')
x
sage: (x/(x^2 + x)).collect_common_factors()
```

(continues on next page)

(continued from previous page)

```
1/(x + 1)

sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: (a*x+a*y).collect_common_factors()
a*(x + y)
sage: (a*x^2+2*a*x*y+a*y^2).collect_common_factors()
(x^2 + 2*x*y + y^2)*a
sage: (a*(b*(a+c)*x+b*((a+c)*x+(a+c)*y)*y)).collect_common_factors()
((x + y)*y + x)*(a + c)*a*b
```

```
>>> from sage.all import *
>>> var('x')
x
>>> (x/(x**Integer(2) + x)).collect_common_factors()
1/(x + 1)

>>> var('a,b,c,x,y')
(a, b, c, x, y)
>>> (a*x+a*y).collect_common_factors()
a*(x + y)
>>> (a*x**Integer(2)+Integer(2)*a*x*y+a*y**Integer(2)).collect_common_
↪factors()
(x^2 + 2*x*y + y^2)*a
>>> (a*(b*(a+c)*x+b*((a+c)*x+(a+c)*y)*y)).collect_common_factors()
((x + y)*y + x)*(a + c)*a*b
```

combine(*deep=False*)

Return a simplified version of this symbolic expression by combining all toplevel terms with the same denominator into a single term.

Please use the keyword `deep=True` to apply the process recursively.

EXAMPLES:

```
sage: var('x, y, a, b, c')
(x, y, a, b, c)
sage: f = x*(x-1)/(x^2 - 7) + y^2/(x^2-7) + 1/(x+1) + b/a + c/a; f
(x - 1)*x/(x^2 - 7) + y^2/(x^2 - 7) + b/a + c/a + 1/(x + 1)
sage: f.combine()
((x - 1)*x + y^2)/(x^2 - 7) + (b + c)/a + 1/(x + 1)
sage: (1/x + 1/x^2 + (x+1)/x).combine()
(x + 2)/x + 1/x^2
sage: ex = 1/x + ((x + 1)/x - 1/x)/x^2 + (x+1)/x; ex
(x + 1)/x + 1/x + ((x + 1)/x - 1/x)/x^2
sage: ex.combine()
(x + 2)/x + ((x + 1)/x - 1/x)/x^2
sage: ex.combine(deep=True)
(x + 2)/x + 1/x^2
sage: (1+sin((x + 1)/x - 1/x)).combine(deep=True)
sin(1) + 1
```

```
>>> from sage.all import *
>>> var('x, y, a, b, c')
(x, y, a, b, c)
>>> f = x*(x-Integer(1))/(x**Integer(2) - Integer(7)) + y**Integer(2)/
↪(x**Integer(2)-Integer(7)) + Integer(1)/(x+Integer(1)) + b/a + c/a; f
(x - 1)*x/(x^2 - 7) + y^2/(x^2 - 7) + b/a + c/a + 1/(x + 1)
>>> f.combine()
((x - 1)*x + y^2)/(x^2 - 7) + (b + c)/a + 1/(x + 1)
>>> (Integer(1)/x + Integer(1)/x**Integer(2) + (x+Integer(1))/x).combine()
(x + 2)/x + 1/x^2
>>> ex = Integer(1)/x + ((x + Integer(1))/x - Integer(1)/x)/x**Integer(2) +␣
↪(x+Integer(1))/x; ex
(x + 1)/x + 1/x + ((x + 1)/x - 1/x)/x^2
>>> ex.combine()
(x + 2)/x + ((x + 1)/x - 1/x)/x^2
>>> ex.combine(deep=True)
(x + 2)/x + 1/x^2
>>> (Integer(1)+sin((x + Integer(1))/x - Integer(1)/x)).combine(deep=True)
sin(1) + 1
```

conjugate(*hold=False*)

Return the complex conjugate of this symbolic expression.

EXAMPLES:

```
sage: a = 1 + 2*I
sage: a.conjugate()
-2*I + 1
sage: a = sqrt(2) + 3^(1/3)*I; a
sqrt(2) + I*3^(1/3)
sage: a.conjugate()
sqrt(2) - I*3^(1/3)

sage: SR(CDF.0).conjugate()
-1.0*I
sage: x.conjugate()
conjugate(x)
sage: SR(RDF(1.5)).conjugate()
1.5
sage: SR(float(1.5)).conjugate()
1.5
sage: SR(I).conjugate()
-I
sage: ( 1+I  + (2-3*I)*x).conjugate()
(3*I + 2)*conjugate(x) - I + 1
```

```
>>> from sage.all import *
>>> a = Integer(1) + Integer(2)*I
>>> a.conjugate()
-2*I + 1
>>> a = sqrt(Integer(2)) + Integer(3)**(Integer(1)/Integer(3))*I; a
sqrt(2) + I*3^(1/3)
>>> a.conjugate()
```

(continues on next page)

(continued from previous page)

```
sqrt(2) - I*3^(1/3)

>>> SR(CDF.gen(0)).conjugate()
-1.0*I
>>> x.conjugate()
conjugate(x)
>>> SR(RDF(RealNumber('1.5'))).conjugate()
1.5
>>> SR(float(RealNumber('1.5'))).conjugate()
1.5
>>> SR(I).conjugate()
-I
>>> ( Integer(1)+I  + (Integer(2)-Integer(3)*I)*x).conjugate()
(3*I + 2)*conjugate(x) - I + 1
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(I).conjugate(hold=True)
conjugate(I)
```

```
>>> from sage.all import *
>>> SR(I).conjugate(hold=True)
conjugate(I)
```

This also works in functional notation:

```
sage: conjugate(I)
-I
sage: conjugate(I,hold=True)
conjugate(I)
```

```
>>> from sage.all import *
>>> conjugate(I)
-I
>>> conjugate(I,hold=True)
conjugate(I)
```

To then evaluate again, we use *unhold()*:

```
sage: a = SR(I).conjugate(hold=True); a.unhold()
-I
```

```
>>> from sage.all import *
>>> a = SR(I).conjugate(hold=True); a.unhold()
-I
```

content(*s*)

Return the content of this expression when considered as a polynomial in `s`.

See also *unit()*, *primitive_part()*, and *unit_content_primitive()*.

INPUT:

- `s` – a symbolic expression

OUTPUT:

The content part of a polynomial as a symbolic expression. It is defined as the gcd of the coefficients.

> **Warning**
>
> The expression is considered to be a univariate polynomial in `s`. The output is different from the `content()` method provided by multivariate polynomial rings in Sage.

EXAMPLES:

```
sage: (2*x+4).content(x)
2
sage: (2*x+1).content(x)
1
sage: (2*x+1/2).content(x)
1/2
sage: var('y')
y
sage: (2*x + 4*sin(y)).content(sin(y))
2
```

```
>>> from sage.all import *
>>> (Integer(2)*x+Integer(4)).content(x)
2
>>> (Integer(2)*x+Integer(1)).content(x)
1
>>> (Integer(2)*x+Integer(1)/Integer(2)).content(x)
1/2
>>> var('y')
y
>>> (Integer(2)*x + Integer(4)*sin(y)).content(sin(y))
2
```

contradicts(*soln*)

Return `True` if this relation is violated by the given variable assignment(s).

EXAMPLES:

```
sage: (x<3).contradicts(x==0)
False
sage: (x<3).contradicts(x==3)
True
sage: (x<=3).contradicts(x==3)
False
sage: y = var('y')
sage: (x<y).contradicts(x==30)
False
sage: (x<y).contradicts({x: 30, y: 20})
True
```

```
>>> from sage.all import *
>>> (x<Integer(3)).contradicts(x==Integer(0))
```

(continues on next page)

(continued from previous page)

```
False
>>> (x<Integer(3)).contradicts(x==Integer(3))
True
>>> (x<=Integer(3)).contradicts(x==Integer(3))
False
>>> y = var('y')
>>> (x<y).contradicts(x==Integer(30))
False
>>> (x<y).contradicts({x: Integer(30), y: Integer(20)})
True
```

convert(*target=None*)

Call the convert function in the units package. For symbolic variables that are not units, this function just returns the variable.

INPUT:

- `self` – the symbolic expression converting from
- `target` – (default: `None`) the symbolic expression converting to

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: units.length.foot.convert()
381/1250*meter
sage: units.mass.kilogram.convert(units.mass.pound)
100000000/45359237*pound
```

```
>>> from sage.all import *
>>> units.length.foot.convert()
381/1250*meter
>>> units.mass.kilogram.convert(units.mass.pound)
100000000/45359237*pound
```

We do not get anything new by converting an ordinary symbolic variable:

```
sage: a = var('a')
sage: a - a.convert()
0
```

```
>>> from sage.all import *
>>> a = var('a')
>>> a - a.convert()
0
```

Raises `ValueError` if `self` and `target` are not convertible:

```
sage: units.mass.kilogram.convert(units.length.foot)
Traceback (most recent call last):
...
ValueError: Incompatible units
sage: (units.length.meter^2).convert(units.length.foot)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: Incompatible units
```

```
>>> from sage.all import *
>>> units.mass.kilogram.convert(units.length.foot)
Traceback (most recent call last):
...
ValueError: Incompatible units
>>> (units.length.meter**Integer(2)).convert(units.length.foot)
Traceback (most recent call last):
...
ValueError: Incompatible units
```

Recognizes derived unit relationships to base units and other derived units:

```
sage: (units.length.foot/units.time.second^2).convert(units.acceleration.
↪galileo)
762/25*galileo
sage: (units.mass.kilogram*units.length.meter/units.time.second^2).
↪convert(units.force.newton)
newton
sage: (units.length.foot^3).convert(units.area.acre*units.length.inch)
1/3630*(acre*inch)
sage: (units.charge.coulomb).convert(units.current.ampere*units.time.second)
(ampere*second)
sage: (units.pressure.pascal*units.si_prefixes.kilo).convert(units.pressure.
↪pounds_per_square_inch)
1290320000000/8896443230521*pounds_per_square_inch
```

```
>>> from sage.all import *
>>> (units.length.foot/units.time.second**Integer(2)).convert(units.
↪acceleration.galileo)
762/25*galileo
>>> (units.mass.kilogram*units.length.meter/units.time.second**Integer(2)).
↪convert(units.force.newton)
newton
>>> (units.length.foot**Integer(3)).convert(units.area.acre*units.length.inch)
1/3630*(acre*inch)
>>> (units.charge.coulomb).convert(units.current.ampere*units.time.second)
(ampere*second)
>>> (units.pressure.pascal*units.si_prefixes.kilo).convert(units.pressure.
↪pounds_per_square_inch)
1290320000000/8896443230521*pounds_per_square_inch
```

For decimal answers multiply by 1.0:

```
sage: (units.pressure.pascal*units.si_prefixes.kilo).convert(units.pressure.
↪pounds_per_square_inch)*1.0
0.145037737730209*pounds_per_square_inch
```

```
>>> from sage.all import *
>>> (units.pressure.pascal*units.si_prefixes.kilo).convert(units.pressure.
↪pounds_per_square_inch)*RealNumber('1.0')
0.145037737730209*pounds_per_square_inch
```

Converting temperatures works as well:

```
sage: s = 68*units.temperature.fahrenheit
sage: s.convert(units.temperature.celsius)
20*celsius
sage: s.convert()
293.150000000000*kelvin
```

```
>>> from sage.all import *
>>> s = Integer(68)*units.temperature.fahrenheit
>>> s.convert(units.temperature.celsius)
20*celsius
>>> s.convert()
293.150000000000*kelvin
```

Trying to multiply temperatures by another unit then converting raises a ValueError:

```
sage: wrong = 50*units.temperature.celsius*units.length.foot
sage: wrong.convert()
Traceback (most recent call last):
...
ValueError: cannot convert
```

```
>>> from sage.all import *
>>> wrong = Integer(50)*units.temperature.celsius*units.length.foot
>>> wrong.convert()
Traceback (most recent call last):
...
ValueError: cannot convert
```

cos(*hold=False*)

Return the cosine of `self`.

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: cos(x^2 + y^2)
cos(x^2 + y^2)
sage: cos(sage.symbolic.constants.pi)
-1
sage: cos(SR(1))
cos(1)
sage: cos(SR(RealField(150)(1)))
0.54030230586813971740093660744297660373231042
```

```
>>> from sage.all import *
>>> var('x, y')
```

(continues on next page)

(continued from previous page)

```
(x, y)
>>> cos(x**Integer(2) + y**Integer(2))
cos(x^2 + y^2)
>>> cos(sage.symbolic.constants.pi)
-1
>>> cos(SR(Integer(1)))
cos(1)
>>> cos(SR(RealField(Integer(150))(Integer(1))))
0.54030230586813971740093660744297660373231042
```

In order to get a numeric approximation use .n():

```
sage: SR(RR(1)).cos().n()
0.540302305868140
sage: SR(float(1)).cos().n()
0.540302305868140
```

```
>>> from sage.all import *
>>> SR(RR(Integer(1))).cos().n()
0.540302305868140
>>> SR(float(Integer(1))).cos().n()
0.540302305868140
```

To prevent automatic evaluation use the `hold` argument:

```
sage: pi.cos()
-1
sage: pi.cos(hold=True)
cos(pi)
```

```
>>> from sage.all import *
>>> pi.cos()
-1
>>> pi.cos(hold=True)
cos(pi)
```

This also works using functional notation:

```
sage: cos(pi,hold=True)
cos(pi)
sage: cos(pi)
-1
```

```
>>> from sage.all import *
>>> cos(pi,hold=True)
cos(pi)
>>> cos(pi)
-1
```

To then evaluate again, we use *unhold()*:

```
sage: a = pi.cos(hold=True); a.unhold()
-1
```

```
>>> from sage.all import *
>>> a = pi.cos(hold=True); a.unhold()
-1
```

cosh(*hold=False*)

Return cosh of `self`.

We have $\cosh(x) = (e^x + e^{-x})/2$.

EXAMPLES:

```
sage: x.cosh()
cosh(x)
sage: SR(1).cosh()
cosh(1)
sage: SR(0).cosh()
1
sage: SR(1.0).cosh()
1.54308063481524
sage: maxima('cosh(1.0)')
1.54308063481524...
sage: SR(1.00000000000000000000000000).cosh()
1.5430806348152437784779056
sage: SR(RIF(1)).cosh()
1.543080634815244?
```

```
>>> from sage.all import *
>>> x.cosh()
cosh(x)
>>> SR(Integer(1)).cosh()
cosh(1)
>>> SR(Integer(0)).cosh()
1
>>> SR(RealNumber('1.0')).cosh()
1.54308063481524
>>> maxima('cosh(1.0)')
1.54308063481524...
>>> SR(RealNumber('1.00000000000000000000000000')).cosh()
1.5430806348152437784779056
>>> SR(RIF(Integer(1))).cosh()
1.543080634815244?
```

To prevent automatic evaluation use the `hold` argument:

```
sage: arcsinh(x).cosh()
sqrt(x^2 + 1)
sage: arcsinh(x).cosh(hold=True)
cosh(arcsinh(x))
```

```
>>> from sage.all import *
>>> arcsinh(x).cosh()
sqrt(x^2 + 1)
>>> arcsinh(x).cosh(hold=True)
cosh(arcsinh(x))
```

This also works using functional notation:

```
sage: cosh(arcsinh(x),hold=True)
cosh(arcsinh(x))
sage: cosh(arcsinh(x))
sqrt(x^2 + 1)
```

```
>>> from sage.all import *
>>> cosh(arcsinh(x),hold=True)
cosh(arcsinh(x))
>>> cosh(arcsinh(x))
sqrt(x^2 + 1)
```

To then evaluate again, we use `unhold()`:

```
sage: a = arcsinh(x).cosh(hold=True); a.unhold()
sqrt(x^2 + 1)
```

```
>>> from sage.all import *
>>> a = arcsinh(x).cosh(hold=True); a.unhold()
sqrt(x^2 + 1)
```

csgn(*hold=False*)

Return the sign of `self`, which is -1 if `self < 0`, 0 if `self == 0`, and 1 if `self > 0`, or unevaluated when `self` is a nonconstant symbolic expression.

If `self` is not real, return the complex half-plane (left or right) in which the number lies. If `self` is pure imaginary, return the sign of the imaginary part of `self`.

EXAMPLES:

```
sage: x = var('x')
sage: SR(-2).csgn()
-1
sage: SR(0.0).csgn()
0
sage: SR(10).csgn()
1
sage: x.csgn()
csgn(x)
sage: SR(CDF.0).csgn()
1
sage: SR(I).csgn()
1
sage: SR(-I).csgn()
-1
sage: SR(1+I).csgn()
```

(continues on next page)

(continued from previous page)

```
1
sage: SR(1-I).csgn()
1
sage: SR(-1+I).csgn()
-1
sage: SR(-1-I).csgn()
-1
```

```
>>> from sage.all import *
>>> x = var('x')
>>> SR(-Integer(2)).csgn()
-1
>>> SR(RealNumber('0.0')).csgn()
0
>>> SR(Integer(10)).csgn()
1
>>> x.csgn()
csgn(x)
>>> SR(CDF.gen(0)).csgn()
1
>>> SR(I).csgn()
1
>>> SR(-I).csgn()
-1
>>> SR(Integer(1)+I).csgn()
1
>>> SR(Integer(1)-I).csgn()
1
>>> SR(-Integer(1)+I).csgn()
-1
>>> SR(-Integer(1)-I).csgn()
-1
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(I).csgn(hold=True)
csgn(I)
```

```
>>> from sage.all import *
>>> SR(I).csgn(hold=True)
csgn(I)
```

decl_assume(*decl*)

decl_forget(*decl*)

default_variable()

Return the default variable, which is by definition the first variable in `self`, or x if there are no variables in `self`. The result is cached.

EXAMPLES:

```
sage: sqrt(2).default_variable()
x
sage: x, theta, a = var('x, theta, a')
sage: f = x^2 + theta^3 - a^x
sage: f.default_variable()
a
```

```
>>> from sage.all import *
>>> sqrt(Integer(2)).default_variable()
x
>>> x, theta, a = var('x, theta, a')
>>> f = x**Integer(2) + theta**Integer(3) - a**x
>>> f.default_variable()
a
```

Note that this is the first *variable*, not the first *argument*:

```
sage: f(theta, a, x) = a + theta^3
sage: f.default_variable()
a
sage: f.variables()
(a, theta)
sage: f.arguments()
(theta, a, x)
```

```
>>> from sage.all import *
>>> __tmp__=var("theta,a,x"); f = symbolic_expression(a + theta**Integer(3)).
↪function(theta,a,x)
>>> f.default_variable()
a
>>> f.variables()
(a, theta)
>>> f.arguments()
(theta, a, x)
```

degree (*s*)

Return the exponent of the highest power of `s` in `self`.

OUTPUT: integer

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y^10 + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + 2*sin(x*y)/x + x/y^10 + 100
sage: f.degree(x)
3
sage: f.degree(y)
1
sage: f.degree(sin(x*y))
1
sage: (x^-3+y).degree(x)
```

(continues on next page)

(continued from previous page)

```
0
sage: (1/x+1/x**2).degree(x)
-1
```

```
>>> from sage.all import *
>>> var('x,y,a')
(x, y, a)
>>> f = Integer(100) + a*x + x**Integer(3)*sin(x*y) + x*y + x/y**Integer(10)␣
↪+ Integer(2)*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + 2*sin(x*y)/x + x/y^10 + 100
>>> f.degree(x)
3
>>> f.degree(y)
1
>>> f.degree(sin(x*y))
1
>>> (x**-Integer(3)+y).degree(x)
0
>>> (Integer(1)/x+Integer(1)/x**Integer(2)).degree(x)
-1
```

demoivre(*force=False*)

Return this symbolic expression with complex exponentials (optionally all exponentials) replaced by (at least partially) trigonometric/hyperbolic expressions.

EXAMPLES:

```
sage: x, a, b = SR.var("x, a, b")
sage: exp(a + I*b).demoivre()
(cos(b) + I*sin(b))*e^a
sage: exp(I*x).demoivre()
cos(x) + I*sin(x)
sage: exp(x).demoivre()
e^x
sage: exp(x).demoivre(force=True)
cosh(x) + sinh(x)
```

```
>>> from sage.all import *
>>> x, a, b = SR.var("x, a, b")
>>> exp(a + I*b).demoivre()
(cos(b) + I*sin(b))*e^a
>>> exp(I*x).demoivre()
cos(x) + I*sin(x)
>>> exp(x).demoivre()
e^x
>>> exp(x).demoivre(force=True)
cosh(x) + sinh(x)
```

denominator(*normalize=True*)

Return the denominator of this symbolic expression.

INPUT:

- `normalize` – boolean (default: `True`)

If `normalize` is `True`, the expression is first normalized to have it as a fraction before getting the denominator.

If `normalize` is `False`, the expression is kept and if it is not a quotient, then this will just return 1.

See also:

normalize(), *numerator()*, *numerator_denominator()*, *combine()*

EXAMPLES:

```
sage: x, y, z, theta = var('x, y, z, theta')
sage: f = (sqrt(x) + sqrt(y) + sqrt(z))/(x^10 - y^10 - sqrt(theta))
sage: f.numerator()
sqrt(x) + sqrt(y) + sqrt(z)
sage: f.denominator()
x^10 - y^10 - sqrt(theta)

sage: f.numerator(normalize=False)
(sqrt(x) + sqrt(y) + sqrt(z))
sage: f.denominator(normalize=False)
x^10 - y^10 - sqrt(theta)

sage: y = var('y')
sage: g = x + y/(x + 2); g
x + y/(x + 2)
sage: g.numerator(normalize=False)
x + y/(x + 2)
sage: g.denominator(normalize=False)
1
```

```
>>> from sage.all import *
>>> x, y, z, theta = var('x, y, z, theta')
>>> f = (sqrt(x) + sqrt(y) + sqrt(z))/(x**Integer(10) - y**Integer(10) -␣
↪sqrt(theta))
>>> f.numerator()
sqrt(x) + sqrt(y) + sqrt(z)
>>> f.denominator()
x^10 - y^10 - sqrt(theta)

>>> f.numerator(normalize=False)
(sqrt(x) + sqrt(y) + sqrt(z))
>>> f.denominator(normalize=False)
x^10 - y^10 - sqrt(theta)

>>> y = var('y')
>>> g = x + y/(x + Integer(2)); g
x + y/(x + 2)
>>> g.numerator(normalize=False)
x + y/(x + 2)
>>> g.denominator(normalize=False)
1
```

derivative(**args*)

Return the derivative of this expressions with respect to the variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global `derivative()` function for more details.

See also:

This is implemented in the `_derivative` method (see the source code).

EXAMPLES:

```
sage: var("x y")
(x, y)
sage: t = (x^2+y)^2
sage: t.derivative(x)
4*(x^2 + y)*x
sage: t.derivative(x, 2)
12*x^2 + 4*y
sage: t.derivative(x, 2, y)
4
sage: t.derivative(y)
2*x^2 + 2*y
```

```
>>> from sage.all import *
>>> var("x y")
(x, y)
>>> t = (x**Integer(2)+y)**Integer(2)
>>> t.derivative(x)
4*(x^2 + y)*x
>>> t.derivative(x, Integer(2))
12*x^2 + 4*y
>>> t.derivative(x, Integer(2), y)
4
>>> t.derivative(y)
2*x^2 + 2*y
```

If the function depends on only one variable, you may omit the variable. Giving just a number (for the order of the derivative) also works:

```
sage: f(x) = x^3 + sin(x)
sage: f.derivative()
x |--> 3*x^2 + cos(x)
sage: f.derivative(2)
x |--> 6*x - sin(x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(3) + sin(x)).
↪function(x)
>>> f.derivative()
x |--> 3*x^2 + cos(x)
>>> f.derivative(Integer(2))
x |--> 6*x - sin(x)
```

Some expressions can't be cleanly differentiated by the chain rule:

```
sage: _ = var('x', domain='real')
sage: _ = var('w z')
sage: (x^z).conjugate().diff(x)
conjugate(x^(z - 1))*conjugate(z)
sage: (w^z).conjugate().diff(w)
w^(z - 1)*z*D[0](conjugate)(w^z)
sage: atanh(x).real_part().diff(x)
-1/(x^2 - 1)
sage: atanh(x).imag_part().diff(x)
0
sage: atanh(w).real_part().diff(w)
-D[0](real_part)(arctanh(w))/(w^2 - 1)
sage: atanh(w).imag_part().diff(w)
-D[0](imag_part)(arctanh(w))/(w^2 - 1)
sage: abs(log(x)).diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
sage: abs(log(z)).diff(z)
1/2*(conjugate(log(z))/z + log(z)/conjugate(z))/abs(log(z))
sage: forget()

sage: t = sin(x+y^2)*tan(x*y)
sage: t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
sage: t.derivative(y)
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)
```

```
>>> from sage.all import *
>>> _ = var('x', domain='real')
>>> _ = var('w z')
>>> (x**z).conjugate().diff(x)
conjugate(x^(z - 1))*conjugate(z)
>>> (w**z).conjugate().diff(w)
w^(z - 1)*z*D[0](conjugate)(w^z)
>>> atanh(x).real_part().diff(x)
-1/(x^2 - 1)
>>> atanh(x).imag_part().diff(x)
0
>>> atanh(w).real_part().diff(w)
-D[0](real_part)(arctanh(w))/(w^2 - 1)
>>> atanh(w).imag_part().diff(w)
-D[0](imag_part)(arctanh(w))/(w^2 - 1)
>>> abs(log(x)).diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
>>> abs(log(z)).diff(z)
1/2*(conjugate(log(z))/z + log(z)/conjugate(z))/abs(log(z))
>>> forget()

>>> t = sin(x+y**Integer(2))*tan(x*y)
>>> t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
>>> t.derivative(y)
```

(continues on next page)

(continued from previous page)

```
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)
```

```
sage: h = sin(x)/cos(x)
sage: derivative(h,x,x,x)
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
sage: derivative(h,x,3)
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
```

```
>>> from sage.all import *
>>> h = sin(x)/cos(x)
>>> derivative(h,x,x,x)
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
>>> derivative(h,x,Integer(3))
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
```

```
sage: var('x, y')
(x, y)
sage: u = (sin(x) + cos(y))*(cos(x) - sin(y))
sage: derivative(u,x,y)
-cos(x)*cos(y) + sin(x)*sin(y)
sage: f = ((x^2+1)/(x^2-1))^(1/4)
sage: g = derivative(f, x); g # this is a complex expression
-1/2*((x^2 + 1)*x/(x^2 - 1)^2 - x/(x^2 - 1))/((x^2 + 1)/(x^2 - 1))^(3/4)
sage: g.factor()
-x/((x + 1)^2*(x - 1)^2*((x^2 + 1)/(x^2 - 1))^(3/4))
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> u = (sin(x) + cos(y))*(cos(x) - sin(y))
>>> derivative(u,x,y)
-cos(x)*cos(y) + sin(x)*sin(y)
>>> f = ((x**Integer(2)+Integer(1))/(x**Integer(2)-Integer(1)))**(Integer(1)/
↪Integer(4))
>>> g = derivative(f, x); g # this is a complex expression
-1/2*((x^2 + 1)*x/(x^2 - 1)^2 - x/(x^2 - 1))/((x^2 + 1)/(x^2 - 1))^(3/4)
>>> g.factor()
-x/((x + 1)^2*(x - 1)^2*((x^2 + 1)/(x^2 - 1))^(3/4))
```

```
sage: y = var('y')
sage: f = y^(sin(x))
sage: derivative(f, x)
y^sin(x)*cos(x)*log(y)
```

```
>>> from sage.all import *
>>> y = var('y')
>>> f = y**(sin(x))
>>> derivative(f, x)
y^sin(x)*cos(x)*log(y)
```

```
sage: g(x) = sqrt(5-2*x)
sage: g_3 = derivative(g, x, 3); g_3(2)
-3
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); g = symbolic_expression(sqrt(Integer(5)-Integer(2)*x)).
↪function(x)
>>> g_3 = derivative(g, x, Integer(3)); g_3(Integer(2))
-3
```

```
sage: f = x*e^(-x)
sage: derivative(f, 100)
x*e^(-x) - 100*e^(-x)
```

```
>>> from sage.all import *
>>> f = x*e**(-x)
>>> derivative(f, Integer(100))
x*e^(-x) - 100*e^(-x)
```

```
sage: g = 1/(sqrt((x^2-1)*(x+5)^6))
sage: derivative(g, x)
-((x + 5)^6*x + 3*(x^2 - 1)*(x + 5)^5)/((x^2 - 1)*(x + 5)^6)^(3/2)
```

```
>>> from sage.all import *
>>> g = Integer(1)/(sqrt((x**Integer(2)-
↪Integer(1))*(x+Integer(5))**Integer(6)))
>>> derivative(g, x)
-((x + 5)^6*x + 3*(x^2 - 1)*(x + 5)^5)/((x^2 - 1)*(x + 5)^6)^(3/2)
```

diff(**args*)

Return the derivative of this expressions with respect to the variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global `derivative()` function for more details.

See also

This is implemented in the `_derivative` method (see the source code).

EXAMPLES:

```
sage: var("x y")
(x, y)
sage: t = (x^2+y)^2
sage: t.derivative(x)
4*(x^2 + y)*x
sage: t.derivative(x, 2)
12*x^2 + 4*y
sage: t.derivative(x, 2, y)
4
```

(continues on next page)

(continued from previous page)

```
sage: t.derivative(y)
2*x^2 + 2*y
```

```
>>> from sage.all import *
>>> var("x y")
(x, y)
>>> t = (x**Integer(2)+y)**Integer(2)
>>> t.derivative(x)
4*(x^2 + y)*x
>>> t.derivative(x, Integer(2))
12*x^2 + 4*y
>>> t.derivative(x, Integer(2), y)
4
>>> t.derivative(y)
2*x^2 + 2*y
```

If the function depends on only one variable, you may omit the variable. Giving just a number (for the order of the derivative) also works:

```
sage: f(x) = x^3 + sin(x)
sage: f.derivative()
x |--> 3*x^2 + cos(x)
sage: f.derivative(2)
x |--> 6*x - sin(x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(3) + sin(x)).
↪function(x)
>>> f.derivative()
x |--> 3*x^2 + cos(x)
>>> f.derivative(Integer(2))
x |--> 6*x - sin(x)
```

Some expressions can't be cleanly differentiated by the chain rule:

```
sage: _ = var('x', domain='real')
sage: _ = var('w z')
sage: (x^z).conjugate().diff(x)
conjugate(x^(z - 1))*conjugate(z)
sage: (w^z).conjugate().diff(w)
w^(z - 1)*z*D[0](conjugate)(w^z)
sage: atanh(x).real_part().diff(x)
-1/(x^2 - 1)
sage: atanh(x).imag_part().diff(x)
0
sage: atanh(w).real_part().diff(w)
-D[0](real_part)(arctanh(w))/(w^2 - 1)
sage: atanh(w).imag_part().diff(w)
-D[0](imag_part)(arctanh(w))/(w^2 - 1)
sage: abs(log(x)).diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
sage: abs(log(z)).diff(z)
```

(continues on next page)

(continued from previous page)

```
1/2*(conjugate(log(z))/z + log(z)/conjugate(z))/abs(log(z))
sage: forget()

sage: t = sin(x+y^2)*tan(x*y)
sage: t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
sage: t.derivative(y)
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)
```

```
>>> from sage.all import *
>>> _ = var('x', domain='real')
>>> _ = var('w z')
>>> (x**z).conjugate().diff(x)
conjugate(x^(z - 1))*conjugate(z)
>>> (w**z).conjugate().diff(w)
w^(z - 1)*z*D[0](conjugate)(w^z)
>>> atanh(x).real_part().diff(x)
-1/(x^2 - 1)
>>> atanh(x).imag_part().diff(x)
0
>>> atanh(w).real_part().diff(w)
-D[0](real_part)(arctanh(w))/(w^2 - 1)
>>> atanh(w).imag_part().diff(w)
-D[0](imag_part)(arctanh(w))/(w^2 - 1)
>>> abs(log(x)).diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
>>> abs(log(z)).diff(z)
1/2*(conjugate(log(z))/z + log(z)/conjugate(z))/abs(log(z))
>>> forget()

>>> t = sin(x+y**Integer(2))*tan(x*y)
>>> t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
>>> t.derivative(y)
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)
```

```
sage: h = sin(x)/cos(x)
sage: derivative(h,x,x,x)
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
sage: derivative(h,x,3)
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
```

```
>>> from sage.all import *
>>> h = sin(x)/cos(x)
>>> derivative(h,x,x,x)
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
>>> derivative(h,x,Integer(3))
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
```

```
sage: var('x, y')
(x, y)
```

(continues on next page)

(continued from previous page)

```
sage: u = (sin(x) + cos(y))*(cos(x) - sin(y))
sage: derivative(u,x,y)
-cos(x)*cos(y) + sin(x)*sin(y)
sage: f = ((x^2+1)/(x^2-1))^(1/4)
sage: g = derivative(f, x); g # this is a complex expression
-1/2*((x^2 + 1)*x/(x^2 - 1)^2 - x/(x^2 - 1))/((x^2 + 1)/(x^2 - 1))^(3/4)
sage: g.factor()
-x/((x + 1)^2*(x - 1)^2*((x^2 + 1)/(x^2 - 1))^(3/4))
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> u = (sin(x) + cos(y))*(cos(x) - sin(y))
>>> derivative(u,x,y)
-cos(x)*cos(y) + sin(x)*sin(y)
>>> f = ((x**Integer(2)+Integer(1))/(x**Integer(2)-Integer(1)))**(Integer(1)/
↪Integer(4))
>>> g = derivative(f, x); g # this is a complex expression
-1/2*((x^2 + 1)*x/(x^2 - 1)^2 - x/(x^2 - 1))/((x^2 + 1)/(x^2 - 1))^(3/4)
>>> g.factor()
-x/((x + 1)^2*(x - 1)^2*((x^2 + 1)/(x^2 - 1))^(3/4))
```

```
sage: y = var('y')
sage: f = y^(sin(x))
sage: derivative(f, x)
y^sin(x)*cos(x)*log(y)
```

```
>>> from sage.all import *
>>> y = var('y')
>>> f = y**(sin(x))
>>> derivative(f, x)
y^sin(x)*cos(x)*log(y)
```

```
sage: g(x) = sqrt(5-2*x)
sage: g_3 = derivative(g, x, 3); g_3(2)
-3
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); g = symbolic_expression(sqrt(Integer(5)-Integer(2)*x)).
↪function(x)
>>> g_3 = derivative(g, x, Integer(3)); g_3(Integer(2))
-3
```

```
sage: f = x*e^(-x)
sage: derivative(f, 100)
x*e^(-x) - 100*e^(-x)
```

```
>>> from sage.all import *
>>> f = x*e**(-x)
>>> derivative(f, Integer(100))
```

(continues on next page)

(continued from previous page)

```
x*e^(-x) - 100*e^(-x)
```

```
sage: g = 1/(sqrt((x^2-1)*(x+5)^6))
sage: derivative(g, x)
-((x + 5)^6*x + 3*(x^2 - 1)*(x + 5)^5)/((x^2 - 1)*(x + 5)^6)^(3/2)
```

```
>>> from sage.all import *
>>> g = Integer(1)/(sqrt((x**Integer(2)-
↪Integer(1))*(x+Integer(5))**Integer(6)))
>>> derivative(g, x)
-((x + 5)^6*x + 3*(x^2 - 1)*(x + 5)^5)/((x^2 - 1)*(x + 5)^6)^(3/2)
```

differentiate(**args*)

Return the derivative of this expressions with respect to the variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global `derivative()` function for more details.

See also

This is implemented in the `_derivative` method (see the source code).

EXAMPLES:

```
sage: var("x y")
(x, y)
sage: t = (x^2+y)^2
sage: t.derivative(x)
4*(x^2 + y)*x
sage: t.derivative(x, 2)
12*x^2 + 4*y
sage: t.derivative(x, 2, y)
4
sage: t.derivative(y)
2*x^2 + 2*y
```

```
>>> from sage.all import *
>>> var("x y")
(x, y)
>>> t = (x**Integer(2)+y)**Integer(2)
>>> t.derivative(x)
4*(x^2 + y)*x
>>> t.derivative(x, Integer(2))
12*x^2 + 4*y
>>> t.derivative(x, Integer(2), y)
4
>>> t.derivative(y)
2*x^2 + 2*y
```

If the function depends on only one variable, you may omit the variable. Giving just a number (for the order of the derivative) also works:

```
sage: f(x) = x^3 + sin(x)
sage: f.derivative()
x |--> 3*x^2 + cos(x)
sage: f.derivative(2)
x |--> 6*x - sin(x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(x**Integer(3) + sin(x)).
↪function(x)
>>> f.derivative()
x |--> 3*x^2 + cos(x)
>>> f.derivative(Integer(2))
x |--> 6*x - sin(x)
```

Some expressions can't be cleanly differentiated by the chain rule:

```
sage: _ = var('x', domain='real')
sage: _ = var('w z')
sage: (x^z).conjugate().diff(x)
conjugate(x^(z - 1))*conjugate(z)
sage: (w^z).conjugate().diff(w)
w^(z - 1)*z*D[0](conjugate)(w^z)
sage: atanh(x).real_part().diff(x)
-1/(x^2 - 1)
sage: atanh(x).imag_part().diff(x)
0
sage: atanh(w).real_part().diff(w)
-D[0](real_part)(arctanh(w))/(w^2 - 1)
sage: atanh(w).imag_part().diff(w)
-D[0](imag_part)(arctanh(w))/(w^2 - 1)
sage: abs(log(x)).diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
sage: abs(log(z)).diff(z)
1/2*(conjugate(log(z))/z + log(z)/conjugate(z))/abs(log(z))
sage: forget()

sage: t = sin(x+y^2)*tan(x*y)
sage: t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
sage: t.derivative(y)
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)
```

```
>>> from sage.all import *
>>> _ = var('x', domain='real')
>>> _ = var('w z')
>>> (x**z).conjugate().diff(x)
conjugate(x^(z - 1))*conjugate(z)
>>> (w**z).conjugate().diff(w)
w^(z - 1)*z*D[0](conjugate)(w^z)
>>> atanh(x).real_part().diff(x)
-1/(x^2 - 1)
>>> atanh(x).imag_part().diff(x)
```

(continues on next page)

(continued from previous page)

```
0
>>> atanh(w).real_part().diff(w)
-D[0](real_part)(arctanh(w))/(w^2 - 1)
>>> atanh(w).imag_part().diff(w)
-D[0](imag_part)(arctanh(w))/(w^2 - 1)
>>> abs(log(x)).diff(x)
1/2*(conjugate(log(x))/x + log(x)/x)/abs(log(x))
>>> abs(log(z)).diff(z)
1/2*(conjugate(log(z))/z + log(z)/conjugate(z))/abs(log(z))
>>> forget()

>>> t = sin(x+y**Integer(2))*tan(x*y)
>>> t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
>>> t.derivative(y)
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)
```

```
sage: h = sin(x)/cos(x)
sage: derivative(h,x,x,x)
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
sage: derivative(h,x,3)
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
```

```
>>> from sage.all import *
>>> h = sin(x)/cos(x)
>>> derivative(h,x,x,x)
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
>>> derivative(h,x,Integer(3))
8*sin(x)^2/cos(x)^2 + 6*sin(x)^4/cos(x)^4 + 2
```

```
sage: var('x, y')
(x, y)
sage: u = (sin(x) + cos(y))*(cos(x) - sin(y))
sage: derivative(u,x,y)
-cos(x)*cos(y) + sin(x)*sin(y)
sage: f = ((x^2+1)/(x^2-1))^(1/4)
sage: g = derivative(f, x); g # this is a complex expression
-1/2*((x^2 + 1)*x/(x^2 - 1)^2 - x/(x^2 - 1))/((x^2 + 1)/(x^2 - 1))^(3/4)
sage: g.factor()
-x/((x + 1)^2*(x - 1)^2*((x^2 + 1)/(x^2 - 1))^(3/4))
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> u = (sin(x) + cos(y))*(cos(x) - sin(y))
>>> derivative(u,x,y)
-cos(x)*cos(y) + sin(x)*sin(y)
>>> f = ((x**Integer(2)+Integer(1))/(x**Integer(2)-Integer(1)))**(Integer(1)/
↪Integer(4))
>>> g = derivative(f, x); g # this is a complex expression
-1/2*((x^2 + 1)*x/(x^2 - 1)^2 - x/(x^2 - 1))/((x^2 + 1)/(x^2 - 1))^(3/4)
```

(continues on next page)

(continued from previous page)

```
>>> g.factor()
-x/((x + 1)^2*(x - 1)^2*((x^2 + 1)/(x^2 - 1))^(3/4))
```

```
sage: y = var('y')
sage: f = y^(sin(x))
sage: derivative(f, x)
y^sin(x)*cos(x)*log(y)
```

```
>>> from sage.all import *
>>> y = var('y')
>>> f = y**(sin(x))
>>> derivative(f, x)
y^sin(x)*cos(x)*log(y)
```

```
sage: g(x) = sqrt(5-2*x)
sage: g_3 = derivative(g, x, 3); g_3(2)
-3
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); g = symbolic_expression(sqrt(Integer(5)-Integer(2)*x)).
↪function(x)
>>> g_3 = derivative(g, x, Integer(3)); g_3(Integer(2))
-3
```

```
sage: f = x*e^(-x)
sage: derivative(f, 100)
x*e^(-x) - 100*e^(-x)
```

```
>>> from sage.all import *
>>> f = x*e**(-x)
>>> derivative(f, Integer(100))
x*e^(-x) - 100*e^(-x)
```

```
sage: g = 1/(sqrt((x^2-1)*(x+5)^6))
sage: derivative(g, x)
-((x + 5)^6*x + 3*(x^2 - 1)*(x + 5)^5)/((x^2 - 1)*(x + 5)^6)^(3/2)
```

```
>>> from sage.all import *
>>> g = Integer(1)/(sqrt((x**Integer(2)-
↪Integer(1))*(x+Integer(5))**Integer(6)))
>>> derivative(g, x)
-((x + 5)^6*x + 3*(x^2 - 1)*(x + 5)^5)/((x^2 - 1)*(x + 5)^6)^(3/2)
```

distribute(*recursive=True*)

Distribute some indexed operators over similar operators in order to allow further groupings or simplifications.

Implemented cases (so far):

- Symbolic sum of a sum ==> sum of symbolic sums
- Integral (definite or not) of a sum ==> sum of integrals.

- Symbolic product of a product ==> product of symbolic products.

INPUT:

- `recursive` – (default: `True`) the distribution proceeds along the subtrees of the expression

AUTHORS:

- Emmanuel Charpentier, Ralf Stephan (05-2017)

divide_both_sides(*x*, *checksign=None*)

Return a relation obtained by dividing both sides of this relation by `x`.

Note

The `checksign` keyword argument is currently ignored and is included for backward compatibility reasons only.

EXAMPLES:

```
sage: theta = var('theta')
sage: eqn =   (x^3 + theta < sin(x*theta))
sage: eqn.divide_both_sides(theta, checksign=False)
(x^3 + theta)/theta < sin(theta*x)/theta
sage: eqn.divide_both_sides(theta)
(x^3 + theta)/theta < sin(theta*x)/theta
sage: eqn/theta
(x^3 + theta)/theta < sin(theta*x)/theta
```

```
>>> from sage.all import *
>>> theta = var('theta')
>>> eqn =   (x**Integer(3) + theta < sin(x*theta))
>>> eqn.divide_both_sides(theta, checksign=False)
(x^3 + theta)/theta < sin(theta*x)/theta
>>> eqn.divide_both_sides(theta)
(x^3 + theta)/theta < sin(theta*x)/theta
>>> eqn/theta
(x^3 + theta)/theta < sin(theta*x)/theta
```

exp(*hold=False*)

Return exponential function of `self`, i.e., e to the power of `self`.

EXAMPLES:

```
sage: x.exp()
e^x
sage: SR(0).exp()
1
sage: SR(1/2).exp()
e^(1/2)
sage: SR(0.5).exp()
1.64872127070013
sage: math.exp(0.5)
1.6487212707001282
```

(continues on next page)

(continued from previous page)

```
sage: SR(0.5).exp().log()
0.500000000000000
sage: (pi*I).exp()
-1
```

```
>>> from sage.all import *
>>> x.exp()
e^x
>>> SR(Integer(0)).exp()
1
>>> SR(Integer(1)/Integer(2)).exp()
e^(1/2)
>>> SR(RealNumber('0.5')).exp()
1.64872127070013
>>> math.exp(RealNumber('0.5'))
1.6487212707001282

>>> SR(RealNumber('0.5')).exp().log()
0.500000000000000
>>> (pi*I).exp()
-1
```

To prevent automatic evaluation use the `hold` argument:

```
sage: (pi*I).exp(hold=True)
e^(I*pi)
```

```
>>> from sage.all import *
>>> (pi*I).exp(hold=True)
e^(I*pi)
```

This also works using functional notation:

```
sage: exp(I*pi,hold=True)
e^(I*pi)
sage: exp(I*pi)
-1
```

```
>>> from sage.all import *
>>> exp(I*pi,hold=True)
e^(I*pi)
>>> exp(I*pi)
-1
```

To then evaluate again, we use *unhold()*:

```
sage: a = (pi*I).exp(hold=True); a.unhold()
-1
```

```
>>> from sage.all import *
>>> a = (pi*I).exp(hold=True); a.unhold()
-1
```

expand(*side=None*)

Expand this symbolic expression. Products of sums and exponentiated sums are multiplied out, numerators of rational expressions which are sums are split into their respective terms, and multiplications are distributed over addition at all levels.

EXAMPLES:

We expand the expression $(x - y)^5$ using both method and functional notation.

```
sage: x,y = var('x,y')
sage: a = (x-y)^5
sage: a.expand()
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
sage: expand(a)
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> a = (x-y)**Integer(5)
>>> a.expand()
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
>>> expand(a)
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
```

We expand some other expressions:

```
sage: expand((x-1)^3/(y-1))
x^3/(y - 1) - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)
sage: expand((x+sin((x+y)^2))^2)
x^2 + 2*x*sin(x^2 + 2*x*y + y^2) + sin(x^2 + 2*x*y + y^2)^2
```

```
>>> from sage.all import *
>>> expand((x-Integer(1))**Integer(3)/(y-Integer(1)))
x^3/(y - 1) - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)
>>> expand((x+sin((x+y)**Integer(2)))**Integer(2))
x^2 + 2*x*sin(x^2 + 2*x*y + y^2) + sin(x^2 + 2*x*y + y^2)^2
```

Observe that `expand()` also expands function arguments:

```
sage: f(x) = function('f')(x)
sage: fx = f(x*(x+1)); fx
f((x + 1)*x)
sage: fx.expand()
f(x^2 + x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(function('f')(x)).function(x)
>>> fx = f(x*(x+Integer(1))); fx
f((x + 1)*x)
>>> fx.expand()
f(x^2 + x)
```

We can expand individual sides of a relation:

```
sage: a = (16*x-13)^2 == (3*x+5)^2/2
sage: a.expand()
256*x^2 - 416*x + 169 == 9/2*x^2 + 15*x + 25/2
sage: a.expand('left')
256*x^2 - 416*x + 169 == 1/2*(3*x + 5)^2
sage: a.expand('right')
(16*x - 13)^2 == 9/2*x^2 + 15*x + 25/2
```

```
>>> from sage.all import *
>>> a = (Integer(16)*x-Integer(13))**Integer(2) ==␣
↪(Integer(3)*x+Integer(5))**Integer(2)/Integer(2)
>>> a.expand()
256*x^2 - 416*x + 169 == 9/2*x^2 + 15*x + 25/2
>>> a.expand('left')
256*x^2 - 416*x + 169 == 1/2*(3*x + 5)^2
>>> a.expand('right')
(16*x - 13)^2 == 9/2*x^2 + 15*x + 25/2
```

expand_log(*algorithm='products'*)

Simplify symbolic expression, which can contain logs.

Expands logarithms of powers, logarithms of products and logarithms of quotients. The option `algorithm` specifies which expression types should be expanded.

INPUT:

- `self` – expression to be simplified
- `algorithm` – (default: `'products'`) governs which expression is expanded. Possible values are
  - `'nothing'` (no expansion),
  - `'powers'` (log(a^r) is expanded),
  - `'products'` (like 'powers' and also log(a*b) are expanded),
  - `'all'` (all possible expansion).

  See also examples below.

DETAILS: This uses the Maxima simplifier and sets `logexpand` option for this simplifier. From the Maxima documentation: "Logexpand:true causes log(a^b) to become b*log(a). If it is set to all, log(a*b) will also simplify to log(a)+log(b). If it is set to super, then log(a/b) will also simplify to log(a)-log(b) for rational numbers a/b, a#1. (log(1/b), for integer b, always simplifies.) If it is set to false, all of these simplifications will be turned off. "

ALIAS: *log_expand()* and *expand_log()* are the same

EXAMPLES:

By default powers and products (and quotients) are expanded, but not quotients of integers:

```
sage: (log(3/4*x^pi)).log_expand()
pi*log(x) + log(3/4)
```

```
>>> from sage.all import *
>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand()
pi*log(x) + log(3/4)
```

To expand also log(3/4) use `algorithm='all'`:

```
sage: (log(3/4*x^pi)).log_expand('all')
pi*log(x) + log(3) - 2*log(2)
```

```
>>> from sage.all import *
>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand('all')
pi*log(x) + log(3) - 2*log(2)
```

To expand only the power use `algorithm='powers'`.:

```
sage: (log(x^6)).log_expand('powers')
6*log(x)
```

```
>>> from sage.all import *
>>> (log(x**Integer(6))).log_expand('powers')
6*log(x)
```

The expression `log((3*x)^6)` is not expanded with `algorithm='powers'`, since it is converted into product first:

```
sage: (log((3*x)^6)).log_expand('powers')
log(729*x^6)
```

```
>>> from sage.all import *
>>> (log((Integer(3)*x)**Integer(6))).log_expand('powers')
log(729*x^6)
```

This shows that the option `algorithm` from the previous call has no influence to future calls (we changed some default Maxima flag, and have to ensure that this flag has been restored):

```
sage: (log(3/4*x^pi)).log_expand()
pi*log(x) + log(3/4)

sage: (log(3/4*x^pi)).log_expand('all')
pi*log(x) + log(3) - 2*log(2)

sage: (log(3/4*x^pi)).log_expand()
pi*log(x) + log(3/4)
```

```
>>> from sage.all import *
>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand()
pi*log(x) + log(3/4)

>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand('all')
pi*log(x) + log(3) - 2*log(2)

>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand()
pi*log(x) + log(3/4)
```

AUTHORS:

- Robert Marik (11-2009)

expand_rational(*side=None*)

Expand this symbolic expression. Products of sums and exponentiated sums are multiplied out, numerators of rational expressions which are sums are split into their respective terms, and multiplications are distributed over addition at all levels.

EXAMPLES:

We expand the expression $(x - y)^5$ using both method and functional notation.

```
sage: x,y = var('x,y')
sage: a = (x-y)^5
sage: a.expand()
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
sage: expand(a)
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> a = (x-y)**Integer(5)
>>> a.expand()
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
>>> expand(a)
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
```

We expand some other expressions:

```
sage: expand((x-1)^3/(y-1))
x^3/(y - 1) - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)
sage: expand((x+sin((x+y)^2))^2)
x^2 + 2*x*sin(x^2 + 2*x*y + y^2) + sin(x^2 + 2*x*y + y^2)^2
```

```
>>> from sage.all import *
>>> expand((x-Integer(1))**Integer(3)/(y-Integer(1)))
x^3/(y - 1) - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)
>>> expand((x+sin((x+y)**Integer(2)))**Integer(2))
x^2 + 2*x*sin(x^2 + 2*x*y + y^2) + sin(x^2 + 2*x*y + y^2)^2
```

Observe that `expand()` also expands function arguments:

```
sage: f(x) = function('f')(x)
sage: fx = f(x*(x+1)); fx
f((x + 1)*x)
sage: fx.expand()
f(x^2 + x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(function('f')(x)).function(x)
>>> fx = f(x*(x+Integer(1))); fx
f((x + 1)*x)
>>> fx.expand()
f(x^2 + x)
```

We can expand individual sides of a relation:

```
sage: a = (16*x-13)^2 == (3*x+5)^2/2
sage: a.expand()
256*x^2 - 416*x + 169 == 9/2*x^2 + 15*x + 25/2
sage: a.expand('left')
256*x^2 - 416*x + 169 == 1/2*(3*x + 5)^2
sage: a.expand('right')
(16*x - 13)^2 == 9/2*x^2 + 15*x + 25/2
```

```
>>> from sage.all import *
>>> a = (Integer(16)*x-Integer(13))**Integer(2) ==␣
↪(Integer(3)*x+Integer(5))**Integer(2)/Integer(2)
>>> a.expand()
256*x^2 - 416*x + 169 == 9/2*x^2 + 15*x + 25/2
>>> a.expand('left')
256*x^2 - 416*x + 169 == 1/2*(3*x + 5)^2
>>> a.expand('right')
(16*x - 13)^2 == 9/2*x^2 + 15*x + 25/2
```

expand_sum()

For every symbolic sum in the given expression, try to expand it, symbolically or numerically.

While symbolic sum expressions with constant limits are evaluated immediately on the command line, unevaluated sums of this kind can result from, e.g., substitution of limit variables.

INPUT:

- `self` – symbolic expression

EXAMPLES:

```
sage: (k,n) = var('k,n')
sage: ex = sum(abs(-k*k+n),k,1,n)(n=8); ex
sum(abs(-k^2 + 8), k, 1, 8)
sage: ex.expand_sum()
162
sage: f(x,k) = sum((2/n)*(sin(n*x)*(-1)^(n+1)), n, 1, k)
sage: f(x,2)
-2*sum((-1)^n*sin(n*x)/n, n, 1, 2)
sage: f(x,2).expand_sum()
-sin(2*x) + 2*sin(x)
```

```
>>> from sage.all import *
>>> (k,n) = var('k,n')
>>> ex = sum(abs(-k*k+n),k,Integer(1),n)(n=Integer(8)); ex
sum(abs(-k^2 + 8), k, 1, 8)
>>> ex.expand_sum()
162
>>> __tmp__=var("x,k"); f = symbolic_expression(sum((Integer(2)/
↪n)*(sin(n*x)*(-Integer(1))**(n+Integer(1))), n, Integer(1), k)).function(x,
↪k)
>>> f(x,Integer(2))
-2*sum((-1)^n*sin(n*x)/n, n, 1, 2)
>>> f(x,Integer(2)).expand_sum()
-sin(2*x) + 2*sin(x)
```

We can use this to do floating-point approximation as well:

```
sage: (k,n) = var('k,n')
sage: f(n)=sum(sqrt(abs(-k*k+n)),k,1,n)
sage: f(n=8)
sum(sqrt(abs(-k^2 + 8)), k, 1, 8)
sage: f(8).expand_sum()
sqrt(41) + sqrt(17) + 2*sqrt(14) + 3*sqrt(7) + 2*sqrt(2) + 3
sage: f(8).expand_sum().n()
31.7752256945384
```

```
>>> from sage.all import *
>>> (k,n) = var('k,n')
>>> __tmp__=var("n"); f = symbolic_expression(sum(sqrt(abs(-k*k+n)),k,
↪Integer(1),n)).function(n)
>>> f(n=Integer(8))
sum(sqrt(abs(-k^2 + 8)), k, 1, 8)
>>> f(Integer(8)).expand_sum()
sqrt(41) + sqrt(17) + 2*sqrt(14) + 3*sqrt(7) + 2*sqrt(2) + 3
>>> f(Integer(8)).expand_sum().n()
31.7752256945384
```

See Issue #9424 for making the following no longer raise an error:

```
sage: f(8).n()
31.7752256945384
```

```
>>> from sage.all import *
>>> f(Integer(8)).n()
31.7752256945384
```

expand_trig(*full=False*, *half_angles=False*, *plus=True*, *times=True*)

Expand trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in `self`.

For best results, `self` should already be expanded.

INPUT:

- `full` – boolean (default: `False`); to enhance user control of simplification, this function expands only one level at a time by default, expanding sums of angles or multiple angles. To obtain full expansion into sines and cosines immediately, set the optional parameter full to `True`.
- `half_angles` – boolean (default: `False`); if `True`, causes half-angles to be simplified away
- `plus` – boolean (default: `True`); controls the sum rule. Expansion of sums (e.g. $\sin(x + y)$) will take place only if `plus` is `True`.
- `times` – boolean (default: `True`); controls the product rule, expansion of products (e.g. $\sin(2x)$) will take place only if `times` is `True`.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: sin(5*x).expand_trig()
5*cos(x)^4*sin(x) - 10*cos(x)^2*sin(x)^3 + sin(x)^5
```

(continues on next page)

(continued from previous page)

```
sage: cos(2*x + var('y')).expand_trig()
cos(2*x)*cos(y) - sin(2*x)*sin(y)
```

```
>>> from sage.all import *
>>> sin(Integer(5)*x).expand_trig()
5*cos(x)^4*sin(x) - 10*cos(x)^2*sin(x)^3 + sin(x)^5
>>> cos(Integer(2)*x + var('y')).expand_trig()
cos(2*x)*cos(y) - sin(2*x)*sin(y)
```

We illustrate various options to this function:

```
sage: f = sin(sin(3*cos(2*x))*x)
sage: f.expand_trig()
sin((3*cos(cos(2*x))^2*sin(cos(2*x)) - sin(cos(2*x))^3)*x)
sage: f.expand_trig(full=True)
sin((3*(cos(cos(x)^2)*cos(sin(x)^2)
      + sin(cos(x)^2)*sin(sin(x)^2))^2*(cos(sin(x)^2)*sin(cos(x)^2)
                                        - cos(cos(x)^2)*sin(sin(x)^2))
      - (cos(sin(x)^2)*sin(cos(x)^2) - cos(cos(x)^2)*sin(sin(x)^2))^3)*x)
sage: sin(2*x).expand_trig(times=False)
sin(2*x)
sage: sin(2*x).expand_trig(times=True)
2*cos(x)*sin(x)
sage: sin(2 + x).expand_trig(plus=False)
sin(x + 2)
sage: sin(2 + x).expand_trig(plus=True)
cos(x)*sin(2) + cos(2)*sin(x)
sage: sin(x/2).expand_trig(half_angles=False)
sin(1/2*x)
sage: sin(x/2).expand_trig(half_angles=True)
(-1)^floor(1/2*x/pi)*sqrt(-1/2*cos(x) + 1/2)
```

```
>>> from sage.all import *
>>> f = sin(sin(Integer(3)*cos(Integer(2)*x))*x)
>>> f.expand_trig()
sin((3*cos(cos(2*x))^2*sin(cos(2*x)) - sin(cos(2*x))^3)*x)
>>> f.expand_trig(full=True)
sin((3*(cos(cos(x)^2)*cos(sin(x)^2)
      + sin(cos(x)^2)*sin(sin(x)^2))^2*(cos(sin(x)^2)*sin(cos(x)^2)
                                        - cos(cos(x)^2)*sin(sin(x)^2))
      - (cos(sin(x)^2)*sin(cos(x)^2) - cos(cos(x)^2)*sin(sin(x)^2))^3)*x)
>>> sin(Integer(2)*x).expand_trig(times=False)
sin(2*x)
>>> sin(Integer(2)*x).expand_trig(times=True)
2*cos(x)*sin(x)
>>> sin(Integer(2) + x).expand_trig(plus=False)
sin(x + 2)
>>> sin(Integer(2) + x).expand_trig(plus=True)
cos(x)*sin(2) + cos(2)*sin(x)
>>> sin(x/Integer(2)).expand_trig(half_angles=False)
sin(1/2*x)
>>> sin(x/Integer(2)).expand_trig(half_angles=True)
```

(continues on next page)

(continued from previous page)

```
(-1)^floor(1/2*x/pi)*sqrt(-1/2*cos(x) + 1/2)
```

If the expression contains terms which are factored, we expand first:

```
sage: (x, k1, k2) = var('x, k1, k2')
sage: cos((k1-k2)*x).expand().expand_trig()
cos(k1*x)*cos(k2*x) + sin(k1*x)*sin(k2*x)
```

```
>>> from sage.all import *
>>> (x, k1, k2) = var('x, k1, k2')
>>> cos((k1-k2)*x).expand().expand_trig()
cos(k1*x)*cos(k2*x) + sin(k1*x)*sin(k2*x)
```

ALIAS:

trig_expand() and *expand_trig()* are the same

exponentialize()

Return this symbolic expression with all circular and hyperbolic functions replaced by their respective exponential expressions.

EXAMPLES:

```
sage: x = SR.var("x")
sage: sin(x).exponentialize()
-1/2*I*e^(I*x) + 1/2*I*e^(-I*x)
sage: sec(x).exponentialize()
2/(e^(I*x) + e^(-I*x))
sage: tan(x).exponentialize()
(-I*e^(I*x) + I*e^(-I*x))/(e^(I*x) + e^(-I*x))
sage: sinh(x).exponentialize()
-1/2*e^(-x) + 1/2*e^x
sage: sech(x).exponentialize()
2/(e^(-x) + e^x)
sage: tanh(x).exponentialize()
-(e^(-x) - e^x)/(e^(-x) + e^x)
```

```
>>> from sage.all import *
>>> x = SR.var("x")
>>> sin(x).exponentialize()
-1/2*I*e^(I*x) + 1/2*I*e^(-I*x)
>>> sec(x).exponentialize()
2/(e^(I*x) + e^(-I*x))
>>> tan(x).exponentialize()
(-I*e^(I*x) + I*e^(-I*x))/(e^(I*x) + e^(-I*x))
>>> sinh(x).exponentialize()
-1/2*e^(-x) + 1/2*e^x
>>> sech(x).exponentialize()
2/(e^(-x) + e^x)
>>> tanh(x).exponentialize()
-(e^(-x) - e^x)/(e^(-x) + e^x)
```

factor(*dontfactor=None*)

Factor the expression, containing any number of variables or functions, into factors irreducible over the integers.

INPUT:

- `self` – a symbolic expression
- `dontfactor` – list (default: `[]`); a list of variables with respect to which factoring is not to occur. Factoring also will not take place with respect to any variables which are less important (using the variable ordering assumed for CRE form) than those on the 'dontfactor' list.

EXAMPLES:

```
sage: x,y,z = var('x, y, z')
sage: (x^3-y^3).factor()
(x^2 + x*y + y^2)*(x - y)
sage: factor(-8*y - 4*x + z^2*(2*y + x))
(x + 2*y)*(z + 2)*(z - 2)
sage: f = -1 - 2*x - x^2 + y^2 + 2*x*y^2 + x^2*y^2
sage: F = factor(f/(36*(1 + 2*y + y^2)), dontfactor=[x]); F
1/36*(x^2 + 2*x + 1)*(y - 1)/(y + 1)
```

```
>>> from sage.all import *
>>> x,y,z = var('x, y, z')
>>> (x**Integer(3)-y**Integer(3)).factor()
(x^2 + x*y + y^2)*(x - y)
>>> factor(-Integer(8)*y - Integer(4)*x + z**Integer(2)*(Integer(2)*y + x))
(x + 2*y)*(z + 2)*(z - 2)
>>> f = -Integer(1) - Integer(2)*x - x**Integer(2) + y**Integer(2) +␣
↪Integer(2)*x*y**Integer(2) + x**Integer(2)*y**Integer(2)
>>> F = factor(f/(Integer(36)*(Integer(1) + Integer(2)*y + y**Integer(2))),␣
↪dontfactor=[x]); F
1/36*(x^2 + 2*x + 1)*(y - 1)/(y + 1)
```

If you are factoring a polynomial with rational coefficients (and dontfactor is empty) the factorization is done using Singular instead of Maxima, so the following is very fast instead of dreadfully slow:

```
sage: var('x,y')
(x, y)
sage: (x^99 + y^99).factor()
(x^60 + x^57*y^3 - x^51*y^9 - x^48*y^12 + x^42*y^18 + x^39*y^21 -
x^33*y^27 - x^30*y^30 - x^27*y^33 + x^21*y^39 + x^18*y^42 -
x^12*y^48 - x^9*y^51 + x^3*y^57 + y^60)*(x^20 + x^19*y -
x^17*y^3 - x^16*y^4 + x^14*y^6 + x^13*y^7 - x^11*y^9 -
x^10*y^10 - x^9*y^11 + x^7*y^13 + x^6*y^14 - x^4*y^16 -
x^3*y^17 + x*y^19 + y^20)*(x^10 - x^9*y + x^8*y^2 - x^7*y^3 +
x^6*y^4 - x^5*y^5 + x^4*y^6 - x^3*y^7 + x^2*y^8 - x*y^9 +
y^10)*(x^6 - x^3*y^3 + y^6)*(x^2 - x*y + y^2)*(x + y)
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> (x**Integer(99) + y**Integer(99)).factor()
(x^60 + x^57*y^3 - x^51*y^9 - x^48*y^12 + x^42*y^18 + x^39*y^21 -
```

(continues on next page)

(continued from previous page)

```
x^33*y^27 - x^30*y^30 - x^27*y^33 + x^21*y^39 + x^18*y^42 -
x^12*y^48 - x^9*y^51 + x^3*y^57 + y^60)*(x^20 + x^19*y -
x^17*y^3 - x^16*y^4 + x^14*y^6 + x^13*y^7 - x^11*y^9 -
x^10*y^10 - x^9*y^11 + x^7*y^13 + x^6*y^14 - x^4*y^16 -
x^3*y^17 + x*y^19 + y^20)*(x^10 - x^9*y + x^8*y^2 - x^7*y^3 +
x^6*y^4 - x^5*y^5 + x^4*y^6 - x^3*y^7 + x^2*y^8 - x*y^9 +
y^10)*(x^6 - x^3*y^3 + y^6)*(x^2 - x*y + y^2)*(x + y)
```

factor_list(*dontfactor=None*)

Return a list of the factors of self, as computed by the factor command.

INPUT:

- `self` – a symbolic expression
- `dontfactor` – see docs for `factor()`

Note

If you already have a factored expression and just want to get at the individual factors, use the `_factor_list` method instead.

EXAMPLES:

```
sage: var('x, y, z')
(x, y, z)
sage: f = x^3-y^3
sage: f.factor()
(x^2 + x*y + y^2)*(x - y)
```

```
>>> from sage.all import *
>>> var('x, y, z')
(x, y, z)
>>> f = x**Integer(3)-y**Integer(3)
>>> f.factor()
(x^2 + x*y + y^2)*(x - y)
```

Notice that the -1 factor is separated out:

```
sage: f.factor_list()
[(x^2 + x*y + y^2, 1), (x - y, 1)]
```

```
>>> from sage.all import *
>>> f.factor_list()
[(x^2 + x*y + y^2, 1), (x - y, 1)]
```

We factor a fairly straightforward expression:

```
sage: factor(-8*y - 4*x + z^2*(2*y + x)).factor_list()
[(x + 2*y, 1), (z + 2, 1), (z - 2, 1)]
```

```
>>> from sage.all import *
>>> factor(-Integer(8)*y - Integer(4)*x + z**Integer(2)*(Integer(2)*y + x)).
↪factor_list()
[(x + 2*y, 1), (z + 2, 1), (z - 2, 1)]
```

A more complicated example:

```
sage: var('x, u, v')
(x, u, v)
sage: f = expand((2*u*v^2-v^2-4*u^3)^2 * (-u)^3 * (x-sin(x))^3)
sage: f.factor()
-(4*u^3 - 2*u*v^2 + v^2)^2*u^3*(x - sin(x))^3
sage: g = f.factor_list(); g
[(4*u^3 - 2*u*v^2 + v^2, 2), (u, 3), (x - sin(x), 3), (-1, 1)]
```

```
>>> from sage.all import *
>>> var('x, u, v')
(x, u, v)
>>> f = expand((Integer(2)*u*v**Integer(2)-v**Integer(2)-
↪Integer(4)*u**Integer(3))**Integer(2) * (-u)**Integer(3) * (x-
↪sin(x))**Integer(3))
>>> f.factor()
-(4*u^3 - 2*u*v^2 + v^2)^2*u^3*(x - sin(x))^3
>>> g = f.factor_list(); g
[(4*u^3 - 2*u*v^2 + v^2, 2), (u, 3), (x - sin(x), 3), (-1, 1)]
```

This function also works for quotients:

```
sage: f = -1 - 2*x - x^2 + y^2 + 2*x*y^2 + x^2*y^2
sage: g = f/(36*(1 + 2*y + y^2)); g
1/36*(x^2*y^2 + 2*x*y^2 - x^2 + y^2 - 2*x - 1)/(y^2 + 2*y + 1)
sage: g.factor(dontfactor=[x])
1/36*(x^2 + 2*x + 1)*(y - 1)/(y + 1)
sage: g.factor_list(dontfactor=[x])
[(x^2 + 2*x + 1, 1), (y + 1, -1), (y - 1, 1), (1/36, 1)]
```

```
>>> from sage.all import *
>>> f = -Integer(1) - Integer(2)*x - x**Integer(2) + y**Integer(2) +␣
↪Integer(2)*x*y**Integer(2) + x**Integer(2)*y**Integer(2)
>>> g = f/(Integer(36)*(Integer(1) + Integer(2)*y + y**Integer(2))); g
1/36*(x^2*y^2 + 2*x*y^2 - x^2 + y^2 - 2*x - 1)/(y^2 + 2*y + 1)
>>> g.factor(dontfactor=[x])
1/36*(x^2 + 2*x + 1)*(y - 1)/(y + 1)
>>> g.factor_list(dontfactor=[x])
[(x^2 + 2*x + 1, 1), (y + 1, -1), (y - 1, 1), (1/36, 1)]
```

This example also illustrates that the exponents do not have to be integers:

```
sage: f = x^(2*sin(x)) * (x-1)^(sqrt(2)*x); f
(x - 1)^(sqrt(2)*x)*x^(2*sin(x))
sage: f.factor_list()
[(x - 1, sqrt(2)*x), (x, 2*sin(x))]
```

```
>>> from sage.all import *
>>> f = x**(Integer(2)*sin(x)) * (x-Integer(1))**(sqrt(Integer(2))*x); f
(x - 1)^(sqrt(2)*x)*x^(2*sin(x))
>>> f.factor_list()
[(x - 1, sqrt(2)*x), (x, 2*sin(x))]
```

factorial(*hold=False*)

Return the factorial of `self`.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: SR(5).factorial()
120
sage: x.factorial()
factorial(x)
sage: (x^2+y^3).factorial()
factorial(y^3 + x^2)
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> SR(Integer(5)).factorial()
120
>>> x.factorial()
factorial(x)
>>> (x**Integer(2)+y**Integer(3)).factorial()
factorial(y^3 + x^2)
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(5).factorial(hold=True)
factorial(5)
```

```
>>> from sage.all import *
>>> SR(Integer(5)).factorial(hold=True)
factorial(5)
```

This also works using functional notation:

```
sage: factorial(5,hold=True)
factorial(5)
sage: factorial(5)
120
```

```
>>> from sage.all import *
>>> factorial(Integer(5),hold=True)
factorial(5)
>>> factorial(Integer(5))
120
```

To then evaluate again, we use *unhold()*:

```
sage: a = SR(5).factorial(hold=True); a.unhold()
120
```

```
>>> from sage.all import *
>>> a = SR(Integer(5)).factorial(hold=True); a.unhold()
120
```

factorial_simplify()

Simplify by combining expressions with factorials, and by expanding binomials into factorials.

ALIAS: factorial_simplify and simplify_factorial are the same

EXAMPLES:

Some examples are relatively clear:

```
sage: var('n,k')
(n, k)
sage: f = factorial(n+1)/factorial(n); f
factorial(n + 1)/factorial(n)
sage: f.simplify_factorial()
n + 1
```

```
>>> from sage.all import *
>>> var('n,k')
(n, k)
>>> f = factorial(n+Integer(1))/factorial(n); f
factorial(n + 1)/factorial(n)
>>> f.simplify_factorial()
n + 1
```

```
sage: f = factorial(n)*(n+1); f
(n + 1)*factorial(n)
sage: simplify(f)
(n + 1)*factorial(n)
sage: f.simplify_factorial()
factorial(n + 1)
```

```
>>> from sage.all import *
>>> f = factorial(n)*(n+Integer(1)); f
(n + 1)*factorial(n)
>>> simplify(f)
(n + 1)*factorial(n)
>>> f.simplify_factorial()
factorial(n + 1)
```

```
sage: f = binomial(n, k)*factorial(k)*factorial(n-k); f
binomial(n, k)*factorial(k)*factorial(-k + n)
sage: f.simplify_factorial()
factorial(n)
```

```
>>> from sage.all import *
>>> f = binomial(n, k)*factorial(k)*factorial(n-k); f
binomial(n, k)*factorial(k)*factorial(-k + n)
>>> f.simplify_factorial()
factorial(n)
```

A more complicated example, which needs further processing:

```
sage: f = factorial(x)/factorial(x-2)/2 + factorial(x+1)/factorial(x)/2; f
1/2*factorial(x + 1)/factorial(x) + 1/2*factorial(x)/factorial(x - 2)
sage: g = f.simplify_factorial(); g
1/2*(x - 1)*x + 1/2*x + 1/2
sage: g.simplify_rational()
1/2*x^2 + 1/2
```

```
>>> from sage.all import *
>>> f = factorial(x)/factorial(x-Integer(2))/Integer(2) +␣
↪factorial(x+Integer(1))/factorial(x)/Integer(2); f
1/2*factorial(x + 1)/factorial(x) + 1/2*factorial(x)/factorial(x - 2)
>>> g = f.simplify_factorial(); g
1/2*(x - 1)*x + 1/2*x + 1/2
>>> g.simplify_rational()
1/2*x^2 + 1/2
```

find(*pattern*)

Find all occurrences of the given pattern in this expression.

Note that once a subexpression matches the pattern, the search does not extend to subexpressions of it.

EXAMPLES:

```
sage: var('x,y,z,a,b')
(x, y, z, a, b)
sage: w0 = SR.wild(0); w1 = SR.wild(1)

sage: (sin(x)*sin(y)).find(sin(w0))
[sin(y), sin(x)]

sage: ((sin(x)+sin(y))*(a+b)).expand().find(sin(w0))
[sin(y), sin(x)]

sage: (1+x+x^2+x^3).find(x)
[x]
sage: (1+x+x^2+x^3).find(x^w0)
[x^2, x^3]

sage: (1+x+x^2+x^3).find(y)
[]

# subexpressions of a match are not listed
sage: ((x^y)^z).find(w0^w1)
[(x^y)^z]
```

```
>>> from sage.all import *
>>> var('x,y,z,a,b')
(x, y, z, a, b)
>>> w0 = SR.wild(Integer(0)); w1 = SR.wild(Integer(1))

>>> (sin(x)*sin(y)).find(sin(w0))
[sin(y), sin(x)]

>>> ((sin(x)+sin(y))*(a+b)).expand().find(sin(w0))
[sin(y), sin(x)]

>>> (Integer(1)+x+x**Integer(2)+x**Integer(3)).find(x)
[x]
>>> (Integer(1)+x+x**Integer(2)+x**Integer(3)).find(x**w0)
[x^2, x^3]

>>> (Integer(1)+x+x**Integer(2)+x**Integer(3)).find(y)
[]

# subexpressions of a match are not listed
>>> ((x**y)**z).find(w0**w1)
[(x^y)^z]
```

find_local_maximum(*a*, *b*, *var=None*, *tol=1.48e-08*, *maxfun=500*, *imaginary_tolerance=1e-08*)

Numerically find a local maximum of the expression `self` on the interval [a,b] (or [b,a]) along with the point at which the maximum is attained.

See the documentation for *find_local_minimum()* for more details.

EXAMPLES:

```
sage: f = x*cos(x)
sage: f.find_local_maximum(0,5)                                              #␣
↪needs scipy
(0.5610963381910451, 0.8603335890...)
sage: f.find_local_maximum(0,5, tol=0.1, maxfun=10)                          #␣
↪needs scipy
(0.561090323458081..., 0.857926501456...)
```

```
>>> from sage.all import *
>>> f = x*cos(x)
>>> f.find_local_maximum(Integer(0),Integer(5))                              ␣
↪               # needs scipy
(0.5610963381910451, 0.8603335890...)
>>> f.find_local_maximum(Integer(0),Integer(5), tol=RealNumber('0.1'),␣
↪maxfun=Integer(10))                              # needs scipy
(0.561090323458081..., 0.857926501456...)
```

find_local_minimum(*a*, *b*, *var=None*, *tol=1.48e-08*, *maxfun=500*, *imaginary_tolerance=1e-08*)

Numerically find a local minimum of the expression `self` on the interval [a,b] (or [b,a]) and the point at which it attains that minimum. Note that `self` must be a function of (at most) one variable.

INPUT:

- `a` – real number; left endpoint of interval on which to minimize

- b – real number; right endpoint of interval on which to minimize
- var – variable (default: first variable in self); the variable in self to maximize over
- tol – positive real (default: 1.48e-08); the convergence tolerance
- maxfun – natural number (default: 500); maximum function evaluations
- imaginary_tolerance – (default: 1e-8) if an imaginary number arises (due, for example, to numerical issues), this tolerance specifies how large it has to be in magnitude before we raise an error. In other words, imaginary parts smaller than this are ignored when we are expecting a real answer.

OUTPUT:

A tuple (minval, x), where

- minval – float; the minimum value that self takes on in the interval [a,b]
- x – float; the point at which self takes on the minimum value

EXAMPLES:

```
sage: # needs scipy
sage: f = x*cos(x)
sage: f.find_local_minimum(1, 5)
(-3.288371395590..., 3.4256184695...)
sage: f.find_local_minimum(1, 5, tol=1e-3)
(-3.288371361890..., 3.4257507903...)
sage: f.find_local_minimum(1, 5, tol=1e-2, maxfun=10)
(-3.288370845983..., 3.4250840220...)
sage: show(f.plot(0, 20))                                                    #␣
↪needs sage.plot
sage: f.find_local_minimum(1, 15)
(-9.477294259479..., 9.5293344109...)
```

```
>>> from sage.all import *
>>> # needs scipy
>>> f = x*cos(x)
>>> f.find_local_minimum(Integer(1), Integer(5))
(-3.288371395590..., 3.4256184695...)
>>> f.find_local_minimum(Integer(1), Integer(5), tol=RealNumber('1e-3'))
(-3.288371361890..., 3.4257507903...)
>>> f.find_local_minimum(Integer(1), Integer(5), tol=RealNumber('1e-2'),␣
↪maxfun=Integer(10))
(-3.288370845983..., 3.4250840220...)
>>> show(f.plot(Integer(0), Integer(20)))                                    ␣
↪              # needs sage.plot
>>> f.find_local_minimum(Integer(1), Integer(15))
(-9.477294259479..., 9.5293344109...)
```

ALGORITHM:

Uses sage.numerical.optimize.find_local_minimum().

AUTHORS:

- William Stein (2007-12-07)

find_root(*a*, *b*, *var=None*, *xtol=1e-12*, *rtol=8.881784197001252e-16*, *maxiter=100*, *full_output=False*, *imaginary_tolerance=1e-08*)

Numerically find a root of `self` on the closed interval [a,b] (or [b,a]) if possible, where `self` is a function in one variable. Note: this function only works in fixed (machine) precision, it is not possible to get arbitrary precision approximations with it.

INPUT:

- `a, b` – endpoints of the interval
- `var` – (optional) variable
- `xtol, rtol` – the routine converges when a root is known to lie within `xtol` of the value return. Should be nonnegative. The routine modifies this to take into account the relative precision of doubles.
- `maxiter` – integer; if convergence is not achieved in maxiter iterations, an error is raised. Must be nonnegative.
- `full_output` – boolean (default: `False`); if `True`, also return object that contains information about convergence
- `imaginary_tolerance` – (default: `1e-8`) if an imaginary number arises (due, for example, to numerical issues), this tolerance specifies how large it has to be in magnitude before we raise an error. In other words, imaginary parts smaller than this are ignored when we are expecting a real answer.

EXAMPLES:

Note that in this example both f(-2) and f(3) are positive, yet we still find a root in that interval:

```
sage: # needs scipy
sage: f = x^2 - 1
sage: f.find_root(-2, 3)
1.0
sage: f.find_root(-2, 3, x)
1.0
sage: z, result = f.find_root(-2, 3, full_output=True)
sage: result.converged
True
sage: result.flag
'converged'
sage: result.function_calls
11
sage: result.iterations
10
sage: result.root
1.0
```

```
>>> from sage.all import *
>>> # needs scipy
>>> f = x**Integer(2) - Integer(1)
>>> f.find_root(-Integer(2), Integer(3))
1.0
>>> f.find_root(-Integer(2), Integer(3), x)
1.0
>>> z, result = f.find_root(-Integer(2), Integer(3), full_output=True)
>>> result.converged
True
>>> result.flag
'converged'
```

(continues on next page)

(continued from previous page)

```
>>> result.function_calls
11
>>> result.iterations
10
>>> result.root
1.0
```

More examples:

```
sage: (sin(x) + exp(x)).find_root(-10, 10)                                   #␣
↪needs scipy
-0.588532743981862...
sage: sin(x).find_root(-1,1)                                                  #␣
↪needs scipy
0.0
```

```
>>> from sage.all import *
>>> (sin(x) + exp(x)).find_root(-Integer(10), Integer(10))                    ␣
↪              # needs scipy
-0.588532743981862...
>>> sin(x).find_root(-Integer(1),Integer(1))                                  ␣
↪              # needs scipy
0.0
```

This example was fixed along with Issue #4942 - there was an error in the example pi is a root for tan(x), but an asymptote to 1/tan(x) added an example to show handling of both cases:

```
sage: (tan(x)).find_root(3,3.5)                                               #␣
↪needs scipy
3.1415926535...
sage: (1/tan(x)).find_root(3, 3.5)                                            #␣
↪needs scipy
Traceback (most recent call last):
...
NotImplementedError: Brent's method failed to find a zero for f on the␣
↪interval
```

```
>>> from sage.all import *
>>> (tan(x)).find_root(Integer(3),RealNumber('3.5'))                          ␣
↪                   # needs scipy
3.1415926535...
>>> (Integer(1)/tan(x)).find_root(Integer(3), RealNumber('3.5'))              ␣
↪                            # needs scipy
Traceback (most recent call last):
...
NotImplementedError: Brent's method failed to find a zero for f on the␣
↪interval
```

An example with a square root:

```
sage: f = 1 + x + sqrt(x+2); f.find_root(-2,10)                               #␣
↪needs scipy
```

(continues on next page)

(continued from previous page)

```
-1.618033988749895
```

```
>>> from sage.all import *
>>> f = Integer(1) + x + sqrt(x+Integer(2)); f.find_root(-Integer(2),
↪Integer(10))                                  # needs scipy
-1.618033988749895
```

Some examples that Ted Kosan came up with:

```
sage: t = var('t')
sage: v = 0.004*(9600*e^(-(1200*t)) - 2400*e^(-(300*t)))
sage: v.find_root(0, 0.002)                                                   #␣
↪needs scipy
0.001540327067911417...
```

```
>>> from sage.all import *
>>> t = var('t')
>>> v = RealNumber('0.004')*(Integer(9600)*e**(-(Integer(1200)*t)) -␣
↪Integer(2400)*e**(-(Integer(300)*t)))
>>> v.find_root(Integer(0), RealNumber('0.002'))                              ␣
↪                    # needs scipy
0.001540327067911417...
```

With this expression, we can see there is a zero very close to the origin:

```
sage: a = .004*(8*e^(-(300*t)) - 8*e^(-(1200*t)))*(720000*e^(-(300*t)) -␣
↪11520000*e^(-(1200*t))) +.004*(9600*e^(-(1200*t)) - 2400*e^(-(300*t)))^2
sage: show(plot(a, 0, .002), xmin=0, xmax=.002)                               #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> a = RealNumber('.004')*(Integer(8)*e**(-(Integer(300)*t)) -␣
↪Integer(8)*e**(-(Integer(1200)*t)))*(Integer(720000)*e**(-(Integer(300)*t))␣
↪- Integer(11520000)*e**(-(Integer(1200)*t))) +RealNumber('.004
↪')*(Integer(9600)*e**(-(Integer(1200)*t)) - Integer(2400)*e**(-
↪(Integer(300)*t)))**Integer(2)
>>> show(plot(a, Integer(0), RealNumber('.002')), xmin=Integer(0),␣
↪xmax=RealNumber('.002'))                          # needs sage.plot
```

It is easy to approximate with `find_root`:

```
sage: a.find_root(0,0.002)                                                    #␣
↪needs scipy
0.0004110514049349...
```

```
>>> from sage.all import *
>>> a.find_root(Integer(0),RealNumber('0.002'))                               ␣
↪                    # needs scipy
0.0004110514049349...
```

Using solve takes more effort, and even then gives only a solution with free (integer) variables:

```
sage: a.solve(t)
[]
sage: b = a.canonicalize_radical(); b
(46080.0*e^(1800*t) - 576000.0*e^(900*t) + 737280.0)*e^(-2400*t)
sage: b.solve(t)
[]
sage: b.solve(t, to_poly_solve=True)
[t == 1/450*I*pi*z... + 1/900*log(-3/4*sqrt(41) + 25/4),
 t == 1/450*I*pi*z... + 1/900*log(3/4*sqrt(41) + 25/4)]
sage: n(1/900*log(-3/4*sqrt(41) + 25/4))
0.000411051404934985
```

```
>>> from sage.all import *
>>> a.solve(t)
[]
>>> b = a.canonicalize_radical(); b
(46080.0*e^(1800*t) - 576000.0*e^(900*t) + 737280.0)*e^(-2400*t)
>>> b.solve(t)
[]
>>> b.solve(t, to_poly_solve=True)
[t == 1/450*I*pi*z... + 1/900*log(-3/4*sqrt(41) + 25/4),
 t == 1/450*I*pi*z... + 1/900*log(3/4*sqrt(41) + 25/4)]
>>> n(Integer(1)/Integer(900)*log(-Integer(3)/Integer(4)*sqrt(Integer(41)) +␣
↪Integer(25)/Integer(4)))
0.000411051404934985
```

We illustrate that root finding is only implemented in one dimension:

```
sage: x, y = var('x,y')
sage: (x-y).find_root(-2,2)
Traceback (most recent call last):
...
NotImplementedError: root finding currently only implemented in 1 dimension.
```

```
>>> from sage.all import *
>>> x, y = var('x,y')
>>> (x-y).find_root(-Integer(2),Integer(2))
Traceback (most recent call last):
...
NotImplementedError: root finding currently only implemented in 1 dimension.
```

forget()

Forget the given constraint.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: forget()
sage: assume(x>0, y < 2)
sage: assumptions()
[x > 0, y < 2]
sage: forget(y < 2)
```

(continues on next page)

(continued from previous page)

```
sage: assumptions()
[x > 0]
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> forget()
>>> assume(x>Integer(0), y < Integer(2))
>>> assumptions()
[x > 0, y < 2]
>>> forget(y < Integer(2))
>>> assumptions()
[x > 0]
```

fraction(*base_ring*)

Return this expression as element of the algebraic fraction field over the base ring given.

EXAMPLES:

```
sage: fr = (1/x).fraction(ZZ); fr
1/x
sage: parent(fr)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: parent(((pi+sqrt(2)/x).fraction(SR)))
Fraction Field of Univariate Polynomial Ring in x over Symbolic Ring
sage: parent(((pi+sqrt(2))/x).fraction(SR))
Fraction Field of Univariate Polynomial Ring in x over Symbolic Ring
sage: y = var('y')
sage: fr = ((3*x^5 - 5*y^5)^7/(x*y)).fraction(GF(7)); fr
(3*x^35 + 2*y^35)/(x*y)
sage: parent(fr)
Fraction Field of Multivariate Polynomial Ring in x, y over Finite Field of␣
↪size 7
```

```
>>> from sage.all import *
>>> fr = (Integer(1)/x).fraction(ZZ); fr
1/x
>>> parent(fr)
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
>>> parent(((pi+sqrt(Integer(2))/x).fraction(SR)))
Fraction Field of Univariate Polynomial Ring in x over Symbolic Ring
>>> parent(((pi+sqrt(Integer(2)))/x).fraction(SR))
Fraction Field of Univariate Polynomial Ring in x over Symbolic Ring
>>> y = var('y')
>>> fr = ((Integer(3)*x**Integer(5) - Integer(5)*y**Integer(5))**Integer(7)/
↪(x*y)).fraction(GF(Integer(7))); fr
(3*x^35 + 2*y^35)/(x*y)
>>> parent(fr)
Fraction Field of Multivariate Polynomial Ring in x, y over Finite Field of␣
↪size 7
```

free_variables()

Return sorted tuple of unbound variables that occur in this expression.

EXAMPLES:

```
sage: (x,y,z) = var('x,y,z')
sage: (x+y).free_variables()
(x, y)
sage: (2*x).free_variables()
(x,)
sage: (x^y).free_variables()
(x, y)
sage: sin(x+y^z).free_variables()
(x, y, z)
sage: _ = function('f')
sage: e = limit( f(x,y), x=0 ); e
limit(f(x, y), x, 0)
sage: e.free_variables()
(y,)
```

```
>>> from sage.all import *
>>> (x,y,z) = var('x,y,z')
>>> (x+y).free_variables()
(x, y)
>>> (Integer(2)*x).free_variables()
(x,)
>>> (x**y).free_variables()
(x, y)
>>> sin(x+y**z).free_variables()
(x, y, z)
>>> _ = function('f')
>>> e = limit( f(x,y), x=Integer(0) ); e
limit(f(x, y), x, 0)
>>> e.free_variables()
(y,)
```

full_simplify()

Apply *simplify_factorial()*, *simplify_rectform()*, *simplify_trig()*, *simplify_rational()*, and then *expand_sum()* to `self` (in that order).

ALIAS: `simplify_full` and `full_simplify` are the same.

EXAMPLES:

```
sage: f = sin(x)^2 + cos(x)^2
sage: f.simplify_full()
1
```

```
>>> from sage.all import *
>>> f = sin(x)**Integer(2) + cos(x)**Integer(2)
>>> f.simplify_full()
1
```

```
sage: f = sin(x/(x^2 + x))
sage: f.simplify_full()
sin(1/(x + 1))
```

```
>>> from sage.all import *
>>> f = sin(x/(x**Integer(2) + x))
>>> f.simplify_full()
sin(1/(x + 1))
```

```
sage: var('n,k')
(n, k)
sage: f = binomial(n,k)*factorial(k)*factorial(n-k)
sage: f.simplify_full()
factorial(n)
```

```
>>> from sage.all import *
>>> var('n,k')
(n, k)
>>> f = binomial(n,k)*factorial(k)*factorial(n-k)
>>> f.simplify_full()
factorial(n)
```

function(**args*)

Return a callable symbolic expression with the given variables.

EXAMPLES:

We will use several symbolic variables in the examples below:

```
sage: var('x, y, z, t, a, w, n')
(x, y, z, t, a, w, n)
```

```
>>> from sage.all import *
>>> var('x, y, z, t, a, w, n')
(x, y, z, t, a, w, n)
```

```
sage: u = sin(x) + x*cos(y)
sage: g = u.function(x,y)
sage: g(x,y)
x*cos(y) + sin(x)
sage: g(t,z)
t*cos(z) + sin(t)
sage: g(x^2, x^y)
x^2*cos(x^y) + sin(x^2)
```

```
>>> from sage.all import *
>>> u = sin(x) + x*cos(y)
>>> g = u.function(x,y)
>>> g(x,y)
x*cos(y) + sin(x)
>>> g(t,z)
t*cos(z) + sin(t)
>>> g(x**Integer(2), x**y)
x^2*cos(x^y) + sin(x^2)
```

```
sage: f = (x^2 + sin(a*w)).function(a,x,w); f
(a, x, w) |--> x^2 + sin(a*w)
sage: f(1,2,3)
sin(3) + 4
```

```
>>> from sage.all import *
>>> f = (x**Integer(2) + sin(a*w)).function(a,x,w); f
(a, x, w) |--> x^2 + sin(a*w)
>>> f(Integer(1),Integer(2),Integer(3))
sin(3) + 4
```

Using the `function()` method we can obtain the above function f, but viewed as a function of different variables:

```
sage: h = f.function(w,a); h
(w, a) |--> x^2 + sin(a*w)
```

```
>>> from sage.all import *
>>> h = f.function(w,a); h
(w, a) |--> x^2 + sin(a*w)
```

This notation also works:

```
sage: h(w,a) = f
sage: h
(w, a) |--> x^2 + sin(a*w)
```

```
>>> from sage.all import *
>>> __tmp__=var("w,a"); h = symbolic_expression(f).function(w,a)
>>> h
(w, a) |--> x^2 + sin(a*w)
```

You can even make a symbolic expression f into a function by writing `f(x,y) = f`:

```
sage: f = x^n + y^n; f
x^n + y^n
sage: f(x,y) = f
sage: f
(x, y) |--> x^n + y^n
sage: f(2,3)
3^n + 2^n
```

```
>>> from sage.all import *
>>> f = x**n + y**n; f
x^n + y^n
>>> __tmp__=var("x,y"); f = symbolic_expression(f).function(x,y)
>>> f
(x, y) |--> x^n + y^n
>>> f(Integer(2),Integer(3))
3^n + 2^n
```

gamma(*hold=False*)

Return the Gamma function evaluated at `self`.

EXAMPLES:

```
sage: x = var('x')
sage: x.gamma()
gamma(x)
sage: SR(2).gamma()
1
sage: SR(10).gamma()
362880
sage: SR(10.0r).gamma()   # For ARM: rel tol 2e-15
362880.0
sage: SR(CDF(1,1)).gamma()
0.49801566811835607 - 0.15494982830181067*I
```

```
>>> from sage.all import *
>>> x = var('x')
>>> x.gamma()
gamma(x)
>>> SR(Integer(2)).gamma()
1
>>> SR(Integer(10)).gamma()
362880
>>> SR(10.0).gamma()   # For ARM: rel tol 2e-15
362880.0
>>> SR(CDF(Integer(1),Integer(1))).gamma()
0.49801566811835607 - 0.15494982830181067*I
```

```
sage: gp('gamma(1+I)')
0.49801566811835604271369111746219809195 - 0.
↪15494982830181068512495513048388660520*I
```

```
>>> from sage.all import *
>>> gp('gamma(1+I)')
0.49801566811835604271369111746219809195 - 0.
↪15494982830181068512495513048388660520*I
```

We plot the familiar plot of this log-convex function:

```
sage: plot(gamma(x), -6, 4).show(ymin=-3, ymax=3)                              #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> plot(gamma(x), -Integer(6), Integer(4)).show(ymin=-Integer(3),␣
↪ymax=Integer(3))                        # needs sage.plot
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(1/2).gamma()
sqrt(pi)
sage: SR(1/2).gamma(hold=True)
gamma(1/2)
```

```
>>> from sage.all import *
>>> SR(Integer(1)/Integer(2)).gamma()
sqrt(pi)
>>> SR(Integer(1)/Integer(2)).gamma(hold=True)
gamma(1/2)
```

This also works using functional notation:

```
sage: gamma(1/2, hold=True)
gamma(1/2)
sage: gamma(1/2)
sqrt(pi)
```

```
>>> from sage.all import *
>>> gamma(Integer(1)/Integer(2), hold=True)
gamma(1/2)
>>> gamma(Integer(1)/Integer(2))
sqrt(pi)
```

To then evaluate again, we use `unhold()`:

```
sage: a = SR(1/2).gamma(hold=True); a.unhold()
sqrt(pi)
```

```
>>> from sage.all import *
>>> a = SR(Integer(1)/Integer(2)).gamma(hold=True); a.unhold()
sqrt(pi)
```

gamma_normalize()

Return the expression with any gamma functions that have a common base converted to that base.

Additionally the expression is normalized so any fractions can be simplified through cancellation.

EXAMPLES:

```
sage: m,n = var('m n', domain='integer')
sage: (gamma(n+2)/gamma(n)).gamma_normalize()
(n + 1)*n
sage: (gamma(n+2)*gamma(n)).gamma_normalize()
(n + 1)*n*gamma(n)^2
sage: (gamma(n+2)*gamma(m-1)/gamma(n)/gamma(m+1)).gamma_normalize()
(n + 1)*n/((m - 1)*m)
```

```
>>> from sage.all import *
>>> m,n = var('m n', domain='integer')
>>> (gamma(n+Integer(2))/gamma(n)).gamma_normalize()
(n + 1)*n
>>> (gamma(n+Integer(2))*gamma(n)).gamma_normalize()
(n + 1)*n*gamma(n)^2
>>> (gamma(n+Integer(2))*gamma(m-Integer(1))/gamma(n)/gamma(m+Integer(1))).
↪gamma_normalize()
(n + 1)*n/((m - 1)*m)
```

Check that Issue #22826 is fixed:

```
sage: _ = var('n')
sage: (n-1).gcd(n+1)
1
sage: ex = (n-1)^2*gamma(2*n+5)/gamma(n+3) + gamma(2*n+3)/gamma(n+1)
sage: ex.gamma_normalize()
(4*n^3 - 2*n^2 - 7*n + 7)*gamma(2*n + 3)/((n + 1)*gamma(n + 1))
```

```
>>> from sage.all import *
>>> _ = var('n')
>>> (n-Integer(1)).gcd(n+Integer(1))
1
>>> ex = (n-Integer(1))**Integer(2)*gamma(Integer(2)*n+Integer(5))/
↪gamma(n+Integer(3)) + gamma(Integer(2)*n+Integer(3))/gamma(n+Integer(1))
>>> ex.gamma_normalize()
(4*n^3 - 2*n^2 - 7*n + 7)*gamma(2*n + 3)/((n + 1)*gamma(n + 1))
```

gcd(*b*)

Return the symbolic gcd of `self` and `b`.

Note that the polynomial GCD is unique up to the multiplication by an invertible constant. The following examples make sure all results are caught.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: SR(10).gcd(SR(15))
5
sage: (x^3 - 1).gcd(x-1) / (x-1) in QQ
True
sage: (x^3 - 1).gcd(x^2+x+1) / (x^2+x+1) in QQ
True
sage: (x^3 - x^2*pi + x^2 - pi^2).gcd(x-pi) / (x-pi) in QQ
True
sage: gcd(sin(x)^2 + sin(x), sin(x)^2 - 1) / (sin(x) + 1) in QQ
True
sage: gcd(x^3 - y^3, x-y) / (x-y) in QQ
True
sage: gcd(x^100-y^100, x^10-y^10) / (x^10-y^10) in QQ
True
sage: r = gcd(expand( (x^2+17*x+3/7*y)*(x^5 - 17*y + 2/3) ), expand((x^
↪13+17*x+3/7*y)*(x^5 - 17*y + 2/3)) )
sage: r / (x^5 - 17*y + 2/3) in QQ
True
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> SR(Integer(10)).gcd(SR(Integer(15)))
5
>>> (x**Integer(3) - Integer(1)).gcd(x-Integer(1)) / (x-Integer(1)) in QQ
True
>>> (x**Integer(3) - Integer(1)).gcd(x**Integer(2)+x+Integer(1)) /␣
```

(continues on next page)

(continued from previous page)

```
↪(x**Integer(2)+x+Integer(1)) in QQ
True
>>> (x**Integer(3) - x**Integer(2)*pi + x**Integer(2) - pi**Integer(2)).gcd(x-
↪pi) / (x-pi) in QQ
True
>>> gcd(sin(x)**Integer(2) + sin(x), sin(x)**Integer(2) - Integer(1)) /
↪(sin(x) + Integer(1)) in QQ
True
>>> gcd(x**Integer(3) - y**Integer(3), x-y) / (x-y) in QQ
True
>>> gcd(x**Integer(100)-y**Integer(100), x**Integer(10)-y**Integer(10)) /
↪(x**Integer(10)-y**Integer(10)) in QQ
True
>>> r = gcd(expand( (x**Integer(2)+Integer(17)*x+Integer(3)/
↪Integer(7)*y)*(x**Integer(5) - Integer(17)*y + Integer(2)/Integer(3)) ),
↪expand((x**Integer(13)+Integer(17)*x+Integer(3)/
↪Integer(7)*y)*(x**Integer(5) - Integer(17)*y + Integer(2)/Integer(3))) )
>>> r / (x**Integer(5) - Integer(17)*y + Integer(2)/Integer(3)) in QQ
True
```

Embedded Sage objects of all kinds get basic support. Note that full algebraic GCD is not implemented yet:

```
sage: gcd(I - I*x, x^2 - 1)
x - 1
sage: gcd(I + I*x, x^2 - 1)
x + 1
sage: alg = SR(QQbar(sqrt(2) + I*sqrt(3)))
sage: gcd(alg + alg*x, x^2 - 1)  # known bug (Issue #28489)
x + 1
sage: gcd(alg - alg*x, x^2 - 1)  # known bug (Issue #28489)
x - 1
sage: sqrt2 = SR(QQbar(sqrt(2)))
sage: gcd(sqrt2 + x, x^2 - 2)     # known bug
1
```

```
>>> from sage.all import *
>>> gcd(I - I*x, x**Integer(2) - Integer(1))
x - 1
>>> gcd(I + I*x, x**Integer(2) - Integer(1))
x + 1
>>> alg = SR(QQbar(sqrt(Integer(2)) + I*sqrt(Integer(3))))
>>> gcd(alg + alg*x, x**Integer(2) - Integer(1))  # known bug (Issue #28489)
x + 1
>>> gcd(alg - alg*x, x**Integer(2) - Integer(1))  # known bug (Issue #28489)
x - 1
>>> sqrt2 = SR(QQbar(sqrt(Integer(2))))
>>> gcd(sqrt2 + x, x**Integer(2) - Integer(2))     # known bug
1
```

gosper_sum(**args*)

Return the summation of this hypergeometric expression using Gosper's algorithm.

INPUT:

- a symbolic expression that may contain rational functions, powers, factorials, gamma function terms, binomial coefficients, and Pochhammer symbols that are rational-linear in their arguments
- the main variable and, optionally, summation limits

EXAMPLES:

```
sage: a,b,k,m,n = var('a b k m n')
sage: SR(1).gosper_sum(n)
n
sage: SR(1).gosper_sum(n,5,8)
4
sage: n.gosper_sum(n)
1/2*(n - 1)*n
sage: n.gosper_sum(n,0,5)
15
sage: n.gosper_sum(n,0,m)
1/2*(m + 1)*m
sage: n.gosper_sum(n,a,b)
-1/2*(a + b)*(a - b - 1)
```

```
>>> from sage.all import *
>>> a,b,k,m,n = var('a b k m n')
>>> SR(Integer(1)).gosper_sum(n)
n
>>> SR(Integer(1)).gosper_sum(n,Integer(5),Integer(8))
4
>>> n.gosper_sum(n)
1/2*(n - 1)*n
>>> n.gosper_sum(n,Integer(0),Integer(5))
15
>>> n.gosper_sum(n,Integer(0),m)
1/2*(m + 1)*m
>>> n.gosper_sum(n,a,b)
-1/2*(a + b)*(a - b - 1)
```

```
sage: (factorial(m + n)/factorial(n)).gosper_sum(n)
n*factorial(m + n)/((m + 1)*factorial(n))
sage: (binomial(m + n, n)).gosper_sum(n)
n*binomial(m + n, n)/(m + 1)
sage: (binomial(m + n, n)).gosper_sum(n, 0, a)
(a + m + 1)*binomial(a + m, a)/(m + 1)
sage: (binomial(m + n, n)).gosper_sum(n, 0, 5)
1/120*(m + 6)*(m + 5)*(m + 4)*(m + 3)*(m + 2)
sage: (rising_factorial(a,n)/rising_factorial(b,n)).gosper_sum(n)
(b + n - 1)*gamma(a + n)*gamma(b)/((a - b + 1)*gamma(a)*gamma(b + n))
sage: factorial(n).gosper_term(n)
Traceback (most recent call last):
...
ValueError: expression not Gosper-summable
```

```
>>> from sage.all import *
>>> (factorial(m + n)/factorial(n)).gosper_sum(n)
n*factorial(m + n)/((m + 1)*factorial(n))
```

(continues on next page)

(continued from previous page)

```
>>> (binomial(m + n, n)).gosper_sum(n)
n*binomial(m + n, n)/(m + 1)
>>> (binomial(m + n, n)).gosper_sum(n, Integer(0), a)
(a + m + 1)*binomial(a + m, a)/(m + 1)
>>> (binomial(m + n, n)).gosper_sum(n, Integer(0), Integer(5))
1/120*(m + 6)*(m + 5)*(m + 4)*(m + 3)*(m + 2)
>>> (rising_factorial(a,n)/rising_factorial(b,n)).gosper_sum(n)
(b + n - 1)*gamma(a + n)*gamma(b)/((a - b + 1)*gamma(a)*gamma(b + n))
>>> factorial(n).gosper_term(n)
Traceback (most recent call last):
...
ValueError: expression not Gosper-summable
```

gosper_term(*n*)

Return Gosper's hypergeometric term for `self`.

Suppose `f``=``self` is a hypergeometric term such that:

$$s_n = \sum_{k=0}^{n-1} f_k$$

and f_k doesn't depend on n. Return a hypergeometric term g_n such that $g_{n+1} - g_n = f_n$.

EXAMPLES:

```
sage: _ = var('n')
sage: SR(1).gosper_term(n)
n
sage: n.gosper_term(n)
1/2*(n^2 - n)/n
sage: (n*factorial(n)).gosper_term(n)
1/n
sage: factorial(n).gosper_term(n)
Traceback (most recent call last):
...
ValueError: expression not Gosper-summable
```

```
>>> from sage.all import *
>>> _ = var('n')
>>> SR(Integer(1)).gosper_term(n)
n
>>> n.gosper_term(n)
1/2*(n^2 - n)/n
>>> (n*factorial(n)).gosper_term(n)
1/n
>>> factorial(n).gosper_term(n)
Traceback (most recent call last):
...
ValueError: expression not Gosper-summable
```

gradient(*variables=None*)

Compute the gradient of a symbolic function.

This function returns a vector whose components are the derivatives of the original function with respect to the arguments of the original function. Alternatively, you can specify the variables as a list.

EXAMPLES:

```
sage: x,y = var('x y')
sage: f = x^2+y^2
sage: f.gradient()
(2*x, 2*y)
sage: g(x,y) = x^2+y^2
sage: g.gradient()
(x, y) |--> (2*x, 2*y)
sage: n = var('n')
sage: f(x,y) = x^n+y^n
sage: f.gradient()
(x, y) |--> (n*x^(n - 1), n*y^(n - 1))
sage: f.gradient([y,x])
(x, y) |--> (n*y^(n - 1), n*x^(n - 1))
```

```
>>> from sage.all import *
>>> x,y = var('x y')
>>> f = x**Integer(2)+y**Integer(2)
>>> f.gradient()
(2*x, 2*y)
>>> __tmp__=var("x,y"); g = symbolic_expression(x**Integer(2)+y**Integer(2)).
↪function(x,y)
>>> g.gradient()
(x, y) |--> (2*x, 2*y)
>>> n = var('n')
>>> __tmp__=var("x,y"); f = symbolic_expression(x**n+y**n).function(x,y)
>>> f.gradient()
(x, y) |--> (n*x^(n - 1), n*y^(n - 1))
>>> f.gradient([y,x])
(x, y) |--> (n*y^(n - 1), n*x^(n - 1))
```

See also

`gradient()` of scalar fields on Euclidean spaces (and more generally pseudo-Riemannian manifolds), in particular for computing the gradient in curvilinear coordinates.

half_angle()

Replace all occurrences of trigonometric (or hyperbolic) functions by rational fractions of the (hyperbolic) tangent of half the original argument.

This can help highlight the algebraic structure of an expression, which can be useful e.g. for integration.

This method has no direct relation with the `half_angles` argument of the *`trig_expand()`* method.

EXAMPLES:

```
sage: x, t = var("x, t")
sage: cos(x).half_angle().subs(tan(x/2) == t)
-(t^2 - 1)/(t^2 + 1)
```

```
>>> from sage.all import *
>>> x, t = var("x, t")
```

(continues on next page)

(continued from previous page)

```
>>> cos(x).half_angle().subs(tan(x/Integer(2)) == t)
-(t^2 - 1)/(t^2 + 1)
```

Note that this structure highlighting works better after expansion:

```
sage: x, t = var("x, t")
sage: a = (cos(3*x)/(4-cos(x)))
sage: b = a.trig_expand()
sage: a.half_angle().subs(tan(x/2) == t).simplify_full()
(2*(t^2 + 1)*cos(3/2*x)^2 - t^2 - 1)/(5*t^2 + 3)
sage: b.half_angle().subs(tan(x/2) == t).simplify_full()
-(t^6 - 15*t^4 + 15*t^2 - 1)/(5*t^6 + 13*t^4 + 11*t^2 + 3)
```

```
>>> from sage.all import *
>>> x, t = var("x, t")
>>> a = (cos(Integer(3)*x)/(Integer(4)-cos(x)))
>>> b = a.trig_expand()
>>> a.half_angle().subs(tan(x/Integer(2)) == t).simplify_full()
(2*(t^2 + 1)*cos(3/2*x)^2 - t^2 - 1)/(5*t^2 + 3)
>>> b.half_angle().subs(tan(x/Integer(2)) == t).simplify_full()
-(t^6 - 15*t^4 + 15*t^2 - 1)/(5*t^6 + 13*t^4 + 11*t^2 + 3)
```

has(*pattern*)

EXAMPLES:

```
sage: var('x,y,a'); w0 = SR.wild(); w1 = SR.wild()
(x, y, a)
sage: (x*sin(x + y + 2*a)).has(y)
True
```

```
>>> from sage.all import *
>>> var('x,y,a'); w0 = SR.wild(); w1 = SR.wild()
(x, y, a)
>>> (x*sin(x + y + Integer(2)*a)).has(y)
True
```

Here “x+y” is not a subexpression of “x+y+2*a” (which has the subexpressions “x”, “y” and “2*a”):

```
sage: (x*sin(x + y + 2*a)).has(x+y)
False
sage: (x*sin(x + y + 2*a)).has(x + y + w0)
True
```

```
>>> from sage.all import *
>>> (x*sin(x + y + Integer(2)*a)).has(x+y)
False
>>> (x*sin(x + y + Integer(2)*a)).has(x + y + w0)
True
```

The following fails because “2*(x+y)” automatically gets converted to “2*x+2*y” of which “x+y” is not a subexpression:

```
sage: (x*sin(2*(x+y) + 2*a)).has(x+y)
False
```

```
>>> from sage.all import *
>>> (x*sin(Integer(2)*(x+y) + Integer(2)*a)).has(x+y)
False
```

Although x^1==x and x^0==1, neither "x" nor "1" are actually of the form "x^something":

```
sage: (x+1).has(x^w0)
False
```

```
>>> from sage.all import *
>>> (x+Integer(1)).has(x**w0)
False
```

Here is another possible pitfall, where the first expression matches because the term "-x" has the form "(-1)*x" in GiNaC. To check whether a polynomial contains a linear term you should use the coeff() function instead.

```
sage: (4*x^2 - x + 3).has(w0*x)
True
sage: (4*x^2 + x + 3).has(w0*x)
False
sage: (4*x^2 + x + 3).has(x)
True
sage: (4*x^2 - x + 3).coefficient(x,1)
-1
sage: (4*x^2 + x + 3).coefficient(x,1)
1
```

```
>>> from sage.all import *
>>> (Integer(4)*x**Integer(2) - x + Integer(3)).has(w0*x)
True
>>> (Integer(4)*x**Integer(2) + x + Integer(3)).has(w0*x)
False
>>> (Integer(4)*x**Integer(2) + x + Integer(3)).has(x)
True
>>> (Integer(4)*x**Integer(2) - x + Integer(3)).coefficient(x,Integer(1))
-1
>>> (Integer(4)*x**Integer(2) + x + Integer(3)).coefficient(x,Integer(1))
1
```

has_wild()

Return `True` if this expression contains a wildcard.

EXAMPLES:

```
sage: (1 + x^2).has_wild()
False
sage: (SR.wild(0) + x^2).has_wild()
True
sage: SR.wild(0).has_wild()
True
```

```
>>> from sage.all import *
>>> (Integer(1) + x**Integer(2)).has_wild()
False
>>> (SR.wild(Integer(0)) + x**Integer(2)).has_wild()
True
>>> SR.wild(Integer(0)).has_wild()
True
```

hessian()

Compute the hessian of a function. This returns a matrix components are the 2nd partial derivatives of the original function.

EXAMPLES:

```
sage: x,y = var('x y')
sage: f = x^2+y^2
sage: f.hessian()
[2 0]
[0 2]
sage: g(x,y) = x^2+y^2
sage: g.hessian()
[(x, y) |--> 2 (x, y) |--> 0]
[(x, y) |--> 0 (x, y) |--> 2]
```

```
>>> from sage.all import *
>>> x,y = var('x y')
>>> f = x**Integer(2)+y**Integer(2)
>>> f.hessian()
[2 0]
[0 2]
>>> __tmp__=var("x,y"); g = symbolic_expression(x**Integer(2)+y**Integer(2)).
↪function(x,y)
>>> g.hessian()
[(x, y) |--> 2 (x, y) |--> 0]
[(x, y) |--> 0 (x, y) |--> 2]
```

horner(*x*)

Rewrite this expression as a polynomial in Horner form in `x`.

EXAMPLES:

```
sage: add((i+1)*x^i for i in range(5)).horner(x)
(((5*x + 4)*x + 3)*x + 2)*x + 1

sage: x, y, z = SR.var('x,y,z')
sage: (x^5 + y*cos(x) + z^3 + (x + y)^2 + y^x).horner(x)
z^3 + ((x^3 + 1)*x + 2*y)*x + y^2 + y*cos(x) + y^x

sage: expr = sin(5*x).expand_trig(); expr
5*cos(x)^4*sin(x) - 10*cos(x)^2*sin(x)^3 + sin(x)^5
sage: expr.horner(sin(x))
(5*cos(x)^4 - (10*cos(x)^2 - sin(x)^2)*sin(x)^2)*sin(x)
sage: expr.horner(cos(x))
sin(x)^5 + 5*(cos(x)^2*sin(x) - 2*sin(x)^3)*cos(x)^2
```

```
>>> from sage.all import *
>>> add((i+Integer(1))*x**i for i in range(Integer(5))).horner(x)
((((5*x + 4)*x + 3)*x + 2)*x + 1

>>> x, y, z = SR.var('x,y,z')
>>> (x**Integer(5) + y*cos(x) + z**Integer(3) + (x + y)**Integer(2) + y**x).
↪horner(x)
z^3 + ((x^3 + 1)*x + 2*y)*x + y^2 + y*cos(x) + y^x

>>> expr = sin(Integer(5)*x).expand_trig(); expr
5*cos(x)^4*sin(x) - 10*cos(x)^2*sin(x)^3 + sin(x)^5
>>> expr.horner(sin(x))
(5*cos(x)^4 - (10*cos(x)^2 - sin(x)^2)*sin(x)^2)*sin(x)
>>> expr.horner(cos(x))
sin(x)^5 + 5*(cos(x)^2*sin(x) - 2*sin(x)^3)*cos(x)^2
```

hypergeometric_simplify(*algorithm='maxima'*)

Simplify an expression containing hypergeometric or confluent hypergeometric functions.

INPUT:

- `algorithm` – (default: `'maxima'`) the algorithm to use for for simplification. Implemented are `'maxima'`, which uses Maxima's `hgfred` function, and `'sage'`, which uses an algorithm implemented in the hypergeometric module

ALIAS: *hypergeometric_simplify()* and *simplify_hypergeometric()* are the same

EXAMPLES:

```
sage: hypergeometric((5, 4), (4, 1, 2, 3),
....:                x).simplify_hypergeometric()
1/144*x^2*hypergeometric((), (3, 4), x) +...
1/3*x*hypergeometric((), (2, 3), x) + hypergeometric((), (1, 2), x)
sage: (2*hypergeometric((), (), x)).simplify_hypergeometric()
2*e^x
sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1)   # not tested,␣
↪unstable
....:  .simplify_hypergeometric())
laguerre(-laguerre(-e^x, x), x)
sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1)   # not tested,␣
↪unstable
....:  .simplify_hypergeometric(algorithm='sage'))
hypergeometric((hypergeometric((e^x,), (1,), x),), (1,), x)
sage: hypergeometric_M(1, 3, x).simplify_hypergeometric()
-2*((x + 1)*e^(-x) - 1)*e^x/x^2
sage: (2 * hypergeometric_U(1, 3, x)).simplify_hypergeometric()
2*(x + 1)/x^2
```

```
>>> from sage.all import *
>>> hypergeometric((Integer(5), Integer(4)), (Integer(4), Integer(1),␣
↪Integer(2), Integer(3)),
...                x).simplify_hypergeometric()
1/144*x^2*hypergeometric((), (3, 4), x) +...
1/3*x*hypergeometric((), (2, 3), x) + hypergeometric((), (1, 2), x)
```

(continues on next page)

(continued from previous page)

```
>>> (Integer(2)*hypergeometric((), (), x)).simplify_hypergeometric()
2*e^x
>>> (nest(lambda y: hypergeometric([y], [Integer(1)], x), Integer(3),␣
↪Integer(1))  # not tested, unstable
...   .simplify_hypergeometric())
laguerre(-laguerre(-e^x, x), x)
>>> (nest(lambda y: hypergeometric([y], [Integer(1)], x), Integer(3),␣
↪Integer(1))  # not tested, unstable
...   .simplify_hypergeometric(algorithm='sage'))
hypergeometric((hypergeometric((e^x,), (1,), x),), (1,), x)
>>> hypergeometric_M(Integer(1), Integer(3), x).simplify_hypergeometric()
-2*((x + 1)*e^(-x) - 1)*e^x/x^2
>>> (Integer(2) * hypergeometric_U(Integer(1), Integer(3), x)).simplify_
↪hypergeometric()
2*(x + 1)/x^2
```

imag(*hold=False*)

Return the imaginary part of this symbolic expression.

EXAMPLES:

```
sage: sqrt(-2).imag_part()
sqrt(2)
```

```
>>> from sage.all import *
>>> sqrt(-Integer(2)).imag_part()
sqrt(2)
```

We simplify $\ln(\exp(z))$ to z. This should only be for $-\pi < \text{Im}(z) <= \pi$, but Maxima does not have a symbolic imaginary part function, so we cannot use `assume` to assume that first:

```
sage: z = var('z')
sage: f = log(exp(z))
sage: f
log(e^z)
sage: f.simplify()
z
sage: forget()
```

```
>>> from sage.all import *
>>> z = var('z')
>>> f = log(exp(z))
>>> f
log(e^z)
>>> f.simplify()
z
>>> forget()
```

A more symbolic example:

```
sage: var('a, b')
(a, b)
```

(continues on next page)

(continued from previous page)

```
sage: f = log(a + b*I)
sage: f.imag_part()
arctan2(imag_part(a) + real_part(b), -imag_part(b) + real_part(a))
```

```
>>> from sage.all import *
>>> var('a, b')
(a, b)
>>> f = log(a + b*I)
>>> f.imag_part()
arctan2(imag_part(a) + real_part(b), -imag_part(b) + real_part(a))
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(I).imag_part()
1
sage: SR(I).imag_part(hold=True)
imag_part(I)
```

```
>>> from sage.all import *
>>> SR(I).imag_part()
1
>>> SR(I).imag_part(hold=True)
imag_part(I)
```

This also works using functional notation:

```
sage: imag_part(I, hold=True)
imag_part(I)
sage: imag_part(SR(I))
1
```

```
>>> from sage.all import *
>>> imag_part(I, hold=True)
imag_part(I)
>>> imag_part(SR(I))
1
```

To then evaluate again, we use *unhold()*:

```
sage: a = SR(I).imag_part(hold=True); a.unhold()
1
```

```
>>> from sage.all import *
>>> a = SR(I).imag_part(hold=True); a.unhold()
1
```

imag_part(*hold=False*)

Return the imaginary part of this symbolic expression.

EXAMPLES:

```
sage: sqrt(-2).imag_part()
sqrt(2)
```

```
>>> from sage.all import *
>>> sqrt(-Integer(2)).imag_part()
sqrt(2)
```

We simplify $\ln(\exp(z))$ to z. This should only be for $-\pi < \mathrm{Im}(z) <= \pi$, but Maxima does not have a symbolic imaginary part function, so we cannot use `assume` to assume that first:

```
sage: z = var('z')
sage: f = log(exp(z))
sage: f
log(e^z)
sage: f.simplify()
z
sage: forget()
```

```
>>> from sage.all import *
>>> z = var('z')
>>> f = log(exp(z))
>>> f
log(e^z)
>>> f.simplify()
z
>>> forget()
```

A more symbolic example:

```
sage: var('a, b')
(a, b)
sage: f = log(a + b*I)
sage: f.imag_part()
arctan2(imag_part(a) + real_part(b), -imag_part(b) + real_part(a))
```

```
>>> from sage.all import *
>>> var('a, b')
(a, b)
>>> f = log(a + b*I)
>>> f.imag_part()
arctan2(imag_part(a) + real_part(b), -imag_part(b) + real_part(a))
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(I).imag_part()
1
sage: SR(I).imag_part(hold=True)
imag_part(I)
```

```
>>> from sage.all import *
>>> SR(I).imag_part()
1
```

(continues on next page)

(continued from previous page)

```
>>> SR(I).imag_part(hold=True)
imag_part(I)
```

This also works using functional notation:

```
sage: imag_part(I, hold=True)
imag_part(I)
sage: imag_part(SR(I))
1
```

```
>>> from sage.all import *
>>> imag_part(I, hold=True)
imag_part(I)
>>> imag_part(SR(I))
1
```

To then evaluate again, we use `unhold()`:

```
sage: a = SR(I).imag_part(hold=True); a.unhold()
1
```

```
>>> from sage.all import *
>>> a = SR(I).imag_part(hold=True); a.unhold()
1
```

implicit_derivative(Y, X, n=*1*)

Return the n-th derivative of Y with respect to X given implicitly by this expression.

INPUT:

- `Y` – the dependent variable of the implicit expression
- `X` – the independent variable with respect to which the derivative is taken
- `n` – (default: 1) the order of the derivative

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: f = cos(x)*sin(y)
sage: f.implicit_derivative(y, x)
sin(x)*sin(y)/(cos(x)*cos(y))
sage: g = x*y^2
sage: g.implicit_derivative(y, x, 3)
-1/4*(y + 2*y/x)/x^2 + 1/4*(2*y^2/x - y^2/x^2)/(x*y) - 3/4*y/x^3
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> f = cos(x)*sin(y)
>>> f.implicit_derivative(y, x)
sin(x)*sin(y)/(cos(x)*cos(y))
>>> g = x*y**Integer(2)
```

(continues on next page)

(continued from previous page)

```
>>> g.implicit_derivative(y, x, Integer(3))
-1/4*(y + 2*y/x)/x^2 + 1/4*(2*y^2/x - y^2/x^2)/(x*y) - 3/4*y/x^3
```

It is an error to not include an independent variable term in the expression:

```
sage: (cos(x)*sin(x)).implicit_derivative(y, x)
Traceback (most recent call last):
...
ValueError: Expression cos(x)*sin(x) contains no y terms
```

```
>>> from sage.all import *
>>> (cos(x)*sin(x)).implicit_derivative(y, x)
Traceback (most recent call last):
...
ValueError: Expression cos(x)*sin(x) contains no y terms
```

integral(**args*, ***kwds*)

Compute the integral of `self`.

Please see *sage.symbolic.integration.integral.integrate()* for more details.

EXAMPLES:

```
sage: sin(x).integral(x,0,3)
-cos(3) + 1
sage: sin(x).integral(x)
-cos(x)
```

```
>>> from sage.all import *
>>> sin(x).integral(x,Integer(0),Integer(3))
-cos(3) + 1
>>> sin(x).integral(x)
-cos(x)
```

integrate(**args*, ***kwds*)

Compute the integral of `self`.

Please see *sage.symbolic.integration.integral.integrate()* for more details.

EXAMPLES:

```
sage: sin(x).integral(x,0,3)
-cos(3) + 1
sage: sin(x).integral(x)
-cos(x)
```

```
>>> from sage.all import *
>>> sin(x).integral(x,Integer(0),Integer(3))
-cos(3) + 1
>>> sin(x).integral(x)
-cos(x)
```

inverse_laplace(*t*, *s*)

Return inverse Laplace transform of `self`.

See `sage.calculus.calculus.inverse_laplace`

EXAMPLES:

```
sage: var('w, m')
(w, m)
sage: f = (1/(w^2+10)).inverse_laplace(w, m); f
1/10*sqrt(10)*sin(sqrt(10)*m)
```

```
>>> from sage.all import *
>>> var('w, m')
(w, m)
>>> f = (Integer(1)/(w**Integer(2)+Integer(10))).inverse_laplace(w, m); f
1/10*sqrt(10)*sin(sqrt(10)*m)
```

is_algebraic()

Return `True` if this expression is known to be algebraic.

EXAMPLES:

```
sage: sqrt(2).is_algebraic()
True
sage: (5*sqrt(2)).is_algebraic()
True
sage: (sqrt(2) + 2^(1/3) - 1).is_algebraic()
True
sage: (I*golden_ratio + sqrt(2)).is_algebraic()
True
sage: (sqrt(2) + pi).is_algebraic()
False
sage: SR(QQ(2/3)).is_algebraic()
True
sage: SR(1.2).is_algebraic()
False

sage: complex_root_of(x^3 - x^2 - x - 1, 0).is_algebraic()
True
```

```
>>> from sage.all import *
>>> sqrt(Integer(2)).is_algebraic()
True
>>> (Integer(5)*sqrt(Integer(2))).is_algebraic()
True
>>> (sqrt(Integer(2)) + Integer(2)**(Integer(1)/Integer(3)) - Integer(1)).is_
↪algebraic()
True
>>> (I*golden_ratio + sqrt(Integer(2))).is_algebraic()
True
>>> (sqrt(Integer(2)) + pi).is_algebraic()
False
>>> SR(QQ(Integer(2)/Integer(3))).is_algebraic()
True
>>> SR(RealNumber('1.2')).is_algebraic()
False
```

(continues on next page)

(continued from previous page)

```
>>> complex_root_of(x**Integer(3) - x**Integer(2) - x - Integer(1),␣
↪Integer(0)).is_algebraic()
True
```

is_callable()

Return True if self is a callable symbolic expression.

EXAMPLES:

```
sage: var('a x y z')
(a, x, y, z)
sage: f(x, y) = a + 2*x + 3*y + z
sage: f.is_callable()
True
sage: (a+2*x).is_callable()
False
```

```
>>> from sage.all import *
>>> var('a x y z')
(a, x, y, z)
>>> __tmp__=var("x,y"); f = symbolic_expression(a + Integer(2)*x +␣
↪Integer(3)*y + z).function(x,y)
>>> f.is_callable()
True
>>> (a+Integer(2)*x).is_callable()
False
```

is_constant()

Return whether this symbolic expression is a constant.

A symbolic expression is constant if it does not contain any variables.

EXAMPLES:

```
sage: pi.is_constant()
True
sage: SR(1).is_constant()
True
sage: SR(2).is_constant()
True
sage: log(2).is_constant()
True
sage: SR(I).is_constant()
True
sage: x.is_constant()
False
```

```
>>> from sage.all import *
>>> pi.is_constant()
True
>>> SR(Integer(1)).is_constant()
True
```

(continues on next page)

(continued from previous page)

```
>>> SR(Integer(2)).is_constant()
True
>>> log(Integer(2)).is_constant()
True
>>> SR(I).is_constant()
True
>>> x.is_constant()
False
```

is_exact()

Return True if this expression only contains exact numerical coefficients.

EXAMPLES:

```
sage: x, y = var('x, y')
sage: (x+y-1).is_exact()
True
sage: (x+y-1.9).is_exact()
False
sage: x.is_exact()
True
sage: pi.is_exact()
True
sage: (sqrt(x-y) - 2*x + 1).is_exact()
True
sage: ((x-y)^0.5 - 2*x + 1).is_exact()
False
```

```
>>> from sage.all import *
>>> x, y = var('x, y')
>>> (x+y-Integer(1)).is_exact()
True
>>> (x+y-RealNumber('1.9')).is_exact()
False
>>> x.is_exact()
True
>>> pi.is_exact()
True
>>> (sqrt(x-y) - Integer(2)*x + Integer(1)).is_exact()
True
>>> ((x-y)**RealNumber('0.5') - Integer(2)*x + Integer(1)).is_exact()
False
```

is_infinity()

Return True if self is an infinite expression.

EXAMPLES:

```
sage: SR(oo).is_infinity()
True
sage: x.is_infinity()
False
```

```
>>> from sage.all import *
>>> SR(oo).is_infinity()
True
>>> x.is_infinity()
False
```

is_integer()

Return `True` if this expression is known to be an integer.

EXAMPLES:

```
sage: SR(5).is_integer()
True
```

```
>>> from sage.all import *
>>> SR(Integer(5)).is_integer()
True
```

is_negative()

Return `True` if this expression is known to be negative.

EXAMPLES:

```
sage: SR(-5).is_negative()
True
```

```
>>> from sage.all import *
>>> SR(-Integer(5)).is_negative()
True
```

Check if we can correctly deduce negativity of mul objects:

```
sage: t0 = SR.symbol("t0", domain='positive')
sage: t0.is_negative()
False
sage: (-t0).is_negative()
True
sage: (-pi).is_negative()
True
```

```
>>> from sage.all import *
>>> t0 = SR.symbol("t0", domain='positive')
>>> t0.is_negative()
False
>>> (-t0).is_negative()
True
>>> (-pi).is_negative()
True
```

Assumptions on symbols are handled correctly:

```
sage: y = var('y')
sage: assume(y < 0)
```

(continues on next page)

(continued from previous page)

```
sage: y.is_positive()
False
sage: y.is_negative()
True
sage: forget()
```

```
>>> from sage.all import *
>>> y = var('y')
>>> assume(y < Integer(0))
>>> y.is_positive()
False
>>> y.is_negative()
True
>>> forget()
```

is_negative_infinity()

Return `True` if `self` is a negative infinite expression.

EXAMPLES:

```
sage: SR(oo).is_negative_infinity()
False
sage: SR(-oo).is_negative_infinity()
True
sage: x.is_negative_infinity()
False
```

```
>>> from sage.all import *
>>> SR(oo).is_negative_infinity()
False
>>> SR(-oo).is_negative_infinity()
True
>>> x.is_negative_infinity()
False
```

is_numeric()

A Pynac numeric is an object you can do arithmetic with that is not a symbolic variable, function, or constant. Return `True` if this expression only consists of a numeric object.

EXAMPLES:

```
sage: SR(1).is_numeric()
True
sage: x.is_numeric()
False
sage: pi.is_numeric()
False
sage: sin(x).is_numeric()
False
```

```
>>> from sage.all import *
>>> SR(Integer(1)).is_numeric()
```

(continues on next page)

(continued from previous page)

```
True
>>> x.is_numeric()
False
>>> pi.is_numeric()
False
>>> sin(x).is_numeric()
False
```

is_polynomial(*var*)

Return `True` if `self` is a polynomial in the given variable.

EXAMPLES:

```
sage: var('x,y,z')
(x, y, z)
sage: t = x^2 + y; t
x^2 + y
sage: t.is_polynomial(x)
True
sage: t.is_polynomial(y)
True
sage: t.is_polynomial(z)
True

sage: t = sin(x) + y; t
y + sin(x)
sage: t.is_polynomial(x)
False
sage: t.is_polynomial(y)
True
sage: t.is_polynomial(sin(x))
True
```

```
>>> from sage.all import *
>>> var('x,y,z')
(x, y, z)
>>> t = x**Integer(2) + y; t
x^2 + y
>>> t.is_polynomial(x)
True
>>> t.is_polynomial(y)
True
>>> t.is_polynomial(z)
True

>>> t = sin(x) + y; t
y + sin(x)
>>> t.is_polynomial(x)
False
>>> t.is_polynomial(y)
True
>>> t.is_polynomial(sin(x))
```

(continues on next page)

(continued from previous page)

```
True
```

is_positive()

Return `True` if this expression is known to be positive.

EXAMPLES:

```
sage: t0 = SR.symbol("t0", domain='positive')
sage: t0.is_positive()
True
sage: t0.is_negative()
False
sage: t0.is_real()
True
sage: t1 = SR.symbol("t1", domain='positive')
sage: (t0*t1).is_positive()
True
sage: (t0 + t1).is_positive()
True
sage: (t0*x).is_positive()
False
```

```
>>> from sage.all import *
>>> t0 = SR.symbol("t0", domain='positive')
>>> t0.is_positive()
True
>>> t0.is_negative()
False
>>> t0.is_real()
True
>>> t1 = SR.symbol("t1", domain='positive')
>>> (t0*t1).is_positive()
True
>>> (t0 + t1).is_positive()
True
>>> (t0*x).is_positive()
False
```

```
sage: forget()
sage: assume(x>0)
sage: x.is_positive()
True
sage: cosh(x).is_positive()
True
sage: f = function('f')(x)
sage: assume(f>0)
sage: f.is_positive()
True
sage: forget()
```

```
>>> from sage.all import *
>>> forget()
```

(continues on next page)

(continued from previous page)

```
>>> assume(x>Integer(0))
>>> x.is_positive()
True
>>> cosh(x).is_positive()
True
>>> f = function('f')(x)
>>> assume(f>Integer(0))
>>> f.is_positive()
True
>>> forget()
```

is_positive_infinity()

Return True if self is a positive infinite expression.

EXAMPLES:

```
sage: SR(oo).is_positive_infinity()
True
sage: SR(-oo).is_positive_infinity()
False
sage: x.is_infinity()
False
```

```
>>> from sage.all import *
>>> SR(oo).is_positive_infinity()
True
>>> SR(-oo).is_positive_infinity()
False
>>> x.is_infinity()
False
```

is_rational_expression()

Return True if this expression if a rational expression, i.e., a quotient of polynomials.

EXAMPLES:

```
sage: var('x y z')
(x, y, z)
sage: ((x + y + z)/(1 + x^2)).is_rational_expression()
True
sage: ((1 + x + y)^10).is_rational_expression()
True
sage: ((1/x + z)^5 - 1).is_rational_expression()
True
sage: (1/(x + y)).is_rational_expression()
True
sage: (exp(x) + 1).is_rational_expression()
False
sage: (sin(x*y) + z^3).is_rational_expression()
False
sage: (exp(x) + exp(-x)).is_rational_expression()
False
```

```
>>> from sage.all import *
>>> var('x y z')
(x, y, z)
>>> ((x + y + z)/(Integer(1) + x**Integer(2))).is_rational_expression()
True
>>> ((Integer(1) + x + y)**Integer(10)).is_rational_expression()
True
>>> ((Integer(1)/x + z)**Integer(5) - Integer(1)).is_rational_expression()
True
>>> (Integer(1)/(x + y)).is_rational_expression()
True
>>> (exp(x) + Integer(1)).is_rational_expression()
False
>>> (sin(x*y) + z**Integer(3)).is_rational_expression()
False
>>> (exp(x) + exp(-x)).is_rational_expression()
False
```

is_real()

Return `True` if this expression is known to be a real number.

EXAMPLES:

```
sage: t0 = SR.symbol("t0", domain='real')
sage: t0.is_real()
True
sage: t0.is_positive()
False
sage: t1 = SR.symbol("t1", domain='positive')
sage: (t0+t1).is_real()
True
sage: (t0+x).is_real()
False
sage: (t0*t1).is_real()
True
sage: t2 = SR.symbol("t2", domain='positive')
sage: (t1**t2).is_real()
True
sage: (t0*x).is_real()
False
sage: (t0^t1).is_real()
False
sage: (t1^t2).is_real()
True
sage: gamma(pi).is_real()
True
sage: cosh(-3).is_real()
True
sage: cos(exp(-3) + log(2)).is_real()
True
sage: gamma(t1).is_real()
True
sage: (x^pi).is_real()
```

(continues on next page)

(continued from previous page)

```
False
sage: (cos(exp(t0) + log(t1))^8).is_real()
True
sage: cos(I + 1).is_real()
False
sage: sin(2 - I).is_real()
False
sage: (2^t0).is_real()
True
```

```
>>> from sage.all import *
>>> t0 = SR.symbol("t0", domain='real')
>>> t0.is_real()
True
>>> t0.is_positive()
False
>>> t1 = SR.symbol("t1", domain='positive')
>>> (t0+t1).is_real()
True
>>> (t0+x).is_real()
False
>>> (t0*t1).is_real()
True
>>> t2 = SR.symbol("t2", domain='positive')
>>> (t1**t2).is_real()
True
>>> (t0*x).is_real()
False
>>> (t0**t1).is_real()
False
>>> (t1**t2).is_real()
True
>>> gamma(pi).is_real()
True
>>> cosh(-Integer(3)).is_real()
True
>>> cos(exp(-Integer(3)) + log(Integer(2))).is_real()
True
>>> gamma(t1).is_real()
True
>>> (x**pi).is_real()
False
>>> (cos(exp(t0) + log(t1))**Integer(8)).is_real()
True
>>> cos(I + Integer(1)).is_real()
False
>>> sin(Integer(2) - I).is_real()
False
>>> (Integer(2)**t0).is_real()
True
```

The following is real, but we cannot deduce that.:

```
sage: (x*x.conjugate()).is_real()
False
```

```
>>> from sage.all import *
>>> (x*x.conjugate()).is_real()
False
```

Assumption of real has the same effect as setting the domain:

```
sage: forget()
sage: assume(x, 'real')
sage: x.is_real()
True
sage: cosh(x).is_real()
True
sage: forget()
```

```
>>> from sage.all import *
>>> forget()
>>> assume(x, 'real')
>>> x.is_real()
True
>>> cosh(x).is_real()
True
>>> forget()
```

The real domain is also set with the integer domain:

```
sage: SR.var('x', domain='integer').is_real()
True
```

```
>>> from sage.all import *
>>> SR.var('x', domain='integer').is_real()
True
```

is_relational()

Return `True` if `self` is a relational expression.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 == x^2 - 2*x + 3
sage: eqn.is_relational()
True
sage: sin(x).is_relational()
False
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eqn = (x-Integer(1))**Integer(2) == x**Integer(2) - Integer(2)*x +␣
↪Integer(3)
>>> eqn.is_relational()
```

(continues on next page)

(continued from previous page)

```
True
>>> sin(x).is_relational()
False
```

is_square()

Return `True` if `self` is the square of another symbolic expression.

This is `True` for all constant, non-relational expressions (containing no variables or comparison), and not implemented otherwise.

EXAMPLES:

```
sage: SR(4).is_square()
True
sage: SR(5).is_square()
True
sage: pi.is_square()
True
sage: x.is_square()
Traceback (most recent call last):
...
NotImplementedError: is_square() not implemented for non-constant
or relational elements of Symbolic Ring
sage: r = SR(4) == SR(5)
sage: r.is_square()
Traceback (most recent call last):
...
NotImplementedError: is_square() not implemented for non-constant
or relational elements of Symbolic Ring
```

```
>>> from sage.all import *
>>> SR(Integer(4)).is_square()
True
>>> SR(Integer(5)).is_square()
True
>>> pi.is_square()
True
>>> x.is_square()
Traceback (most recent call last):
...
NotImplementedError: is_square() not implemented for non-constant
or relational elements of Symbolic Ring
>>> r = SR(Integer(4)) == SR(Integer(5))
>>> r.is_square()
Traceback (most recent call last):
...
NotImplementedError: is_square() not implemented for non-constant
or relational elements of Symbolic Ring
```

is_symbol()

Return `True` if this symbolic expression consists of only a symbol, i.e., a symbolic variable.

EXAMPLES:

```
sage: x.is_symbol()
True
sage: var('y')
y
sage: y.is_symbol()
True
sage: (x*y).is_symbol()
False
sage: pi.is_symbol()
False
```

```
>>> from sage.all import *
>>> x.is_symbol()
True
>>> var('y')
y
>>> y.is_symbol()
True
>>> (x*y).is_symbol()
False
>>> pi.is_symbol()
False
```

```
sage: ((x*y)/y).is_symbol()
True
sage: (x^y).is_symbol()
False
```

```
>>> from sage.all import *
>>> ((x*y)/y).is_symbol()
True
>>> (x**y).is_symbol()
False
```

is_terminating_series()

Return `True` if `self` is a series without order term.

A series is terminating if it can be represented exactly, without requiring an order term. You can explicitly request terminating series by setting the order to positive infinity.

OUTPUT: boolean; whether `self` was constructed by `series()` and has no order term

EXAMPLES:

```
sage: (x^5+x^2+1).series(x, +oo)
1 + 1*x^2 + 1*x^5
sage: (x^5+x^2+1).series(x,+oo).is_terminating_series()
True
sage: SR(5).is_terminating_series()
False
sage: var('x')
x
sage: x.is_terminating_series()
```

(continues on next page)

(continued from previous page)

```
False
sage: exp(x).series(x,10).is_terminating_series()
False
```

```
>>> from sage.all import *
>>> (x**Integer(5)+x**Integer(2)+Integer(1)).series(x, +oo)
1 + 1*x^2 + 1*x^5
>>> (x**Integer(5)+x**Integer(2)+Integer(1)).series(x,+oo).is_terminating_
↪series()
True
>>> SR(Integer(5)).is_terminating_series()
False
>>> var('x')
x
>>> x.is_terminating_series()
False
>>> exp(x).series(x,Integer(10)).is_terminating_series()
False
```

is_trivial_zero()

Check if this expression is trivially equal to zero without any simplification.

This method is intended to be used in library code where trying to obtain a mathematically correct result by applying potentially expensive rewrite rules is not desirable.

EXAMPLES:

```
sage: SR(0).is_trivial_zero()
True
sage: SR(0.0).is_trivial_zero()
True
sage: SR(float(0.0)).is_trivial_zero()
True

sage: (SR(1)/2^1000).is_trivial_zero()
False
sage: SR(1./2^10000).is_trivial_zero()
False
```

```
>>> from sage.all import *
>>> SR(Integer(0)).is_trivial_zero()
True
>>> SR(RealNumber('0.0')).is_trivial_zero()
True
>>> SR(float(RealNumber('0.0'))).is_trivial_zero()
True

>>> (SR(Integer(1))/Integer(2)**Integer(1000)).is_trivial_zero()
False
>>> SR(RealNumber('1.')/Integer(2)**Integer(10000)).is_trivial_zero()
False
```

The `is_zero()` method is more capable:

```
sage: t = pi + (pi - 1)*pi - pi^2
sage: t.is_trivial_zero()
False
sage: t.is_zero()
True
sage: t = pi + x*pi + (pi - 1 - x)*pi - pi^2
sage: t.is_zero()
True
sage: u = sin(x)^2 + cos(x)^2 - 1
sage: u.is_trivial_zero()
False
sage: u.is_zero()
True
```

```
>>> from sage.all import *
>>> t = pi + (pi - Integer(1))*pi - pi**Integer(2)
>>> t.is_trivial_zero()
False
>>> t.is_zero()
True
>>> t = pi + x*pi + (pi - Integer(1) - x)*pi - pi**Integer(2)
>>> t.is_zero()
True
>>> u = sin(x)**Integer(2) + cos(x)**Integer(2) - Integer(1)
>>> u.is_trivial_zero()
False
>>> u.is_zero()
True
```

is_trivially_equal(*other*)

Check if this expression is trivially equal to the argument expression, without any simplification.

Note that the expressions may still be subject to immediate evaluation.

This method is intended to be used in library code where trying to obtain a mathematically correct result by applying potentially expensive rewrite rules is not desirable.

EXAMPLES:

```
sage: (x^2).is_trivially_equal(x^2)
True
sage: ((x+1)^2 - 2*x - 1).is_trivially_equal(x^2)
False
sage: (x*(x+1)).is_trivially_equal((x+1)*x)
True
sage: (x^2 + x).is_trivially_equal((x+1)*x)
False
sage: ((x+1)*(x+1)).is_trivially_equal((x+1)^2)
True
sage: (x^2 + 2*x + 1).is_trivially_equal((x+1)^2)
False
sage: (x^-1).is_trivially_equal(1/x)
True
sage: (x/x^2).is_trivially_equal(1/x)
```

(continues on next page)

(continued from previous page)

```
True
sage: ((x^2+x) / (x+1)).is_trivially_equal(1/x)
False
```

```
>>> from sage.all import *
>>> (x**Integer(2)).is_trivially_equal(x**Integer(2))
True
>>> ((x+Integer(1))**Integer(2) - Integer(2)*x - Integer(1)).is_trivially_
↪equal(x**Integer(2))
False
>>> (x*(x+Integer(1))).is_trivially_equal((x+Integer(1))*x)
True
>>> (x**Integer(2) + x).is_trivially_equal((x+Integer(1))*x)
False
>>> ((x+Integer(1))*(x+Integer(1))).is_trivially_
↪equal((x+Integer(1))**Integer(2))
True
>>> (x**Integer(2) + Integer(2)*x + Integer(1)).is_trivially_
↪equal((x+Integer(1))**Integer(2))
False
>>> (x**-Integer(1)).is_trivially_equal(Integer(1)/x)
True
>>> (x/x**Integer(2)).is_trivially_equal(Integer(1)/x)
True
>>> ((x**Integer(2)+x) / (x+Integer(1))).is_trivially_equal(Integer(1)/x)
False
```

is_unit()

Return `True` if this expression is a unit of the symbolic ring.

Note that a proof may be attempted to get the result. To avoid this use `(ex-1).is_trivial_zero()`.

EXAMPLES:

```
sage: SR(1).is_unit()
True
sage: SR(-1).is_unit()
True
sage: SR(0).is_unit()
False
```

```
>>> from sage.all import *
>>> SR(Integer(1)).is_unit()
True
>>> SR(-Integer(1)).is_unit()
True
>>> SR(Integer(0)).is_unit()
False
```

iterator()

Return an iterator over the operands of this expression.

EXAMPLES:

```
sage: x,y,z = var('x,y,z')
sage: list((x+y+z).iterator())
[x, y, z]
sage: list((x*y*z).iterator())
[x, y, z]
sage: list((x^y*z*(x+y)).iterator())
[x + y, x^y, z]
```

```
>>> from sage.all import *
>>> x,y,z = var('x,y,z')
>>> list((x+y+z).iterator())
[x, y, z]
>>> list((x*y*z).iterator())
[x, y, z]
>>> list((x**y*z*(x+y)).iterator())
[x + y, x^y, z]
```

Note that symbols, constants and numeric objects do not have operands, so the iterator function raises an error in these cases:

```
sage: x.iterator()
Traceback (most recent call last):
...
ValueError: expressions containing only a numeric coefficient,
constant or symbol have no operands
sage: pi.iterator()
Traceback (most recent call last):
...
ValueError: expressions containing only a numeric coefficient,
constant or symbol have no operands
sage: SR(5).iterator()
Traceback (most recent call last):
...
ValueError: expressions containing only a numeric coefficient,
constant or symbol have no operands
```

```
>>> from sage.all import *
>>> x.iterator()
Traceback (most recent call last):
...
ValueError: expressions containing only a numeric coefficient,
constant or symbol have no operands
>>> pi.iterator()
Traceback (most recent call last):
...
ValueError: expressions containing only a numeric coefficient,
constant or symbol have no operands
>>> SR(Integer(5)).iterator()
Traceback (most recent call last):
...
ValueError: expressions containing only a numeric coefficient,
constant or symbol have no operands
```

laplace(*t*, *s*)

Return Laplace transform of self.

See *sage.calculus.calculus.laplace*

EXAMPLES:

```
sage: var('x,s,z')
(x, s, z)
sage: (z + exp(x)).laplace(x, s)
z/s + 1/(s - 1)
```

```
>>> from sage.all import *
>>> var('x,s,z')
(x, s, z)
>>> (z + exp(x)).laplace(x, s)
z/s + 1/(s - 1)
```

laurent_polynomial(*base_ring=None*, *ring=None*)

Return this symbolic expression as a Laurent polynomial over the given base ring, if possible.

INPUT:

- base_ring – (optional) the base ring for the polynomial
- ring – (optional) the parent for the polynomial

You can specify either the base ring (base_ring) you want the output Laurent polynomial to be over, or you can specify the full laurent polynomial ring (ring) you want the output laurent polynomial to be an element of.

EXAMPLES:

```
sage: f = x^2 -2/3/x + 1
sage: f.laurent_polynomial(QQ)
-2/3*x^-1 + 1 + x^2
sage: f.laurent_polynomial(GF(19))
12*x^-1 + 1 + x^2
```

```
>>> from sage.all import *
>>> f = x**Integer(2) -Integer(2)/Integer(3)/x + Integer(1)
>>> f.laurent_polynomial(QQ)
-2/3*x^-1 + 1 + x^2
>>> f.laurent_polynomial(GF(Integer(19)))
12*x^-1 + 1 + x^2
```

lcm(*b*)

Return the lcm of self and b.

The lcm is computed from the gcd of self and b implicitly from the relation self * b = gcd(self, b) * lcm(self, b).

> **Note**
>
> In agreement with the convention in use for integers, if self * b == 0, then gcd(self, b) == max(self, b) and lcm(self, b) == 0.

Note

Since the polynomial lcm is computed from the gcd, and the polynomial gcd is unique up to a constant factor (which can be negative), the polynomial lcm is unique up to a factor of -1.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: SR(10).lcm(SR(15))
30
sage: (x^3 - 1).lcm(x-1)
x^3 - 1
sage: (x^3 - 1).lcm(x^2+x+1)
x^3 - 1
sage: (x^3 - sage.symbolic.constants.pi).lcm(x-sage.symbolic.constants.pi)
(pi - x^3)*(pi - x)
sage: lcm(x^3 - y^3, x-y) / (x^3 - y^3) in [1,-1]
True
sage: lcm(x^100-y^100, x^10-y^10) / (x^100 - y^100) in [1,-1]
True
sage: a = expand( (x^2+17*x+3/7*y)*(x^5 - 17*y + 2/3) )
sage: b = expand((x^13+17*x+3/7*y)*(x^5 - 17*y + 2/3) )
sage: gcd(a,b) * lcm(a,b) / (a * b) in [1,-1]
True
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> SR(Integer(10)).lcm(SR(Integer(15)))
30
>>> (x**Integer(3) - Integer(1)).lcm(x-Integer(1))
x^3 - 1
>>> (x**Integer(3) - Integer(1)).lcm(x**Integer(2)+x+Integer(1))
x^3 - 1
>>> (x**Integer(3) - sage.symbolic.constants.pi).lcm(x-sage.symbolic.
↪constants.pi)
(pi - x^3)*(pi - x)
>>> lcm(x**Integer(3) - y**Integer(3), x-y) / (x**Integer(3) - y**Integer(3))␣
↪in [Integer(1),-Integer(1)]
True
>>> lcm(x**Integer(100)-y**Integer(100), x**Integer(10)-y**Integer(10)) /␣
↪(x**Integer(100) - y**Integer(100)) in [Integer(1),-Integer(1)]
True
>>> a = expand( (x**Integer(2)+Integer(17)*x+Integer(3)/
↪Integer(7)*y)*(x**Integer(5) - Integer(17)*y + Integer(2)/Integer(3)) )
>>> b = expand((x**Integer(13)+Integer(17)*x+Integer(3)/
↪Integer(7)*y)*(x**Integer(5) - Integer(17)*y + Integer(2)/Integer(3)) )
>>> gcd(a,b) * lcm(a,b) / (a * b) in [Integer(1),-Integer(1)]
True
```

The result is not automatically simplified:

```
sage: ex = lcm(sin(x)^2 - 1, sin(x)^2 + sin(x)); ex
(sin(x)^2 + sin(x))*(sin(x)^2 - 1)/(sin(x) + 1)
sage: ex.simplify_full()
sin(x)^3 - sin(x)
```

```
>>> from sage.all import *
>>> ex = lcm(sin(x)**Integer(2) - Integer(1), sin(x)**Integer(2) + sin(x)); ex
(sin(x)^2 + sin(x))*(sin(x)^2 - 1)/(sin(x) + 1)
>>> ex.simplify_full()
sin(x)^3 - sin(x)
```

leading_coeff(*s*)

Return the leading coefficient of `s` in `self`.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.leading_coefficient(x)
sin(x*y)
sage: f.leading_coefficient(y)
x
sage: f.leading_coefficient(sin(x*y))
x^3 + 2/x
```

```
>>> from sage.all import *
>>> var('x,y,a')
(x, y, a)
>>> f = Integer(100) + a*x + x**Integer(3)*sin(x*y) + x*y + x/y +
→Integer(2)*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
>>> f.leading_coefficient(x)
sin(x*y)
>>> f.leading_coefficient(y)
x
>>> f.leading_coefficient(sin(x*y))
x^3 + 2/x
```

leading_coefficient(*s*)

Return the leading coefficient of `s` in `self`.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.leading_coefficient(x)
sin(x*y)
sage: f.leading_coefficient(y)
x
```

(continues on next page)

(continued from previous page)

```
sage: f.leading_coefficient(sin(x*y))
x^3 + 2/x
```

```
>>> from sage.all import *
>>> var('x,y,a')
(x, y, a)
>>> f = Integer(100) + a*x + x**Integer(3)*sin(x*y) + x*y + x/y +␣
↪Integer(2)*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
>>> f.leading_coefficient(x)
sin(x*y)
>>> f.leading_coefficient(y)
x
>>> f.leading_coefficient(sin(x*y))
x^3 + 2/x
```

left()

If `self` is a relational expression, return the left hand side of the relation. Otherwise, raise a `ValueError`.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 == x^2 - 2*x + 3
sage: eqn.left_hand_side()
(x - 1)^2
sage: eqn.lhs()
(x - 1)^2
sage: eqn.left()
(x - 1)^2
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eqn = (x-Integer(1))**Integer(2) == x**Integer(2) - Integer(2)*x +␣
↪Integer(3)
>>> eqn.left_hand_side()
(x - 1)^2
>>> eqn.lhs()
(x - 1)^2
>>> eqn.left()
(x - 1)^2
```

left_hand_side()

If `self` is a relational expression, return the left hand side of the relation. Otherwise, raise a `ValueError`.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 == x^2 - 2*x + 3
sage: eqn.left_hand_side()
(x - 1)^2
sage: eqn.lhs()
(x - 1)^2
```

(continues on next page)

(continued from previous page)

```
sage: eqn.left()
(x - 1)^2
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eqn = (x-Integer(1))**Integer(2) == x**Integer(2) - Integer(2)*x +␣
↪Integer(3)
>>> eqn.left_hand_side()
(x - 1)^2
>>> eqn.lhs()
(x - 1)^2
>>> eqn.left()
(x - 1)^2
```

lhs()

If self is a relational expression, return the left hand side of the relation. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 == x^2 - 2*x + 3
sage: eqn.left_hand_side()
(x - 1)^2
sage: eqn.lhs()
(x - 1)^2
sage: eqn.left()
(x - 1)^2
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eqn = (x-Integer(1))**Integer(2) == x**Integer(2) - Integer(2)*x +␣
↪Integer(3)
>>> eqn.left_hand_side()
(x - 1)^2
>>> eqn.lhs()
(x - 1)^2
>>> eqn.left()
(x - 1)^2
```

limit(**args*, ***kwds*)

Return a symbolic limit.

See *sage.calculus.calculus.limit*

EXAMPLES:

```
sage: (sin(x)/x).limit(x=0)
1
```

```
>>> from sage.all import *
>>> (sin(x)/x).limit(x=Integer(0))
1
```

list(*x=None*)

Return the coefficients of this symbolic expression as a polynomial in x.

INPUT:

- x – (optional) variable

OUTPUT:

A list of expressions where the n-th element is the coefficient of x^n when self is seen as polynomial in x.

EXAMPLES:

```
sage: var('x, y, a')
(x, y, a)
sage: (x^5).list()
[0, 0, 0, 0, 0, 1]
sage: p = x - x^3 + 5/7*x^5
sage: p.list()
[0, 1, 0, -1, 0, 5/7]
sage: p = expand((x-a*sqrt(2))^2 + x + 1); p
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
sage: p.list(a)
[x^2 + x + 1, -2*sqrt(2)*x, 2]
sage: s = (1/(1-x)).series(x,6); s
1 + 1*x + 1*x^2 + 1*x^3 + 1*x^4 + 1*x^5 + Order(x^6)
sage: s.list()
[1, 1, 1, 1, 1, 1]
```

```
>>> from sage.all import *
>>> var('x, y, a')
(x, y, a)
>>> (x**Integer(5)).list()
[0, 0, 0, 0, 0, 1]
>>> p = x - x**Integer(3) + Integer(5)/Integer(7)*x**Integer(5)
>>> p.list()
[0, 1, 0, -1, 0, 5/7]
>>> p = expand((x-a*sqrt(Integer(2)))**Integer(2) + x + Integer(1)); p
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
>>> p.list(a)
[x^2 + x + 1, -2*sqrt(2)*x, 2]
>>> s = (Integer(1)/(Integer(1)-x)).series(x,Integer(6)); s
1 + 1*x + 1*x^2 + 1*x^3 + 1*x^4 + 1*x^5 + Order(x^6)
>>> s.list()
[1, 1, 1, 1, 1, 1]
```

log(*b=None*, *hold=False*)

Return the logarithm of self.

EXAMPLES:

```
sage: x, y = var('x, y')
sage: x.log()
log(x)
sage: (x^y + y^x).log()
log(x^y + y^x)
```

(continues on next page)

(continued from previous page)

```
sage: SR(0).log()
-Infinity
sage: SR(-1).log()
I*pi
sage: SR(1).log()
0
sage: SR(1/2).log()
log(1/2)
sage: SR(0.5).log()
-0.693147180559945
sage: SR(0.5).log().exp()
0.500000000000000
sage: math.log(0.5)
-0.6931471805599453
sage: plot(lambda x: SR(x).log(), 0.1,10)                                   #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> x, y = var('x, y')
>>> x.log()
log(x)
>>> (x**y + y**x).log()
log(x^y + y^x)
>>> SR(Integer(0)).log()
-Infinity
>>> SR(-Integer(1)).log()
I*pi
>>> SR(Integer(1)).log()
0
>>> SR(Integer(1)/Integer(2)).log()
log(1/2)
>>> SR(RealNumber('0.5')).log()
-0.693147180559945
>>> SR(RealNumber('0.5')).log().exp()
0.500000000000000
>>> math.log(RealNumber('0.5'))
-0.6931471805599453
>>> plot(lambda x: SR(x).log(), RealNumber('0.1'),Integer(10))              ␣
↪                    # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

To prevent automatic evaluation use the `hold` argument:

```
sage: I.log()
1/2*I*pi
sage: I.log(hold=True)
log(I)
```

```
>>> from sage.all import *
>>> I.log()
1/2*I*pi
```

(continues on next page)

(continued from previous page)

```
>>> I.log(hold=True)
log(I)
```

To then evaluate again, we use `unhold()`:

```
sage: a = I.log(hold=True); a.unhold()
1/2*I*pi
```

```
>>> from sage.all import *
>>> a = I.log(hold=True); a.unhold()
1/2*I*pi
```

The `hold` parameter also works in functional notation:

```
sage: log(-1, hold=True)
log(-1)
sage: log(-1)
I*pi
```

```
>>> from sage.all import *
>>> log(-Integer(1), hold=True)
log(-1)
>>> log(-Integer(1))
I*pi
```

log_expand(*algorithm='products'*)

Simplify symbolic expression, which can contain logs.

Expands logarithms of powers, logarithms of products and logarithms of quotients. The option `algorithm` specifies which expression types should be expanded.

INPUT:

- `self` – expression to be simplified
- `algorithm` – (default: `'products'`) governs which expression is expanded. Possible values are
  - `'nothing'` (no expansion),
  - `'powers'` (log(a^r) is expanded),
  - `'products'` (like 'powers' and also log(a*b) are expanded),
  - `'all'` (all possible expansion).

  See also examples below.

DETAILS: This uses the Maxima simplifier and sets `logexpand` option for this simplifier. From the Maxima documentation: "Logexpand:true causes log(a^b) to become b*log(a). If it is set to all, log(a*b) will also simplify to log(a)+log(b). If it is set to super, then log(a/b) will also simplify to log(a)-log(b) for rational numbers a/b, a#1. (log(1/b), for integer b, always simplifies.) If it is set to false, all of these simplifications will be turned off. "

ALIAS: `log_expand()` and `expand_log()` are the same

EXAMPLES:

By default powers and products (and quotients) are expanded, but not quotients of integers:

```
sage: (log(3/4*x^pi)).log_expand()
pi*log(x) + log(3/4)
```

```
>>> from sage.all import *
>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand()
pi*log(x) + log(3/4)
```

To expand also log(3/4) use `algorithm='all'`:

```
sage: (log(3/4*x^pi)).log_expand('all')
pi*log(x) + log(3) - 2*log(2)
```

```
>>> from sage.all import *
>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand('all')
pi*log(x) + log(3) - 2*log(2)
```

To expand only the power use `algorithm='powers'`.:

```
sage: (log(x^6)).log_expand('powers')
6*log(x)
```

```
>>> from sage.all import *
>>> (log(x**Integer(6))).log_expand('powers')
6*log(x)
```

The expression `log((3*x)^6)` is not expanded with `algorithm='powers'`, since it is converted into product first:

```
sage: (log((3*x)^6)).log_expand('powers')
log(729*x^6)
```

```
>>> from sage.all import *
>>> (log((Integer(3)*x)**Integer(6))).log_expand('powers')
log(729*x^6)
```

This shows that the option `algorithm` from the previous call has no influence to future calls (we changed some default Maxima flag, and have to ensure that this flag has been restored):

```
sage: (log(3/4*x^pi)).log_expand()
pi*log(x) + log(3/4)

sage: (log(3/4*x^pi)).log_expand('all')
pi*log(x) + log(3) - 2*log(2)

sage: (log(3/4*x^pi)).log_expand()
pi*log(x) + log(3/4)
```

```
>>> from sage.all import *
>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand()
pi*log(x) + log(3/4)

>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand('all')
```

(continues on next page)

(continued from previous page)

```
pi*log(x) + log(3) - 2*log(2)

>>> (log(Integer(3)/Integer(4)*x**pi)).log_expand()
pi*log(x) + log(3/4)
```

AUTHORS:

- Robert Marik (11-2009)

log_gamma(*hold=False*)

Return the log gamma function evaluated at `self`. This is the logarithm of gamma of `self`, where gamma is a complex function such that $gamma(n)$ equals $factorial(n-1)$.

EXAMPLES:

```
sage: x = var('x')
sage: x.log_gamma()
log_gamma(x)
sage: SR(2).log_gamma()
0
sage: SR(5).log_gamma()
log(24)
sage: a = SR(5).log_gamma(); a.n()
3.17805383034795
sage: SR(5-1).factorial().log()
log(24)
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(-1)
sage: plot(lambda x: SR(x).log_gamma(), -7,8, plot_points=1000).show()       #␣
↪needs sage.plot
sage: math.exp(0.5)
1.6487212707001282
sage: plot(lambda x: (SR(x).exp() - SR(-x).exp())/2 - SR(x).sinh(), -1, 1)   #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> x = var('x')
>>> x.log_gamma()
log_gamma(x)
>>> SR(Integer(2)).log_gamma()
0
>>> SR(Integer(5)).log_gamma()
log(24)
>>> a = SR(Integer(5)).log_gamma(); a.n()
3.17805383034795
>>> SR(Integer(5)-Integer(1)).factorial().log()
log(24)
>>> from sage.misc.verbose import set_verbose
>>> set_verbose(-Integer(1))
>>> plot(lambda x: SR(x).log_gamma(), -Integer(7),Integer(8), plot_
↪points=Integer(1000)).show()       # needs sage.plot
>>> math.exp(RealNumber('0.5'))
```

(continues on next page)

(continued from previous page)

```
1.6487212707001282
>>> plot(lambda x: (SR(x).exp() - SR(-x).exp())/Integer(2) - SR(x).sinh(), -
↪Integer(1), Integer(1))   # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(5).log_gamma(hold=True)
log_gamma(5)
```

```
>>> from sage.all import *
>>> SR(Integer(5)).log_gamma(hold=True)
log_gamma(5)
```

To evaluate again, currently we must use numerical evaluation via `n()`:

```
sage: a = SR(5).log_gamma(hold=True); a.n()
3.17805383034795
```

```
>>> from sage.all import *
>>> a = SR(Integer(5)).log_gamma(hold=True); a.n()
3.17805383034795
```

log_simplify(*algorithm=None*)

Simplify a (real) symbolic expression that contains logarithms.

The given expression is scanned recursively, transforming subexpressions of the form $a\log(b) + c\log(d)$ into $\log(b^a d^c)$ before simplifying within the `log()`.

The user can specify conditions that a and c must satisfy before this transformation will be performed using the optional parameter `algorithm`.

Warning

This is only safe to call if every variable in the given expression is assumed to be real. The simplification it performs is in general not valid over the complex numbers. For example:

```
sage: x,y = SR.var('x,y')
sage: f = log(x*y) - (log(x) + log(y))
sage: f(x=-1, y=i)
-2*I*pi
sage: f.simplify_log()
0
```

```
>>> from sage.all import *
>>> x,y = SR.var('x,y')
>>> f = log(x*y) - (log(x) + log(y))
>>> f(x=-Integer(1), y=i)
-2*I*pi
>>> f.simplify_log()
0
```

INPUT:

- `self` – expression to be simplified
- `algorithm` – (default: `None`) governs the condition on a and c which must be satisfied to contract expression $a\log(b) + c\log(d)$. Values are
  - `None` (use Maxima default, integers),
  - `'one'` (1 and -1),
  - `'ratios'` (rational numbers),
  - `'constants'` (constants),
  - `'all'` (all expressions).

ALGORITHM:

This uses the Maxima `logcontract()` command.

ALIAS:

log_simplify() and *simplify_log()* are the same.

EXAMPLES:

```
sage: x,y,t = var('x y t')
```

```
>>> from sage.all import *
>>> x,y,t = var('x y t')
```

Only two first terms are contracted in the following example; the logarithm with coefficient $\frac{1}{2}$ is not contracted:

```
sage: f = log(x)+2*log(y)+1/2*log(t)
sage: f.simplify_log()
log(x*y^2) + 1/2*log(t)
```

```
>>> from sage.all import *
>>> f = log(x)+Integer(2)*log(y)+Integer(1)/Integer(2)*log(t)
>>> f.simplify_log()
log(x*y^2) + 1/2*log(t)
```

To contract all terms in the previous example, we use the `'ratios'` `algorithm`:

```
sage: f.simplify_log(algorithm='ratios')
log(sqrt(t)*x*y^2)
```

```
>>> from sage.all import *
>>> f.simplify_log(algorithm='ratios')
log(sqrt(t)*x*y^2)
```

To contract terms with no coefficient (more precisely, with coefficients 1 and -1), we use the `'one'` `algorithm`:

```
sage: f = log(x)+2*log(y)-log(t)
sage: f.simplify_log('one')
2*log(y) + log(x/t)
```

```
>>> from sage.all import *
>>> f = log(x)+Integer(2)*log(y)-log(t)
>>> f.simplify_log('one')
2*log(y) + log(x/t)
```

```
sage: f = log(x)+log(y)-1/3*log((x+1))
sage: f.simplify_log()
log(x*y) - 1/3*log(x + 1)

sage: f.simplify_log('ratios')
log(x*y/(x + 1)^(1/3))
```

```
>>> from sage.all import *
>>> f = log(x)+log(y)-Integer(1)/Integer(3)*log((x+Integer(1)))
>>> f.simplify_log()
log(x*y) - 1/3*log(x + 1)

>>> f.simplify_log('ratios')
log(x*y/(x + 1)^(1/3))
```

π is an irrational number; to contract logarithms in the following example we have to set `algorithm` to `'constants'` or `'all'`:

```
sage: f = log(x)+log(y)-pi*log((x+1))
sage: f.simplify_log('constants')
log(x*y/(x + 1)^pi)
```

```
>>> from sage.all import *
>>> f = log(x)+log(y)-pi*log((x+Integer(1)))
>>> f.simplify_log('constants')
log(x*y/(x + 1)^pi)
```

`x*log(9)` is contracted only if `algorithm` is `'all'`:

```
sage: (x*log(9)).simplify_log()
2*x*log(3)
sage: (x*log(9)).simplify_log('all')
log(3^(2*x))
```

```
>>> from sage.all import *
>>> (x*log(Integer(9))).simplify_log()
2*x*log(3)
>>> (x*log(Integer(9))).simplify_log('all')
log(3^(2*x))
```

AUTHORS:

- Robert Marik (11-2009)

low_degree(*s*)

Return the exponent of the lowest power of `s` in `self`.

OUTPUT: integer

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y^10 + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + 2*sin(x*y)/x + x/y^10 + 100
sage: f.low_degree(x)
-1
sage: f.low_degree(y)
-10
sage: f.low_degree(sin(x*y))
0
sage: (x^3+y).low_degree(x)
0
sage: (x+x**2).low_degree(x)
1
```

```
>>> from sage.all import *
>>> var('x,y,a')
(x, y, a)
>>> f = Integer(100) + a*x + x**Integer(3)*sin(x*y) + x*y + x/y**Integer(10)␣
↪+ Integer(2)*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + 2*sin(x*y)/x + x/y^10 + 100
>>> f.low_degree(x)
-1
>>> f.low_degree(y)
-10
>>> f.low_degree(sin(x*y))
0
>>> (x**Integer(3)+y).low_degree(x)
0
>>> (x+x**Integer(2)).low_degree(x)
1
```

match (*pattern*)

Check if `self` matches the given pattern.

INPUT:

- `pattern` – a symbolic expression, possibly containing wildcards to match for

OUTPUT:

`None` if there is no match, or a dictionary mapping the wildcards to the matching values if a match was found. Note that the dictionary is empty if there were no wildcards in the given pattern.

See also http://www.ginac.de/tutorial/Pattern-matching-and-advanced-substitutions.html

EXAMPLES:

```
sage: var('x,y,z,a,b,c,d,f,g')
(x, y, z, a, b, c, d, f, g)
sage: w0 = SR.wild(0); w1 = SR.wild(1); w2 = SR.wild(2)
sage: ((x+y)^a).match((x+y)^a)  # no wildcards, so empty dict
{}
sage: print(((x+y)^a).match((x+y)^b))
None
```

(continues on next page)

(continued from previous page)

```
sage: t = ((x+y)^a).match(w0^w1)
sage: t[w0], t[w1]
(x + y, a)
sage: print(((x+y)^a).match(w0^w0))
None
sage: ((x+y)^(x+y)).match(w0^w0)
{$0: x + y}
sage: t = ((a+b)*(a+c)).match((a+w0)*(a+w1))
sage: set([t[w0], t[w1]]) == set([b, c])
True
sage: ((a+b)*(a+c)).match((w0+b)*(w0+c))
{$0: a}
sage: t = ((a+b)*(a+c)).match((w0+w1)*(w0+w2))
sage: t[w0]
a
sage: set([t[w1], t[w2]]) == set([b, c])
True
sage: t = ((a+b)*(a+c)).match((w0+w1)*(w1+w2))
sage: t[w1]
a
sage: set([t[w0], t[w2]]) == set([b, c])
True
sage: t = (a*(x+y)+a*z+b).match(a*w0+w1)
sage: s = set([t[w0], t[w1]])
sage: s == set([x+y, a*z+b]) or s == set([z, a*(x+y)+b])
True
sage: print((a+b+c+d+f+g).match(c))
None
sage: (a+b+c+d+f+g).has(c)
True
sage: (a+b+c+d+f+g).match(c+w0)
{$0: a + b + d + f + g}
sage: (a+b+c+d+f+g).match(c+g+w0)
{$0: a + b + d + f}
sage: (a+b).match(a+b+w0) # known bug
{$0: 0}
sage: print((a*b^2).match(a^w0*b^w1))
None
sage: (a*b^2).match(a*b^w1)
{$1: 2}
sage: (x*x.arctan2(x^2)).match(w0*w0.arctan2(w0^2))
{$0: x}
```

```
>>> from sage.all import *
>>> var('x,y,z,a,b,c,d,f,g')
(x, y, z, a, b, c, d, f, g)
>>> w0 = SR.wild(Integer(0)); w1 = SR.wild(Integer(1)); w2 = SR.
↪wild(Integer(2))
>>> ((x+y)**a).match((x+y)**a)  # no wildcards, so empty dict
{}
>>> print(((x+y)**a).match((x+y)**b))
None
```

(continues on next page)

(continued from previous page)

```
>>> t = ((x+y)**a).match(w0**w1)
>>> t[w0], t[w1]
(x + y, a)
>>> print(((x+y)**a).match(w0**w0))
None
>>> ((x+y)**(x+y)).match(w0**w0)
{$0: x + y}
>>> t = ((a+b)*(a+c)).match((a+w0)*(a+w1))
>>> set([t[w0], t[w1]]) == set([b, c])
True
>>> ((a+b)*(a+c)).match((w0+b)*(w0+c))
{$0: a}
>>> t = ((a+b)*(a+c)).match((w0+w1)*(w0+w2))
>>> t[w0]
a
>>> set([t[w1], t[w2]]) == set([b, c])
True
>>> t = ((a+b)*(a+c)).match((w0+w1)*(w1+w2))
>>> t[w1]
a
>>> set([t[w0], t[w2]]) == set([b, c])
True
>>> t = (a*(x+y)+a*z+b).match(a*w0+w1)
>>> s = set([t[w0], t[w1]])
>>> s == set([x+y, a*z+b]) or s == set([z, a*(x+y)+b])
True
>>> print((a+b+c+d+f+g).match(c))
None
>>> (a+b+c+d+f+g).has(c)
True
>>> (a+b+c+d+f+g).match(c+w0)
{$0: a + b + d + f + g}
>>> (a+b+c+d+f+g).match(c+g+w0)
{$0: a + b + d + f}
>>> (a+b).match(a+b+w0)  # known bug
{$0: 0}
>>> print((a*b**Integer(2)).match(a**w0*b**w1))
None
>>> (a*b**Integer(2)).match(a*b**w1)
{$1: 2}
>>> (x*x.arctan2(x**Integer(2))).match(w0*w0.arctan2(w0**Integer(2)))
{$0: x}
```

Beware that behind-the-scenes simplification can lead to surprising results in matching:

```
sage: print((x+x).match(w0+w1))
None
sage: t = x+x; t
2*x
sage: t.operator()
<function mul_vararg ...>
```

```
>>> from sage.all import *
>>> print((x+x).match(w0+w1))
None
>>> t = x+x; t
2*x
>>> t.operator()
<function mul_vararg ...>
```

Since asking to match w0+w1 looks for an addition operator, there is no match.

maxima_methods()

Provide easy access to maxima methods, converting the result to a Sage expression automatically.

EXAMPLES:

```
sage: t = log(sqrt(2) - 1) + log(sqrt(2) + 1); t
log(sqrt(2) + 1) + log(sqrt(2) - 1)
sage: res = t.maxima_methods().logcontract(); res
log((sqrt(2) + 1)*(sqrt(2) - 1))
sage: type(res)
<class 'sage.symbolic.expression.Expression'>
```

```
>>> from sage.all import *
>>> t = log(sqrt(Integer(2)) - Integer(1)) + log(sqrt(Integer(2)) +␣
↪Integer(1)); t
log(sqrt(2) + 1) + log(sqrt(2) - 1)
>>> res = t.maxima_methods().logcontract(); res
log((sqrt(2) + 1)*(sqrt(2) - 1))
>>> type(res)
<class 'sage.symbolic.expression.Expression'>
```

minpoly(**args*, ***kwds*)

Return the minimal polynomial of this symbolic expression.

EXAMPLES:

```
sage: golden_ratio.minpoly()
x^2 - x - 1
```

```
>>> from sage.all import *
>>> golden_ratio.minpoly()
x^2 - x - 1
```

mul(*hold=False*, **args*)

Return the product of the current expression and the given arguments.

To prevent automatic evaluation use the `hold` argument.

EXAMPLES:

```
sage: x.mul(x)
x^2
sage: x.mul(x, hold=True)
x*x
```

(continues on next page)

(continued from previous page)

```
sage: x.mul(x, (2+x), hold=True)
(x + 2)*x*x
sage: x.mul(x, (2+x), x, hold=True)
(x + 2)*x*x*x
sage: x.mul(x, (2+x), x, 2*x, hold=True)
(2*x)*(x + 2)*x*x*x
```

```
>>> from sage.all import *
>>> x.mul(x)
x^2
>>> x.mul(x, hold=True)
x*x
>>> x.mul(x, (Integer(2)+x), hold=True)
(x + 2)*x*x
>>> x.mul(x, (Integer(2)+x), x, hold=True)
(x + 2)*x*x*x
>>> x.mul(x, (Integer(2)+x), x, Integer(2)*x, hold=True)
(2*x)*(x + 2)*x*x*x
```

To then evaluate again, we use `unhold()`:

```
sage: a = x.mul(x, hold=True); a.unhold()
x^2
```

```
>>> from sage.all import *
>>> a = x.mul(x, hold=True); a.unhold()
x^2
```

multiply_both_sides(*x*, *checksign=None*)

Return a relation obtained by multiplying both sides of this relation by `x`.

Note

The *checksign* keyword argument is currently ignored and is included for backward compatibility reasons only.

EXAMPLES:

```
sage: var('x,y'); f = x + 3 < y - 2
(x, y)
sage: f.multiply_both_sides(7)
7*x + 21 < 7*y - 14
sage: f.multiply_both_sides(-1/2)
-1/2*x - 3/2 < -1/2*y + 1
sage: f*(-2/3)
-2/3*x - 2 < -2/3*y + 4/3
sage: f*(-pi)
-pi*(x + 3) < -pi*(y - 2)
```

```
>>> from sage.all import *
>>> var('x,y'); f = x + Integer(3) < y - Integer(2)
(x, y)
>>> f.multiply_both_sides(Integer(7))
7*x + 21 < 7*y - 14
>>> f.multiply_both_sides(-Integer(1)/Integer(2))
-1/2*x - 3/2 < -1/2*y + 1
>>> f*(-Integer(2)/Integer(3))
-2/3*x - 2 < -2/3*y + 4/3
>>> f*(-pi)
-pi*(x + 3) < -pi*(y - 2)
```

Since the direction of the inequality never changes when doing arithmetic with equations, you can multiply or divide the equation by a quantity with unknown sign:

```
sage: f*(1+I)
(I + 1)*x + 3*I + 3 < (I + 1)*y - 2*I - 2
sage: f = sqrt(2) + x == y^3
sage: f.multiply_both_sides(I)
I*x + I*sqrt(2) == I*y^3
sage: f.multiply_both_sides(-1)
-x - sqrt(2) == -y^3
```

```
>>> from sage.all import *
>>> f*(Integer(1)+I)
(I + 1)*x + 3*I + 3 < (I + 1)*y - 2*I - 2
>>> f = sqrt(Integer(2)) + x == y**Integer(3)
>>> f.multiply_both_sides(I)
I*x + I*sqrt(2) == I*y^3
>>> f.multiply_both_sides(-Integer(1))
-x - sqrt(2) == -y^3
```

Note that the direction of the following inequalities is not reversed:

```
sage: (x^3 + 1 > 2*sqrt(3)) * (-1)
-x^3 - 1 > -2*sqrt(3)
sage: (x^3 + 1 >= 2*sqrt(3)) * (-1)
-x^3 - 1 >= -2*sqrt(3)
sage: (x^3 + 1 <= 2*sqrt(3)) * (-1)
-x^3 - 1 <= -2*sqrt(3)
```

```
>>> from sage.all import *
>>> (x**Integer(3) + Integer(1) > Integer(2)*sqrt(Integer(3))) * (-Integer(1))
-x^3 - 1 > -2*sqrt(3)
>>> (x**Integer(3) + Integer(1) >= Integer(2)*sqrt(Integer(3))) * (-
↪Integer(1))
-x^3 - 1 >= -2*sqrt(3)
>>> (x**Integer(3) + Integer(1) <= Integer(2)*sqrt(Integer(3))) * (-
↪Integer(1))
-x^3 - 1 <= -2*sqrt(3)
```

negation()

Return the negated version of `self`.

This is the relation that is `False` iff `self` is `True`.

EXAMPLES:

```
sage: (x < 5).negation()
x >= 5
sage: (x == sin(3)).negation()
x != sin(3)
sage: (2*x >= sqrt(2)).negation()
2*x < sqrt(2)
```

```
>>> from sage.all import *
>>> (x < Integer(5)).negation()
x >= 5
>>> (x == sin(Integer(3))).negation()
x != sin(3)
>>> (Integer(2)*x >= sqrt(Integer(2))).negation()
2*x < sqrt(2)
```

nintegral(**args*, ***kwds*)

Compute the numerical integral of `self`.

Please see *sage.calculus.calculus.nintegral* for more details.

EXAMPLES:

```
sage: sin(x).nintegral(x,0,3)
(1.989992496600..., 2.209335488557...e-14, 21, 0)
```

```
>>> from sage.all import *
>>> sin(x).nintegral(x,Integer(0),Integer(3))
(1.989992496600..., 2.209335488557...e-14, 21, 0)
```

nintegrate(**args*, ***kwds*)

Compute the numerical integral of `self`.

Please see *sage.calculus.calculus.nintegral* for more details.

EXAMPLES:

```
sage: sin(x).nintegral(x,0,3)
(1.989992496600..., 2.209335488557...e-14, 21, 0)
```

```
>>> from sage.all import *
>>> sin(x).nintegral(x,Integer(0),Integer(3))
(1.989992496600..., 2.209335488557...e-14, 21, 0)
```

nops()

Return the number of operands of this expression.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: a.number_of_operands()
```

(continues on next page)

(continued from previous page)

```
0
sage: (a^2 + b^2 + (x+y)^2).number_of_operands()
3
sage: (a^2).number_of_operands()
2
sage: (a*b^2*c).number_of_operands()
3
```

```
>>> from sage.all import *
>>> var('a,b,c,x,y')
(a, b, c, x, y)
>>> a.number_of_operands()
0
>>> (a**Integer(2) + b**Integer(2) + (x+y)**Integer(2)).number_of_operands()
3
>>> (a**Integer(2)).number_of_operands()
2
>>> (a*b**Integer(2)*c).number_of_operands()
3
```

norm()

Return the complex norm of this symbolic expression, i.e., the expression times its complex conjugate. If $c = a + bi$ is a complex number, then the norm of c is defined as the product of c and its complex conjugate

$$\text{norm}(c) = \text{norm}(a + bi) = c \cdot \bar{c} = a^2 + b^2.$$

The norm of a complex number is different from its absolute value. The absolute value of a complex number is defined to be the square root of its norm. A typical use of the complex norm is in the integral domain $\mathbf{Z}[i]$ of Gaussian integers, where the norm of each Gaussian integer $c = a + bi$ is defined as its complex norm.

See also

`sage.misc.functional.norm()`

EXAMPLES:

```
sage: a = 1 + 2*I
sage: a.norm()
5
sage: a = sqrt(2) + 3^(1/3)*I; a
sqrt(2) + I*3^(1/3)
sage: a.norm()
3^(2/3) + 2
sage: CDF(a).norm()
4.080083823051...
sage: CDF(a.norm())
4.080083823051904
```

```
>>> from sage.all import *
>>> a = Integer(1) + Integer(2)*I
>>> a.norm()
```

(continues on next page)

(continued from previous page)

```
5
>>> a = sqrt(Integer(2)) + Integer(3)**(Integer(1)/Integer(3))*I; a
sqrt(2) + I*3^(1/3)
>>> a.norm()
3^(2/3) + 2
>>> CDF(a).norm()
4.080083823051...
>>> CDF(a.norm())
4.080083823051904
```

normalize()

Return this expression normalized as a fraction.

> **See also**
>
> *numerator()*, *denominator()*, *numerator_denominator()*, *combine()*

EXAMPLES:

```
sage: var('x, y, a, b, c')
(x, y, a, b, c)
sage: g = x + y/(x + 2)
sage: g.normalize()
(x^2 + 2*x + y)/(x + 2)

sage: f = x*(x-1)/(x^2 - 7) + y^2/(x^2-7) + 1/(x+1) + b/a + c/a
sage: f.normalize()
(a*x^3 + b*x^3 + c*x^3 + a*x*y^2 + a*x^2 + b*x^2 + c*x^2 +
        a*y^2 - a*x - 7*b*x - 7*c*x - 7*a - 7*b - 7*c)/((x^2 -
            7)*a*(x + 1))
```

```
>>> from sage.all import *
>>> var('x, y, a, b, c')
(x, y, a, b, c)
>>> g = x + y/(x + Integer(2))
>>> g.normalize()
(x^2 + 2*x + y)/(x + 2)

>>> f = x*(x-Integer(1))/(x**Integer(2) - Integer(7)) + y**Integer(2)/
↪(x**Integer(2)-Integer(7)) + Integer(1)/(x+Integer(1)) + b/a + c/a
>>> f.normalize()
(a*x^3 + b*x^3 + c*x^3 + a*x*y^2 + a*x^2 + b*x^2 + c*x^2 +
        a*y^2 - a*x - 7*b*x - 7*c*x - 7*a - 7*b - 7*c)/((x^2 -
            7)*a*(x + 1))
```

ALGORITHM: Uses GiNaC.

number_of_arguments()

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = x + y
sage: f.number_of_arguments()
2

sage: g = f.function(x)
sage: g.number_of_arguments()
1
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> f = x + y
>>> f.number_of_arguments()
2

>>> g = f.function(x)
>>> g.number_of_arguments()
1
```

```
sage: x,y,z = var('x,y,z')
sage: (x+y).number_of_arguments()
2
sage: (x+1).number_of_arguments()
1
sage: (sin(x)+1).number_of_arguments()
1
sage: (sin(z)+x+y).number_of_arguments()
3
sage: (sin(x+y)).number_of_arguments()
2
```

```
>>> from sage.all import *
>>> x,y,z = var('x,y,z')
>>> (x+y).number_of_arguments()
2
>>> (x+Integer(1)).number_of_arguments()
1
>>> (sin(x)+Integer(1)).number_of_arguments()
1
>>> (sin(z)+x+y).number_of_arguments()
3
>>> (sin(x+y)).number_of_arguments()
2
```

number_of_operands()

Return the number of operands of this expression.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: a.number_of_operands()
```

(continues on next page)

(continued from previous page)

```
0
sage: (a^2 + b^2 + (x+y)^2).number_of_operands()
3
sage: (a^2).number_of_operands()
2
sage: (a*b^2*c).number_of_operands()
3
```

```
>>> from sage.all import *
>>> var('a,b,c,x,y')
(a, b, c, x, y)
>>> a.number_of_operands()
0
>>> (a**Integer(2) + b**Integer(2) + (x+y)**Integer(2)).number_of_operands()
3
>>> (a**Integer(2)).number_of_operands()
2
>>> (a*b**Integer(2)*c).number_of_operands()
3
```

numerator(*normalize=True*)

Return the numerator of this symbolic expression.

INPUT:

- `normalize` – boolean (default: `True`)

If `normalize` is `True`, the expression is first normalized to have it as a fraction before getting the numerator.

If `normalize` is `False`, the expression is kept and if it is not a quotient, then this will return the expression itself.

See also

normalize(), *denominator()*, *numerator_denominator()*, *combine()*

EXAMPLES:

```
sage: a, x, y = var('a,x,y')
sage: f = x*(x-a)/((x^2 - y)*(x-a)); f
x/(x^2 - y)
sage: f.numerator()
x
sage: f.denominator()
x^2 - y
sage: f.numerator(normalize=False)
x
sage: f.denominator(normalize=False)
x^2 - y

sage: y = var('y')
sage: g = x + y/(x + 2); g
x + y/(x + 2)
```

(continues on next page)

(continued from previous page)

```
sage: g.numerator()
x^2 + 2*x + y
sage: g.denominator()
x + 2
sage: g.numerator(normalize=False)
x + y/(x + 2)
sage: g.denominator(normalize=False)
1
```

```
>>> from sage.all import *
>>> a, x, y = var('a,x,y')
>>> f = x*(x-a)/((x**Integer(2) - y)*(x-a)); f
x/(x^2 - y)
>>> f.numerator()
x
>>> f.denominator()
x^2 - y
>>> f.numerator(normalize=False)
x
>>> f.denominator(normalize=False)
x^2 - y

>>> y = var('y')
>>> g = x + y/(x + Integer(2)); g
x + y/(x + 2)
>>> g.numerator()
x^2 + 2*x + y
>>> g.denominator()
x + 2
>>> g.numerator(normalize=False)
x + y/(x + 2)
>>> g.denominator(normalize=False)
1
```

numerator_denominator(*normalize=True*)

Return the numerator and the denominator of this symbolic expression.

INPUT:

- `normalize` – boolean (default: `True`)

If `normalize` is `True`, the expression is first normalized to have it as a fraction before getting the numerator and denominator.

If `normalize` is `False`, the expression is kept and if it is not a quotient, then this will return the expression itself together with 1.

See also

normalize(), *numerator()*, *denominator()*, *combine()*

EXAMPLES:

```
sage: x, y, a = var("x y a")
sage: ((x+y)^2/(x-y)^3*x^3).numerator_denominator()
((x + y)^2*x^3, (x - y)^3)

sage: ((x+y)^2/(x-y)^3*x^3).numerator_denominator(False)
((x + y)^2*x^3, (x - y)^3)

sage: g = x + y/(x + 2)
sage: g.numerator_denominator()
(x^2 + 2*x + y, x + 2)
sage: g.numerator_denominator(normalize=False)
(x + y/(x + 2), 1)

sage: g = x^2*(x + 2)
sage: g.numerator_denominator()
((x + 2)*x^2, 1)
sage: g.numerator_denominator(normalize=False)
((x + 2)*x^2, 1)
```

```
>>> from sage.all import *
>>> x, y, a = var("x y a")
>>> ((x+y)**Integer(2)/(x-y)**Integer(3)*x**Integer(3)).numerator_
↪denominator()
((x + y)^2*x^3, (x - y)^3)

>>> ((x+y)**Integer(2)/(x-y)**Integer(3)*x**Integer(3)).numerator_
↪denominator(False)
((x + y)^2*x^3, (x - y)^3)

>>> g = x + y/(x + Integer(2))
>>> g.numerator_denominator()
(x^2 + 2*x + y, x + 2)
>>> g.numerator_denominator(normalize=False)
(x + y/(x + 2), 1)

>>> g = x**Integer(2)*(x + Integer(2))
>>> g.numerator_denominator()
((x + 2)*x^2, 1)
>>> g.numerator_denominator(normalize=False)
((x + 2)*x^2, 1)
```

numerical_approx(*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
```

```
>>> from sage.all import *
>>> sin(x).subs(x=Integer(5)).n()
-0.958924274663138
>>> sin(x).subs(x=Integer(5)).n(Integer(100))
-0.95892427466313846889315440616
>>> sin(x).subs(x=Integer(5)).n(digits=Integer(50))
-0.95892427466313846889315440615599397335246154396460
>>> zeta(x).subs(x=Integer(2)).numerical_approx(digits=Integer(50))
1.6449340668482264364724151666460251892189499012068

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

op

Provide access to the operands of an expression through a property.

EXAMPLES:

```
sage: t = 1+x+x^2
sage: t.op
Operands of x^2 + x + 1
sage: x.op
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
TypeError: expressions containing only a numeric coefficient,
constant or symbol have no operands
sage: t.op[0]
x^2
```

```
>>> from sage.all import *
>>> t = Integer(1)+x+x**Integer(2)
>>> t.op
Operands of x^2 + x + 1
>>> x.op
Traceback (most recent call last):
...
TypeError: expressions containing only a numeric coefficient,
constant or symbol have no operands
>>> t.op[Integer(0)]
x^2
```

Indexing directly with `t[1]` causes problems with numpy types.

> sage: t[1] Traceback (most recent call last): … TypeError: 'sage.symbolic.expression.Expression' object …

operands()

Return a list containing the operands of this expression.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: (a^2 + b^2 + (x+y)^2).operands()
[a^2, b^2, (x + y)^2]
sage: (a^2).operands()
[a, 2]
sage: (a*b^2*c).operands()
[a, b^2, c]
```

```
>>> from sage.all import *
>>> var('a,b,c,x,y')
(a, b, c, x, y)
>>> (a**Integer(2) + b**Integer(2) + (x+y)**Integer(2)).operands()
[a^2, b^2, (x + y)^2]
>>> (a**Integer(2)).operands()
[a, 2]
>>> (a*b**Integer(2)*c).operands()
[a, b^2, c]
```

operator()

Return the topmost operator in this expression.

EXAMPLES:

```
sage: x,y,z = var('x,y,z')
sage: (x+y).operator()
```

(continues on next page)

(continued from previous page)

```
<function add_vararg ...>
sage: (x^y).operator()
<built-in function pow>
sage: (x^y * z).operator()
<function mul_vararg ...>
sage: (x < y).operator()
<built-in function lt>

sage: abs(x).operator()
abs
sage: r = gamma(x).operator(); type(r)
<class 'sage.functions.gamma.Function_gamma'>

sage: psi = function('psi', nargs=1)
sage: psi(x).operator()
psi

sage: r = psi(x).operator()
sage: r == psi
True

sage: f = function('f', nargs=1, conjugate_func=lambda self, x: 2*x)
sage: nf = f(x).operator()
sage: nf(x).conjugate()
2*x

sage: f = function('f')
sage: a = f(x).diff(x); a
diff(f(x), x)
sage: a.operator()
D[0](f)
```

```
>>> from sage.all import *
>>> x,y,z = var('x,y,z')
>>> (x+y).operator()
<function add_vararg ...>
>>> (x**y).operator()
<built-in function pow>
>>> (x**y * z).operator()
<function mul_vararg ...>
>>> (x < y).operator()
<built-in function lt>

>>> abs(x).operator()
abs
>>> r = gamma(x).operator(); type(r)
<class 'sage.functions.gamma.Function_gamma'>

>>> psi = function('psi', nargs=Integer(1))
>>> psi(x).operator()
psi
```

(continues on next page)

(continued from previous page)

```
>>> r = psi(x).operator()
>>> r == psi
True

>>> f = function('f', nargs=Integer(1), conjugate_func=lambda self, x:␣
↪Integer(2)*x)
>>> nf = f(x).operator()
>>> nf(x).conjugate()
2*x

>>> f = function('f')
>>> a = f(x).diff(x); a
diff(f(x), x)
>>> a.operator()
D[0](f)
```

partial_fraction(*var=None*)

Return the partial fraction expansion of `self` with respect to the given variable.

INPUT:

- `var` – variable name or string (default: first variable)

OUTPUT: a symbolic expression

See also:

partial_fraction_decomposition()

EXAMPLES:

```
sage: f = x^2/(x+1)^3
sage: f.partial_fraction()
1/(x + 1) - 2/(x + 1)^2 + 1/(x + 1)^3
```

```
>>> from sage.all import *
>>> f = x**Integer(2)/(x+Integer(1))**Integer(3)
>>> f.partial_fraction()
1/(x + 1) - 2/(x + 1)^2 + 1/(x + 1)^3
```

Notice that the first variable in the expression is used by default:

```
sage: y = var('y')
sage: f = y^2/(y+1)^3
sage: f.partial_fraction()
1/(y + 1) - 2/(y + 1)^2 + 1/(y + 1)^3

sage: f = y^2/(y+1)^3 + x/(x-1)^3
sage: f.partial_fraction()
y^2/(y^3 + 3*y^2 + 3*y + 1) + 1/(x - 1)^2 + 1/(x - 1)^3
```

```
>>> from sage.all import *
>>> y = var('y')
>>> f = y**Integer(2)/(y+Integer(1))**Integer(3)
>>> f.partial_fraction()
1/(y + 1) - 2/(y + 1)^2 + 1/(y + 1)^3

>>> f = y**Integer(2)/(y+Integer(1))**Integer(3) + x/(x-
↪Integer(1))**Integer(3)
>>> f.partial_fraction()
y^2/(y^3 + 3*y^2 + 3*y + 1) + 1/(x - 1)^2 + 1/(x - 1)^3
```

You can explicitly specify which variable is used:

```
sage: f.partial_fraction(y)
x/(x^3 - 3*x^2 + 3*x - 1) + 1/(y + 1) - 2/(y + 1)^2 + 1/(y + 1)^3
```

```
>>> from sage.all import *
>>> f.partial_fraction(y)
x/(x^3 - 3*x^2 + 3*x - 1) + 1/(y + 1) - 2/(y + 1)^2 + 1/(y + 1)^3
```

partial_fraction_decomposition(*var=None*)

Return the partial fraction decomposition of `self` with respect to the given variable.

INPUT:

- `var` – variable name or string (default: first variable)

OUTPUT: list of symbolic expressions

See also

partial_fraction()

EXAMPLES:

```
sage: f = x^2/(x+1)^3
sage: f.partial_fraction_decomposition()
[1/(x + 1), -2/(x + 1)^2, (x + 1)^(-3)]
sage: (4+f).partial_fraction_decomposition()
[1/(x + 1), -2/(x + 1)^2, (x + 1)^(-3), 4]
```

```
>>> from sage.all import *
>>> f = x**Integer(2)/(x+Integer(1))**Integer(3)
>>> f.partial_fraction_decomposition()
[1/(x + 1), -2/(x + 1)^2, (x + 1)^(-3)]
>>> (Integer(4)+f).partial_fraction_decomposition()
[1/(x + 1), -2/(x + 1)^2, (x + 1)^(-3), 4]
```

Notice that the first variable in the expression is used by default:

```
sage: y = var('y')
sage: f = y^2/(y+1)^3
sage: f.partial_fraction_decomposition()
```

(continues on next page)

(continued from previous page)

```
[1/(y + 1), -2/(y + 1)^2, (y + 1)^(-3)]

sage: f = y^2/(y+1)^3 + x/(x-1)^3
sage: f.partial_fraction_decomposition()
[y^2/(y^3 + 3*y^2 + 3*y + 1), (x - 1)^(-2), (x - 1)^(-3)]
```

```
>>> from sage.all import *
>>> y = var('y')
>>> f = y**Integer(2)/(y+Integer(1))**Integer(3)
>>> f.partial_fraction_decomposition()
[1/(y + 1), -2/(y + 1)^2, (y + 1)^(-3)]

>>> f = y**Integer(2)/(y+Integer(1))**Integer(3) + x/(x-
↪Integer(1))**Integer(3)
>>> f.partial_fraction_decomposition()
[y^2/(y^3 + 3*y^2 + 3*y + 1), (x - 1)^(-2), (x - 1)^(-3)]
```

You can explicitly specify which variable is used:

```
sage: f.partial_fraction_decomposition(y)
[1/(y + 1), -2/(y + 1)^2, (y + 1)^(-3), x/(x^3 - 3*x^2 + 3*x - 1)]
```

```
>>> from sage.all import *
>>> f.partial_fraction_decomposition(y)
[1/(y + 1), -2/(y + 1)^2, (y + 1)^(-3), x/(x^3 - 3*x^2 + 3*x - 1)]
```

plot(**args*, ***kwds*)

Plot a symbolic expression. All arguments are passed onto the standard `plot()` command.

EXAMPLES:

This displays a straight line:

```
sage: sin(2).plot((x,0,3))                                              #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> sin(Integer(2)).plot((x,Integer(0),Integer(3)))                      ␣
↪                         # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

This draws a red oscillatory curve:

```
sage: sin(x^2).plot((x,0,2*pi), rgbcolor=(1,0,0))                       #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> sin(x**Integer(2)).plot((x,Integer(0),Integer(2)*pi),␣
↪rgbcolor=(Integer(1),Integer(0),Integer(0)))                        #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

Another plot using the variable theta:

```
sage: var('theta')
theta
sage: (cos(theta) - erf(theta)).plot((theta,-2*pi,2*pi))                  #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> var('theta')
theta
>>> (cos(theta) - erf(theta)).plot((theta,-Integer(2)*pi,Integer(2)*pi))       ␣
↪               # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

A very thick green plot with a frame:

```
sage: sin(x).plot((x, -4*pi, 4*pi),                                        #␣
↪needs sage.plot
....:             thickness=20, rgbcolor=(0,0.7,0)).show(frame=True)
```

```
>>> from sage.all import *
>>> sin(x).plot((x, -Integer(4)*pi, Integer(4)*pi),                           ␣
↪               # needs sage.plot
...             thickness=Integer(20), rgbcolor=(Integer(0),RealNumber('0.7'),
↪Integer(0))).show(frame=True)
```

You can embed 2d plots in 3d space as follows:

```
sage: plot(sin(x^2), (x, -pi, pi), thickness=2).plot3d(z=1)          # long␣
↪time, needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> plot(sin(x**Integer(2)), (x, -pi, pi), thickness=Integer(2)).
↪plot3d(z=Integer(1))          # long time, needs sage.plot
Graphics3d Object
```

A more complicated family:

```
sage: G = sum(plot(sin(n*x), (x, -2*pi, 2*pi)).plot3d(z=n)                 #␣
↪needs sage.plot
....:         for n in [0,0.1,..1])
sage: G.show(frame_aspect_ratio=[1,1,1/2])  # long time (5s on sage.math,␣
↪2012), needs sage.plot
```

```
>>> from sage.all import *
>>> G = sum(plot(sin(n*x), (x, -Integer(2)*pi, Integer(2)*pi)).plot3d(z=n)    ␣
↪               # needs sage.plot
...         for n in (ellipsis_range(Integer(0),RealNumber('0.1'),Ellipsis,
↪Integer(1))))
>>> G.show(frame_aspect_ratio=[Integer(1),Integer(1),Integer(1)/Integer(2)])
↪# long time (5s on sage.math, 2012), needs sage.plot
```

A plot involving the floor function:

```
sage: plot(1.0 - x * floor(1/x), (x,0.00001,1.0))                        #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(RealNumber('1.0') - x * floor(Integer(1)/x), (x,RealNumber('0.00001
↪'),RealNumber('1.0')))                           # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

Sage used to allow symbolic functions with “no arguments”; this no longer works:

```
sage: plot(2*sin, -4, 4)                                                 #␣
↪needs sage.plot
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
'Integer Ring' and '<class 'sage.functions.trig.Function_sin'>'
```

```
>>> from sage.all import *
>>> plot(Integer(2)*sin, -Integer(4), Integer(4))                         ␣
↪                       # needs sage.plot
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
'Integer Ring' and '<class 'sage.functions.trig.Function_sin'>'
```

You should evaluate the function first:

```
sage: plot(2*sin(x), -4, 4)                                              #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(Integer(2)*sin(x), -Integer(4), Integer(4))                      ␣
↪                       # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

poly(*x=None*)

Express this symbolic expression as a polynomial in *x*. If this is not a polynomial in *x*, then some coefficients may be functions of *x*.

> **Warning**
>
> This is different from *polynomial()* which returns a Sage polynomial over a given base ring.

EXAMPLES:

```
sage: var('a, x')
(a, x)
sage: p = expand((x-a*sqrt(2))^2 + x + 1); p
```

(continues on next page)

(continued from previous page)

```
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
sage: p.poly(a)
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
sage: bool(p.poly(a) == (x-a*sqrt(2))^2 + x + 1)
True
sage: p.poly(x)
2*a^2 - (2*sqrt(2)*a - 1)*x + x^2 + 1
```

```
>>> from sage.all import *
>>> var('a, x')
(a, x)
>>> p = expand((x-a*sqrt(Integer(2)))**Integer(2) + x + Integer(1)); p
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
>>> p.poly(a)
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
>>> bool(p.poly(a) == (x-a*sqrt(Integer(2)))**Integer(2) + x + Integer(1))
True
>>> p.poly(x)
2*a^2 - (2*sqrt(2)*a - 1)*x + x^2 + 1
```

polynomial(*base_ring=None*, *ring=None*)

Return this symbolic expression as an algebraic polynomial over the given base ring, if possible.

The point of this function is that it converts purely symbolic polynomials into optimised algebraic polynomials over a given base ring.

You can specify either the base ring (`base_ring`) you want the output polynomial to be over, or you can specify the full polynomial ring (`ring`) you want the output polynomial to be an element of.

INPUT:

- `base_ring` – (optional) the base ring for the polynomial
- `ring` – (optional) the parent for the polynomial

Warning

This is different from `poly()` which is used to rewrite `self` as a polynomial in terms of one of the variables.

EXAMPLES:

```
sage: f = x^2 -2/3*x + 1
sage: f.polynomial(QQ)
x^2 - 2/3*x + 1
sage: f.polynomial(GF(19))
x^2 + 12*x + 1
```

```
>>> from sage.all import *
>>> f = x**Integer(2) -Integer(2)/Integer(3)*x + Integer(1)
>>> f.polynomial(QQ)
x^2 - 2/3*x + 1
```

(continues on next page)

(continued from previous page)

```
>>> f.polynomial(GF(Integer(19)))
x^2 + 12*x + 1
```

Polynomials can be useful for getting the coefficients of an expression:

```
sage: g = 6*x^2 - 5
sage: g.coefficients()
[[-5, 0], [6, 2]]
sage: g.polynomial(QQ).list()
[-5, 0, 6]
sage: g.polynomial(QQ).monomial_coefficients()
{0: -5, 2: 6}
```

```
>>> from sage.all import *
>>> g = Integer(6)*x**Integer(2) - Integer(5)
>>> g.coefficients()
[[-5, 0], [6, 2]]
>>> g.polynomial(QQ).list()
[-5, 0, 6]
>>> g.polynomial(QQ).monomial_coefficients()
{0: -5, 2: 6}
```

```
sage: f = x^2*e + x + pi/e
sage: f.polynomial(RDF)   # abs tol 5e-16
2.718281828459045*x^2 + x + 1.1557273497909217
sage: g = f.polynomial(RR); g
2.71828182845905*x^2 + x + 1.15572734979092
sage: g.parent()
Univariate Polynomial Ring in x over Real Field with 53 bits of precision
sage: f.polynomial(RealField(100))
2.7182818284590452353602874714*x^2 + x + 1.1557273497909217179100931833
sage: f.polynomial(CDF)   # abs tol 5e-16
2.718281828459045*x^2 + x + 1.1557273497909217
sage: f.polynomial(CC)
2.71828182845905*x^2 + x + 1.15572734979092
```

```
>>> from sage.all import *
>>> f = x**Integer(2)*e + x + pi/e
>>> f.polynomial(RDF)   # abs tol 5e-16
2.718281828459045*x^2 + x + 1.1557273497909217
>>> g = f.polynomial(RR); g
2.71828182845905*x^2 + x + 1.15572734979092
>>> g.parent()
Univariate Polynomial Ring in x over Real Field with 53 bits of precision
>>> f.polynomial(RealField(Integer(100)))
2.7182818284590452353602874714*x^2 + x + 1.1557273497909217179100931833
>>> f.polynomial(CDF)   # abs tol 5e-16
2.718281828459045*x^2 + x + 1.1557273497909217
>>> f.polynomial(CC)
2.71828182845905*x^2 + x + 1.15572734979092
```

We coerce a multivariate polynomial with complex symbolic coefficients:

```
sage: x, y, n = var('x, y, n')
sage: f = pi^3*x - y^2*e - I; f
pi^3*x - y^2*e - I
sage: f.polynomial(CDF)   # abs tol 1e-15
(-2.718281828459045)*y^2 + 31.006276680299816*x - 1.0*I
sage: f.polynomial(CC)
(-2.71828182845905)*y^2 + 31.0062766802998*x - 1.00000000000000*I
sage: f.polynomial(ComplexField(70))
(-2.7182818284590452354)*y^2 + 31.006276680299820175*x - 1.
↪0000000000000000000*I
```

```
>>> from sage.all import *
>>> x, y, n = var('x, y, n')
>>> f = pi**Integer(3)*x - y**Integer(2)*e - I; f
pi^3*x - y^2*e - I
>>> f.polynomial(CDF)   # abs tol 1e-15
(-2.718281828459045)*y^2 + 31.006276680299816*x - 1.0*I
>>> f.polynomial(CC)
(-2.71828182845905)*y^2 + 31.0062766802998*x - 1.00000000000000*I
>>> f.polynomial(ComplexField(Integer(70)))
(-2.7182818284590452354)*y^2 + 31.006276680299820175*x - 1.
↪0000000000000000000*I
```

Another polynomial:

```
sage: f = sum((e*I)^n*x^n for n in range(5)); f
x^4*e^4 - I*x^3*e^3 - x^2*e^2 + I*x*e + 1
sage: f.polynomial(CDF)    # abs tol 5e-16
54.598150033144236*x^4 - 20.085536923187668*I*x^3 - 7.38905609893065*x^2
 + 2.718281828459045*I*x + 1.0
sage: f.polynomial(CC)
54.5981500331442*x^4 - 20.0855369231877*I*x^3 - 7.38905609893065*x^2
 + 2.71828182845905*I*x + 1.00000000000000
```

```
>>> from sage.all import *
>>> f = sum((e*I)**n*x**n for n in range(Integer(5))); f
x^4*e^4 - I*x^3*e^3 - x^2*e^2 + I*x*e + 1
>>> f.polynomial(CDF)    # abs tol 5e-16
54.598150033144236*x^4 - 20.085536923187668*I*x^3 - 7.38905609893065*x^2
 + 2.718281828459045*I*x + 1.0
>>> f.polynomial(CC)
54.5981500331442*x^4 - 20.0855369231877*I*x^3 - 7.38905609893065*x^2
 + 2.71828182845905*I*x + 1.00000000000000
```

A multivariate polynomial over a finite field:

```
sage: f = (3*x^5 - 5*y^5)^7; f
(3*x^5 - 5*y^5)^7
sage: g = f.polynomial(GF(7)); g
3*x^35 + 2*y^35
sage: parent(g)
Multivariate Polynomial Ring in x, y over Finite Field of size 7
```

```
>>> from sage.all import *
>>> f = (Integer(3)*x**Integer(5) - Integer(5)*y**Integer(5))**Integer(7); f
(3*x^5 - 5*y^5)^7
>>> g = f.polynomial(GF(Integer(7))); g
3*x^35 + 2*y^35
>>> parent(g)
Multivariate Polynomial Ring in x, y over Finite Field of size 7
```

We check to make sure constants are converted appropriately:

```
sage: (pi*x).polynomial(SR)
pi*x
```

```
>>> from sage.all import *
>>> (pi*x).polynomial(SR)
pi*x
```

Using the `ring` parameter, you can also create polynomials rings over the symbolic ring where only certain variables are considered generators of the polynomial ring and the others are considered "constants":

```
sage: a, x, y = var('a,x,y')
sage: f = a*x^10*y+3*x
sage: B = f.polynomial(ring=SR['x,y'])
sage: B.coefficients()
[a, 3]
```

```
>>> from sage.all import *
>>> a, x, y = var('a,x,y')
>>> f = a*x**Integer(10)*y+Integer(3)*x
>>> B = f.polynomial(ring=SR['x,y'])
>>> B.coefficients()
[a, 3]
```

power(*exp*, *hold=False*)

Return the current expression to the power `exp`.

To prevent automatic evaluation use the `hold` argument.

EXAMPLES:

```
sage: (x^2).power(2)
x^4
sage: (x^2).power(2, hold=True)
(x^2)^2
```

```
>>> from sage.all import *
>>> (x**Integer(2)).power(Integer(2))
x^4
>>> (x**Integer(2)).power(Integer(2), hold=True)
(x^2)^2
```

To then evaluate again, we use *unhold()*:

```
sage: a = (x^2).power(2, hold=True); a.unhold()
x^4
```

```
>>> from sage.all import *
>>> a = (x**Integer(2)).power(Integer(2), hold=True); a.unhold()
x^4
```

power_series(*base_ring*)

Return algebraic power series associated to this symbolic expression, which must be a polynomial in one variable, with coefficients coercible to the base ring.

The power series is truncated one more than the degree.

EXAMPLES:

```
sage: theta = var('theta')
sage: f = theta^3 + (1/3)*theta - 17/3
sage: g = f.power_series(QQ); g
-17/3 + 1/3*theta + theta^3 + O(theta^4)
sage: g^3
-4913/27 + 289/9*theta - 17/9*theta^2 + 2602/27*theta^3 + O(theta^4)
sage: g.parent()
Power Series Ring in theta over Rational Field
```

```
>>> from sage.all import *
>>> theta = var('theta')
>>> f = theta**Integer(3) + (Integer(1)/Integer(3))*theta - Integer(17)/
↪Integer(3)
>>> g = f.power_series(QQ); g
-17/3 + 1/3*theta + theta^3 + O(theta^4)
>>> g**Integer(3)
-4913/27 + 289/9*theta - 17/9*theta^2 + 2602/27*theta^3 + O(theta^4)
>>> g.parent()
Power Series Ring in theta over Rational Field
```

primitive_part(*s*)

Return the primitive polynomial of this expression when considered as a polynomial in `s`.

See also *unit()*, *content()*, and *unit_content_primitive()*.

INPUT:

- `s` – a symbolic expression

OUTPUT:

The primitive polynomial as a symbolic expression. It is defined as the quotient by the *unit()* and *content()* parts (with respect to the variable `s`).

EXAMPLES:

```
sage: (2*x+4).primitive_part(x)
x + 2
sage: (2*x+1).primitive_part(x)
2*x + 1
sage: (2*x+1/2).primitive_part(x)
```

(continues on next page)

(continued from previous page)

```
4*x + 1
sage: var('y')
y
sage: (2*x + 4*sin(y)).primitive_part(sin(y))
x + 2*sin(y)
```

```
>>> from sage.all import *
>>> (Integer(2)*x+Integer(4)).primitive_part(x)
x + 2
>>> (Integer(2)*x+Integer(1)).primitive_part(x)
2*x + 1
>>> (Integer(2)*x+Integer(1)/Integer(2)).primitive_part(x)
4*x + 1
>>> var('y')
y
>>> (Integer(2)*x + Integer(4)*sin(y)).primitive_part(sin(y))
x + 2*sin(y)
```

prod(**args*, ***kwds*)

Return the symbolic product $\prod_{v=a}^{b}$ `self`.

This is the product respect to the variable v with endpoints a and b.

INPUT:

- `expression` – a symbolic expression
- `v` – a variable or variable name
- `a` – lower endpoint of the product
- `b` – upper endpoint of the product
- `algorithm` – (default: `'maxima'`) one of
  - `'maxima'` – use Maxima (the default)
  - `'giac'` – (optional) use Giac
  - `'sympy'` – use SymPy
- `hold` – boolean (default: `False`); if `True`, don't evaluate

pyobject()

Get the underlying Python object.

OUTPUT:

The Python object corresponding to this expression, assuming this expression is a single numerical value or an infinity representable in Python. Otherwise, a `TypeError` is raised.

EXAMPLES:

```
sage: var('x')
x
sage: b = -17.3
sage: a = SR(b)
sage: a.pyobject()
-17.3000000000000
```

(continues on next page)

(continued from previous page)

```
sage: a.pyobject() is b
True
```

```
>>> from sage.all import *
>>> var('x')
x
>>> b = -RealNumber('17.3')
>>> a = SR(b)
>>> a.pyobject()
-17.3000000000000
>>> a.pyobject() is b
True
```

Integers and Rationals are converted internally though, so you won't get back the same object:

```
sage: b = -17/3
sage: a = SR(b)
sage: a.pyobject()
-17/3
sage: a.pyobject() is b
False
```

```
>>> from sage.all import *
>>> b = -Integer(17)/Integer(3)
>>> a = SR(b)
>>> a.pyobject()
-17/3
>>> a.pyobject() is b
False
```

rational_expand(*side=None*)

Expand this symbolic expression. Products of sums and exponentiated sums are multiplied out, numerators of rational expressions which are sums are split into their respective terms, and multiplications are distributed over addition at all levels.

EXAMPLES:

We expand the expression $(x-y)^5$ using both method and functional notation.

```
sage: x,y = var('x,y')
sage: a = (x-y)^5
sage: a.expand()
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
sage: expand(a)
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> a = (x-y)**Integer(5)
>>> a.expand()
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
>>> expand(a)
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
```

We expand some other expressions:

```
sage: expand((x-1)^3/(y-1))
x^3/(y - 1) - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)
sage: expand((x+sin((x+y)^2))^2)
x^2 + 2*x*sin(x^2 + 2*x*y + y^2) + sin(x^2 + 2*x*y + y^2)^2
```

```
>>> from sage.all import *
>>> expand((x-Integer(1))**Integer(3)/(y-Integer(1)))
x^3/(y - 1) - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)
>>> expand((x+sin((x+y)**Integer(2)))**Integer(2))
x^2 + 2*x*sin(x^2 + 2*x*y + y^2) + sin(x^2 + 2*x*y + y^2)^2
```

Observe that *expand()* also expands function arguments:

```
sage: f(x) = function('f')(x)
sage: fx = f(x*(x+1)); fx
f((x + 1)*x)
sage: fx.expand()
f(x^2 + x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(function('f')(x)).function(x)
>>> fx = f(x*(x+Integer(1))); fx
f((x + 1)*x)
>>> fx.expand()
f(x^2 + x)
```

We can expand individual sides of a relation:

```
sage: a = (16*x-13)^2 == (3*x+5)^2/2
sage: a.expand()
256*x^2 - 416*x + 169 == 9/2*x^2 + 15*x + 25/2
sage: a.expand('left')
256*x^2 - 416*x + 169 == 1/2*(3*x + 5)^2
sage: a.expand('right')
(16*x - 13)^2 == 9/2*x^2 + 15*x + 25/2
```

```
>>> from sage.all import *
>>> a = (Integer(16)*x-Integer(13))**Integer(2) ==␣
↪(Integer(3)*x+Integer(5))**Integer(2)/Integer(2)
>>> a.expand()
256*x^2 - 416*x + 169 == 9/2*x^2 + 15*x + 25/2
>>> a.expand('left')
256*x^2 - 416*x + 169 == 1/2*(3*x + 5)^2
>>> a.expand('right')
(16*x - 13)^2 == 9/2*x^2 + 15*x + 25/2
```

rational_simplify(*algorithm='full'*, *map=False*)

Simplify rational expressions.

INPUT:

- `self` – symbolic expression

- `algorithm` – (default: `'full'`) string which switches the algorithm for simplifications. Possible values are
  - `'simple'` (simplify rational functions into quotient of two polynomials)
  - `'full'` (apply repeatedly, if necessary)
  - `'noexpand'` (convert to common denominator and add)
- `map` – boolean (default: `False`); if `True`, the result is an expression whose leading operator is the same as that of the expression `self` but whose subparts are the results of applying simplification rules to the corresponding subparts of the expressions.

ALIAS: `rational_simplify()` and `simplify_rational()` are the same

DETAILS: We call Maxima functions ratsimp, fullratsimp and xthru. If each part of the expression has to be simplified separately, we use Maxima function map.

EXAMPLES:

```
sage: f = sin(x/(x^2 + x))
sage: f
sin(x/(x^2 + x))
sage: f.simplify_rational()
sin(1/(x + 1))
```

```
>>> from sage.all import *
>>> f = sin(x/(x**Integer(2) + x))
>>> f
sin(x/(x^2 + x))
>>> f.simplify_rational()
sin(1/(x + 1))
```

```
sage: f = ((x - 1)^(3/2) - (x + 1)*sqrt(x - 1))/sqrt((x - 1)*(x + 1)); f
-((x + 1)*sqrt(x - 1) - (x - 1)^(3/2))/sqrt((x + 1)*(x - 1))
sage: f.simplify_rational()
-2*sqrt(x - 1)/sqrt(x^2 - 1)
```

```
>>> from sage.all import *
>>> f = ((x - Integer(1))**(Integer(3)/Integer(2)) - (x + Integer(1))*sqrt(x -
↪ Integer(1)))/sqrt((x - Integer(1))*(x + Integer(1))); f
-((x + 1)*sqrt(x - 1) - (x - 1)^(3/2))/sqrt((x + 1)*(x - 1))
>>> f.simplify_rational()
-2*sqrt(x - 1)/sqrt(x^2 - 1)
```

With `map=True` each term in a sum is simplified separately and thus the results are shorter for functions which are combination of rational and nonrational functions. In the following example, we use this option if we want not to combine logarithm and the rational function into one fraction:

```
sage: f = (x^2-1)/(x+1)-ln(x)/(x+2)
sage: f.simplify_rational()
(x^2 + x - log(x) - 2)/(x + 2)
sage: f.simplify_rational(map=True)
x - log(x)/(x + 2) - 1
```

```
>>> from sage.all import *
>>> f = (x**Integer(2)-Integer(1))/(x+Integer(1))-ln(x)/(x+Integer(2))
>>> f.simplify_rational()
(x^2 + x - log(x) - 2)/(x + 2)
>>> f.simplify_rational(map=True)
x - log(x)/(x + 2) - 1
```

Here is an example from the Maxima documentation of where `algorithm='simple'` produces an (possibly useful) intermediate step:

```
sage: y = var('y')
sage: g = (x^(y/2) + 1)^2*(x^(y/2) - 1)^2/(x^y - 1)
sage: g.simplify_rational(algorithm='simple')
(x^(2*y) - 2*x^y + 1)/(x^y - 1)
sage: g.simplify_rational()
x^y - 1
```

```
>>> from sage.all import *
>>> y = var('y')
>>> g = (x**(y/Integer(2)) + Integer(1))**Integer(2)*(x**(y/Integer(2)) -␣
↪Integer(1))**Integer(2)/(x**y - Integer(1))
>>> g.simplify_rational(algorithm='simple')
(x^(2*y) - 2*x^y + 1)/(x^y - 1)
>>> g.simplify_rational()
x^y - 1
```

With option `algorithm='noexpand'` we only convert to common denominators and add. No expansion of products is performed:

```
sage: f = 1/(x+1)+x/(x+2)^2
sage: f.simplify_rational()
(2*x^2 + 5*x + 4)/(x^3 + 5*x^2 + 8*x + 4)
sage: f.simplify_rational(algorithm='noexpand')
((x + 2)^2 + (x + 1)*x)/((x + 2)^2*(x + 1))
```

```
>>> from sage.all import *
>>> f = Integer(1)/(x+Integer(1))+x/(x+Integer(2))**Integer(2)
>>> f.simplify_rational()
(2*x^2 + 5*x + 4)/(x^3 + 5*x^2 + 8*x + 4)
>>> f.simplify_rational(algorithm='noexpand')
((x + 2)^2 + (x + 1)*x)/((x + 2)^2*(x + 1))
```

real(*hold=False*)

Return the real part of this symbolic expression.

EXAMPLES:

```
sage: x = var('x')
sage: x.real_part()
real_part(x)
sage: SR(2+3*I).real_part()
2
sage: SR(CDF(2,3)).real_part()
```

(continues on next page)

(continued from previous page)

```
2.0
sage: SR(CC(2,3)).real_part()
2.00000000000000

sage: f = log(x)
sage: f.real_part()
log(abs(x))
```

```
>>> from sage.all import *
>>> x = var('x')
>>> x.real_part()
real_part(x)
>>> SR(Integer(2)+Integer(3)*I).real_part()
2
>>> SR(CDF(Integer(2),Integer(3))).real_part()
2.0
>>> SR(CC(Integer(2),Integer(3))).real_part()
2.00000000000000

>>> f = log(x)
>>> f.real_part()
log(abs(x))
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(2).real_part()
2
sage: SR(2).real_part(hold=True)
real_part(2)
```

```
>>> from sage.all import *
>>> SR(Integer(2)).real_part()
2
>>> SR(Integer(2)).real_part(hold=True)
real_part(2)
```

This also works using functional notation:

```
sage: real_part(I,hold=True)
real_part(I)
sage: real_part(I)
0
```

```
>>> from sage.all import *
>>> real_part(I,hold=True)
real_part(I)
>>> real_part(I)
0
```

To then evaluate again, we use `unhold()`:

```
sage: a = SR(2).real_part(hold=True); a.unhold()
2
```

```
>>> from sage.all import *
>>> a = SR(Integer(2)).real_part(hold=True); a.unhold()
2
```

real_part(*hold=False*)

Return the real part of this symbolic expression.

EXAMPLES:

```
sage: x = var('x')
sage: x.real_part()
real_part(x)
sage: SR(2+3*I).real_part()
2
sage: SR(CDF(2,3)).real_part()
2.0
sage: SR(CC(2,3)).real_part()
2.00000000000000

sage: f = log(x)
sage: f.real_part()
log(abs(x))
```

```
>>> from sage.all import *
>>> x = var('x')
>>> x.real_part()
real_part(x)
>>> SR(Integer(2)+Integer(3)*I).real_part()
2
>>> SR(CDF(Integer(2),Integer(3))).real_part()
2.0
>>> SR(CC(Integer(2),Integer(3))).real_part()
2.00000000000000

>>> f = log(x)
>>> f.real_part()
log(abs(x))
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(2).real_part()
2
sage: SR(2).real_part(hold=True)
real_part(2)
```

```
>>> from sage.all import *
>>> SR(Integer(2)).real_part()
2
>>> SR(Integer(2)).real_part(hold=True)
real_part(2)
```

This also works using functional notation:

```
sage: real_part(I,hold=True)
real_part(I)
sage: real_part(I)
0
```

```
>>> from sage.all import *
>>> real_part(I,hold=True)
real_part(I)
>>> real_part(I)
0
```

To then evaluate again, we use `unhold()`:

```
sage: a = SR(2).real_part(hold=True); a.unhold()
2
```

```
>>> from sage.all import *
>>> a = SR(Integer(2)).real_part(hold=True); a.unhold()
2
```

rectform()

Convert this symbolic expression to rectangular form; that is, the form $a + bi$ where a and b are real numbers and i is the imaginary unit.

> **Note**
>
> The name "rectangular" comes from the fact that, in the complex plane, a and bi are perpendicular.

INPUT:

- `self` – the expression to convert

OUTPUT:

A new expression, equivalent to the original, but expressed in the form $a + bi$.

ALGORITHM:

We call Maxima's `rectform()` and return the result unmodified.

EXAMPLES:

The exponential form of $\sin(x)$:

```
sage: f = (e^(I*x) - e^(-I*x)) / (2*I)
sage: f.rectform()
sin(x)
```

```
>>> from sage.all import *
>>> f = (e**(I*x) - e**(-I*x)) / (Integer(2)*I)
>>> f.rectform()
sin(x)
```

And $\cos(x)$:

```
sage: f = (e^(I*x) + e^(-I*x)) / 2
sage: f.rectform()
cos(x)
```

```
>>> from sage.all import *
>>> f = (e**(I*x) + e**(-I*x)) / Integer(2)
>>> f.rectform()
cos(x)
```

In some cases, this will simplify the given expression. For example, here, $e^{ik\pi}$, $\sin(k\pi) = 0$ should cancel leaving only $\cos(k\pi)$ which can then be simplified:

```
sage: k = var('k')
sage: assume(k, 'integer')
sage: f = e^(I*pi*k)
sage: f.rectform()
(-1)^k
```

```
>>> from sage.all import *
>>> k = var('k')
>>> assume(k, 'integer')
>>> f = e**(I*pi*k)
>>> f.rectform()
(-1)^k
```

However, in general, the resulting expression may be more complicated than the original:

```
sage: f = e^(I*x)
sage: f.rectform()
cos(x) + I*sin(x)
```

```
>>> from sage.all import *
>>> f = e**(I*x)
>>> f.rectform()
cos(x) + I*sin(x)
```

reduce_trig(*var=None*)

Combine products and powers of trigonometric and hyperbolic sin's and cos's of x into those of multiples of x. It also tries to eliminate these functions when they occur in denominators.

INPUT:

- `self` – a symbolic expression
- `var` – (default: `None`) the variable which is used for these transformations. If not specified, all variables are used.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: y = var('y')
sage: f = sin(x)*cos(x)^3+sin(y)^2
sage: f.reduce_trig()
-1/2*cos(2*y) + 1/8*sin(4*x) + 1/4*sin(2*x) + 1/2
```

```
>>> from sage.all import *
>>> y = var('y')
>>> f = sin(x)*cos(x)**Integer(3)+sin(y)**Integer(2)
>>> f.reduce_trig()
-1/2*cos(2*y) + 1/8*sin(4*x) + 1/4*sin(2*x) + 1/2
```

To reduce only the expressions involving x we use optional parameter:

```
sage: f.reduce_trig(x)
sin(y)^2 + 1/8*sin(4*x) + 1/4*sin(2*x)
```

```
>>> from sage.all import *
>>> f.reduce_trig(x)
sin(y)^2 + 1/8*sin(4*x) + 1/4*sin(2*x)
```

ALIAS: `trig_reduce()` and `reduce_trig()` are the same

residue(*symbol*)

Calculate the residue of `self` with respect to `symbol`.

INPUT:

- `symbol` – a symbolic variable or symbolic equality such as `x == 5`. If an equality is given, the expansion is around the value on the right hand side of the equality, otherwise at `0`.

OUTPUT: the residue of `self`

Say, `symbol` is `x == a`, then this function calculates the residue of `self` at $x = a$, i.e., the coefficient of $1/(x - a)$ of the series expansion of `self` around a.

EXAMPLES:

```
sage: (1/x).residue(x == 0)
1
sage: (1/x).residue(x == oo)
-1
sage: (1/x^2).residue(x == 0)
0
sage: (1/sin(x)).residue(x == 0)
1
sage: var('q, n, z')
(q, n, z)
sage: (-z^(-n-1)/(1-z/q)^2).residue(z == q).simplify_full()
(n + 1)/q^n
sage: var('s')
s
sage: zeta(s).residue(s == 1)
1
```

```
>>> from sage.all import *
>>> (Integer(1)/x).residue(x == Integer(0))
1
>>> (Integer(1)/x).residue(x == oo)
-1
>>> (Integer(1)/x**Integer(2)).residue(x == Integer(0))
```

(continues on next page)

(continued from previous page)

```
0
>>> (Integer(1)/sin(x)).residue(x == Integer(0))
1
>>> var('q, n, z')
(q, n, z)
>>> (-z**(-n-Integer(1))/(Integer(1)-z/q)**Integer(2)).residue(z == q).
↪simplify_full()
(n + 1)/q^n
>>> var('s')
s
>>> zeta(s).residue(s == Integer(1))
1
```

We can also compute the residue at more general places, given that the pole is recognized:

```
sage: k = var('k', domain='integer')
sage: (gamma(1+x)/(1 - exp(-x))).residue(x==2*I*pi*k)
gamma(2*I*pi*k + 1)
sage: csc(x).residue(x==2*pi*k)
1
```

```
>>> from sage.all import *
>>> k = var('k', domain='integer')
>>> (gamma(Integer(1)+x)/(Integer(1) - exp(-x))).residue(x==Integer(2)*I*pi*k)
gamma(2*I*pi*k + 1)
>>> csc(x).residue(x==Integer(2)*pi*k)
1
```

resultant(*other*, *var*)

Compute the resultant of this polynomial expression and the first argument with respect to the variable given as the second argument.

EXAMPLES:

```
sage: _ = var('a b n k u x y')
sage: x.resultant(y, x)
y
sage: (x+y).resultant(x-y, x)
-2*y
sage: r = (x^4*y^2+x^2*y-y).resultant(x*y-y*a-x*b+a*b+u,x)
sage: r.coefficient(a^4)
b^4*y^2 - 4*b^3*y^3 + 6*b^2*y^4 - 4*b*y^5 + y^6
sage: x.resultant(sin(x), x)
Traceback (most recent call last):
...
RuntimeError: resultant(): arguments must be polynomials
```

```
>>> from sage.all import *
>>> _ = var('a b n k u x y')
>>> x.resultant(y, x)
y
>>> (x+y).resultant(x-y, x)
```

(continues on next page)

(continued from previous page)

```
-2*y
>>> r = (x**Integer(4)*y**Integer(2)+x**Integer(2)*y-y).resultant(x*y-y*a-
↪x*b+a*b+u,x)
>>> r.coefficient(a**Integer(4))
b^4*y^2 - 4*b^3*y^3 + 6*b^2*y^4 - 4*b*y^5 + y^6
>>> x.resultant(sin(x), x)
Traceback (most recent call last):
...
RuntimeError: resultant(): arguments must be polynomials
```

rhs()

If self is a relational expression, return the right hand side of the relation. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 <= x^2 - 2*x + 3
sage: eqn.right_hand_side()
x^2 - 2*x + 3
sage: eqn.rhs()
x^2 - 2*x + 3
sage: eqn.right()
x^2 - 2*x + 3
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eqn = (x-Integer(1))**Integer(2) <= x**Integer(2) - Integer(2)*x +␣
↪Integer(3)
>>> eqn.right_hand_side()
x^2 - 2*x + 3
>>> eqn.rhs()
x^2 - 2*x + 3
>>> eqn.right()
x^2 - 2*x + 3
```

right()

If self is a relational expression, return the right hand side of the relation. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 <= x^2 - 2*x + 3
sage: eqn.right_hand_side()
x^2 - 2*x + 3
sage: eqn.rhs()
x^2 - 2*x + 3
sage: eqn.right()
x^2 - 2*x + 3
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eqn = (x-Integer(1))**Integer(2) <= x**Integer(2) - Integer(2)*x +␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(3)
>>> eqn.right_hand_side()
x^2 - 2*x + 3
>>> eqn.rhs()
x^2 - 2*x + 3
>>> eqn.right()
x^2 - 2*x + 3
```

right_hand_side()

If `self` is a relational expression, return the right hand side of the relation. Otherwise, raise a `ValueError`.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 <= x^2 - 2*x + 3
sage: eqn.right_hand_side()
x^2 - 2*x + 3
sage: eqn.rhs()
x^2 - 2*x + 3
sage: eqn.right()
x^2 - 2*x + 3
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eqn = (x-Integer(1))**Integer(2) <= x**Integer(2) - Integer(2)*x +␣
↪Integer(3)
>>> eqn.right_hand_side()
x^2 - 2*x + 3
>>> eqn.rhs()
x^2 - 2*x + 3
>>> eqn.right()
x^2 - 2*x + 3
```

roots(*x=None*, *explicit_solutions=True*, *multiplicities=True*, *ring=None*)

Return roots of `self` that can be found exactly, possibly with multiplicities. Not all roots are guaranteed to be found.

Warning

This is *not* a numerical solver - use `find_root()` to solve for `self == 0` numerically on an interval.

INPUT:

- `x` – variable to view the function in terms of (use default variable if not given)
- `explicit_solutions` – boolean (default: `True`); require that roots be explicit rather than implicit
- `multiplicities` – boolean (default: `True`); when `True`, return multiplicities
- `ring` – a ring (default: `None`); if not `None`, convert `self` to a polynomial over `ring` and find roots over `ring`

OUTPUT:

A list of pairs `(root, multiplicity)` or list of roots.

If there are infinitely many roots, e.g., a function like $\sin(x)$, only one is returned.

EXAMPLES:

```
sage: var('x, a')
(x, a)
```

```
>>> from sage.all import *
>>> var('x, a')
(x, a)
```

A simple example:

```
sage: ((x^2-1)^2).roots()
[(-1, 2), (1, 2)]
sage: ((x^2-1)^2).roots(multiplicities=False)
[-1, 1]
```

```
>>> from sage.all import *
>>> ((x**Integer(2)-Integer(1))**Integer(2)).roots()
[(-1, 2), (1, 2)]
>>> ((x**Integer(2)-Integer(1))**Integer(2)).roots(multiplicities=False)
[-1, 1]
```

A complicated example:

```
sage: f = expand((x^2 - 1)^3*(x^2 + 1)*(x-a)); f
-a*x^8 + x^9 + 2*a*x^6 - 2*x^7 - 2*a*x^2 + 2*x^3 + a - x
```

```
>>> from sage.all import *
>>> f = expand((x**Integer(2) - Integer(1))**Integer(3)*(x**Integer(2) +
↪Integer(1))*(x-a)); f
-a*x^8 + x^9 + 2*a*x^6 - 2*x^7 - 2*a*x^2 + 2*x^3 + a - x
```

The default variable is a, since it is the first in alphabetical order:

```
sage: f.roots()
[(x, 1)]
```

```
>>> from sage.all import *
>>> f.roots()
[(x, 1)]
```

As a polynomial in a, x is indeed a root:

```
sage: f.poly(a)
x^9 - 2*x^7 + 2*x^3 - (x^8 - 2*x^6 + 2*x^2 - 1)*a - x
sage: f(a=x)
0
```

```
>>> from sage.all import *
>>> f.poly(a)
x^9 - 2*x^7 + 2*x^3 - (x^8 - 2*x^6 + 2*x^2 - 1)*a - x
```

(continues on next page)

(continued from previous page)

```
>>> f(a=x)
0
```

The roots in terms of x are what we expect:

```
sage: f.roots(x)
[(a, 1), (-I, 1), (I, 1), (1, 3), (-1, 3)]
```

```
>>> from sage.all import *
>>> f.roots(x)
[(a, 1), (-I, 1), (I, 1), (1, 3), (-1, 3)]
```

Only one root of $\sin(x) = 0$ is given:

```
sage: f = sin(x)
sage: f.roots(x)
[(0, 1)]
```

```
>>> from sage.all import *
>>> f = sin(x)
>>> f.roots(x)
[(0, 1)]
```

Note

It is possible to solve a greater variety of equations using *solve()* and the keyword `to_poly_solve`, but only at the price of possibly encountering approximate solutions. See documentation for *solve()* for more details.

We derive the roots of a general quadratic polynomial:

```
sage: var('a,b,c,x')
(a, b, c, x)
sage: (a*x^2 + b*x + c).roots(x)
[(-1/2*(b + sqrt(b^2 - 4*a*c))/a, 1), (-1/2*(b - sqrt(b^2 - 4*a*c))/a, 1)]
```

```
>>> from sage.all import *
>>> var('a,b,c,x')
(a, b, c, x)
>>> (a*x**Integer(2) + b*x + c).roots(x)
[(-1/2*(b + sqrt(b^2 - 4*a*c))/a, 1), (-1/2*(b - sqrt(b^2 - 4*a*c))/a, 1)]
```

By default, all the roots are required to be explicit rather than implicit. To get implicit roots, pass `explicit_solutions=False` to `.roots()`

```
sage: var('x')
x
sage: f = x^(1/9) + (2^(8/9) - 2^(1/9))*(x - 1) - x^(8/9)
sage: f.roots()
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
RuntimeError: no explicit roots found
sage: f.roots(explicit_solutions=False)
[((2^(8/9) + x^(8/9) - 2^(1/9) - x^(1/9))/(2^(8/9) - 2^(1/9)), 1)]
```

```
>>> from sage.all import *
>>> var('x')
x
>>> f = x**(Integer(1)/Integer(9)) + (Integer(2)**(Integer(8)/Integer(9)) -␣
→Integer(2)**(Integer(1)/Integer(9)))*(x - Integer(1)) - x**(Integer(8)/
→Integer(9))
>>> f.roots()
Traceback (most recent call last):
...
RuntimeError: no explicit roots found
>>> f.roots(explicit_solutions=False)
[((2^(8/9) + x^(8/9) - 2^(1/9) - x^(1/9))/(2^(8/9) - 2^(1/9)), 1)]
```

Another example, but involving a degree 5 poly whose roots do not get computed explicitly:

```
sage: f = x^5 + x^3 + 17*x + 1
sage: f.roots()
Traceback (most recent call last):
...
RuntimeError: no explicit roots found
sage: f.roots(explicit_solutions=False)
[(x^5 + x^3 + 17*x + 1, 1)]
sage: f.roots(explicit_solutions=False, multiplicities=False)
[x^5 + x^3 + 17*x + 1]
```

```
>>> from sage.all import *
>>> f = x**Integer(5) + x**Integer(3) + Integer(17)*x + Integer(1)
>>> f.roots()
Traceback (most recent call last):
...
RuntimeError: no explicit roots found
>>> f.roots(explicit_solutions=False)
[(x^5 + x^3 + 17*x + 1, 1)]
>>> f.roots(explicit_solutions=False, multiplicities=False)
[x^5 + x^3 + 17*x + 1]
```

Now let us find some roots over different rings:

```
sage: f.roots(ring=CC)
[(-0.0588115223184..., 1),
 (-1.331099917875... - 1.52241655183732*I, 1),
 (-1.331099917875... + 1.52241655183732*I, 1),
 (1.36050567903502 - 1.51880872209965*I, 1),
 (1.36050567903502 + 1.51880872209965*I, 1)]
sage: (2.5*f).roots(ring=RR)
[(-0.058811522318449..., 1)]
sage: f.roots(ring=CC, multiplicities=False)
[-0.05881152231844...,
```

(continues on next page)

(continued from previous page)

```
 -1.331099917875... - 1.52241655183732*I,
 -1.331099917875... + 1.52241655183732*I,
 1.36050567903502 - 1.51880872209965*I,
 1.36050567903502 + 1.51880872209965*I]
sage: f.roots(ring=QQ)
[]
sage: f.roots(ring=QQbar, multiplicities=False)
[-0.05881152231844944?,
 -1.331099917875796? - 1.522416551837318?*I,
 -1.331099917875796? + 1.522416551837318?*I,
 1.360505679035020? - 1.518808722099650?*I,
 1.360505679035020? + 1.518808722099650?*I]
```

```
>>> from sage.all import *
>>> f.roots(ring=CC)
[(-0.0588115223184..., 1),
 (-1.331099917875... - 1.52241655183732*I, 1),
 (-1.331099917875... + 1.52241655183732*I, 1),
 (1.36050567903502 - 1.51880872209965*I, 1),
 (1.36050567903502 + 1.51880872209965*I, 1)]
>>> (RealNumber('2.5')*f).roots(ring=RR)
[(-0.058811522318449..., 1)]
>>> f.roots(ring=CC, multiplicities=False)
[-0.05881152231844...,
 -1.331099917875... - 1.52241655183732*I,
 -1.331099917875... + 1.52241655183732*I,
 1.36050567903502 - 1.51880872209965*I,
 1.36050567903502 + 1.51880872209965*I]
>>> f.roots(ring=QQ)
[]
>>> f.roots(ring=QQbar, multiplicities=False)
[-0.05881152231844944?,
 -1.331099917875796? - 1.522416551837318?*I,
 -1.331099917875796? + 1.522416551837318?*I,
 1.360505679035020? - 1.518808722099650?*I,
 1.360505679035020? + 1.518808722099650?*I]
```

Root finding over finite fields:

```
sage: f.roots(ring=GF(7^2, 'a'))                                        #␣
↪needs sage.rings.finite_rings
[(3, 1), (4*a + 6, 2), (3*a + 3, 2)]
```

```
>>> from sage.all import *
>>> f.roots(ring=GF(Integer(7)**Integer(2), 'a'))                        ␣
↪                # needs sage.rings.finite_rings
[(3, 1), (4*a + 6, 2), (3*a + 3, 2)]
```

round()

Round this expression to the nearest integer.

EXAMPLES:

```
sage: u = sqrt(43203735824841025516773866131535024)
sage: u.round()
207855083711803945
sage: t = sqrt(Integer('1'*1000)).round(); print(str(t)[-10:])
3333333333
sage: (-sqrt(110)).round()
-10
sage: (-sqrt(115)).round()
-11
sage: (sqrt(-3)).round()
Traceback (most recent call last):
...
ValueError: could not convert sqrt(-3) to a real number
```

```
>>> from sage.all import *
>>> u = sqrt(Integer(43203735824841025516773866131535024))
>>> u.round()
207855083711803945
>>> t = sqrt(Integer('1'*Integer(1000))).round(); print(str(t)[-Integer(10):])
3333333333
>>> (-sqrt(Integer(110))).round()
-10
>>> (-sqrt(Integer(115))).round()
-11
>>> (sqrt(-Integer(3))).round()
Traceback (most recent call last):
...
ValueError: could not convert sqrt(-3) to a real number
```

series(*symbol*, *order=None*)

Return the power series expansion of `self` in terms of the given variable to the given order.

INPUT:

- `symbol` – a symbolic variable or symbolic equality such as `x == 5`; if an equality is given, the expansion is around the value on the right hand side of the equality
- `order` – integer; if nothing given, it is set to the global default (`20`), which can be changed using `set_series_precision()`

OUTPUT: a power series

To truncate the power series and obtain a normal expression, use the `truncate()` command.

EXAMPLES:

We expand a polynomial in x about 0, about 1, and also truncate it back to a polynomial:

```
sage: var('x,y')
(x, y)
sage: f = (x^3 - sin(y)*x^2 - 5*x + 3); f
x^3 - x^2*sin(y) - 5*x + 3
sage: g = f.series(x, 4); g
3 + (-5)*x + (-sin(y))*x^2 + 1*x^3 + Order(x^4)
sage: g.truncate()
x^3 - x^2*sin(y) - 5*x + 3
```

(continues on next page)

(continued from previous page)

```
sage: g = f.series(x==1, 4); g
(-sin(y) - 1) + (-2*sin(y) - 2)*(x - 1) + (-sin(y) + 3)*(x - 1)^2
+ 1*(x - 1)^3 + Order((x - 1)^4)
sage: h = g.truncate(); h
(x - 1)^3 - (x - 1)^2*(sin(y) - 3) - 2*(x - 1)*(sin(y) + 1) - sin(y) - 1
sage: h.expand()
x^3 - x^2*sin(y) - 5*x + 3
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> f = (x**Integer(3) - sin(y)*x**Integer(2) - Integer(5)*x + Integer(3)); f
x^3 - x^2*sin(y) - 5*x + 3
>>> g = f.series(x, Integer(4)); g
3 + (-5)*x + (-sin(y))*x^2 + 1*x^3 + Order(x^4)
>>> g.truncate()
x^3 - x^2*sin(y) - 5*x + 3
>>> g = f.series(x==Integer(1), Integer(4)); g
(-sin(y) - 1) + (-2*sin(y) - 2)*(x - 1) + (-sin(y) + 3)*(x - 1)^2
+ 1*(x - 1)^3 + Order((x - 1)^4)
>>> h = g.truncate(); h
(x - 1)^3 - (x - 1)^2*(sin(y) - 3) - 2*(x - 1)*(sin(y) + 1) - sin(y) - 1
>>> h.expand()
x^3 - x^2*sin(y) - 5*x + 3
```

We computer another series expansion of an analytic function:

```
sage: f = sin(x)/x^2
sage: f.series(x,7)
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
sage: f.series(x)
1*x^(-1) + (-1/6)*x + ... + Order(x^20)
sage: f.series(x==1,3)
(sin(1)) + (cos(1) - 2*sin(1))*(x - 1) + (-2*cos(1) + 5/2*sin(1))*(x - 1)^2
+ Order((x - 1)^3)
sage: f.series(x==1,3).truncate().expand()
-2*x^2*cos(1) + 5/2*x^2*sin(1) + 5*x*cos(1) - 7*x*sin(1) - 3*cos(1) + 11/
↪2*sin(1)
```

```
>>> from sage.all import *
>>> f = sin(x)/x**Integer(2)
>>> f.series(x,Integer(7))
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
>>> f.series(x)
1*x^(-1) + (-1/6)*x + ... + Order(x^20)
>>> f.series(x==Integer(1),Integer(3))
(sin(1)) + (cos(1) - 2*sin(1))*(x - 1) + (-2*cos(1) + 5/2*sin(1))*(x - 1)^2
+ Order((x - 1)^3)
>>> f.series(x==Integer(1),Integer(3)).truncate().expand()
-2*x^2*cos(1) + 5/2*x^2*sin(1) + 5*x*cos(1) - 7*x*sin(1) - 3*cos(1) + 11/
↪2*sin(1)
```

Expressions formed by combining series can be expanded by applying series again:

```
sage: (1/(1-x)).series(x, 3)+(1/(1+x)).series(x,3)
(1 + 1*x + 1*x^2 + Order(x^3)) + (1 + (-1)*x + 1*x^2 + Order(x^3))
sage: _.series(x,3)
2 + 2*x^2 + Order(x^3)
sage: (1/(1-x)).series(x, 3)*(1/(1+x)).series(x,3)
(1 + 1*x + 1*x^2 + Order(x^3))*(1 + (-1)*x + 1*x^2 + Order(x^3))
sage: _.series(x,3)
1 + 1*x^2 + Order(x^3)
```

```
>>> from sage.all import *
>>> (Integer(1)/(Integer(1)-x)).series(x, Integer(3))+(Integer(1)/
↪(Integer(1)+x)).series(x,Integer(3))
(1 + 1*x + 1*x^2 + Order(x^3)) + (1 + (-1)*x + 1*x^2 + Order(x^3))
>>> _.series(x,Integer(3))
2 + 2*x^2 + Order(x^3)
>>> (Integer(1)/(Integer(1)-x)).series(x, Integer(3))*(Integer(1)/
↪(Integer(1)+x)).series(x,Integer(3))
(1 + 1*x + 1*x^2 + Order(x^3))*(1 + (-1)*x + 1*x^2 + Order(x^3))
>>> _.series(x,Integer(3))
1 + 1*x^2 + Order(x^3)
```

Following the GiNaC tutorial, we use John Machin's amazing formula $\pi = 16\tan^{-1}(1/5) - 4\tan^{-1}(1/239)$ to compute digits of π. We expand the arc tangent around 0 and insert the fractions 1/5 and 1/239.

```
sage: x = var('x')
sage: f = atan(x).series(x, 10); f
1*x + (-1/3)*x^3 + 1/5*x^5 + (-1/7)*x^7 + 1/9*x^9 + Order(x^10)
sage: float(16*f.subs(x==1/5) - 4*f.subs(x==1/239))
3.1415926824043994
```

```
>>> from sage.all import *
>>> x = var('x')
>>> f = atan(x).series(x, Integer(10)); f
1*x + (-1/3)*x^3 + 1/5*x^5 + (-1/7)*x^7 + 1/9*x^9 + Order(x^10)
>>> float(Integer(16)*f.subs(x==Integer(1)/Integer(5)) - Integer(4)*f.
↪subs(x==Integer(1)/Integer(239)))
3.1415926824043994
```

show()

Pretty-print this symbolic expression.

This typesets it nicely and prints it immediately.

OUTPUT:

This method does not return anything. Like `print`, output is sent directly to the screen.

Note that the output depends on the display preferences. For details, see `pretty_print()`.

EXAMPLES:

```
sage: (x^2 + 1).show()
x^2 + 1
```

```
>>> from sage.all import *
>>> (x**Integer(2) + Integer(1)).show()
x^2 + 1
```

EXAMPLES:

```
sage: %display ascii_art  # not tested
sage: (x^2 + 1).show()
 2
x  + 1
```

```
>>> from sage.all import *
>>> %display ascii_art  # not tested
>>> (x**Integer(2) + Integer(1)).show()
 2
x  + 1
```

simplify(*algorithm='maxima'*, ***kwds*)

Return a simplified version of this symbolic expression.

INPUT:

- `algorithm` – one of :
  - `maxima` : (default) sends the expression to `maxima` and converts it back to Sage
  - `sympy` : converts the expression to `sympy`, simplifies it (passing any optional keyword(s)), and converts the result to Sage
  - `giac` : converts the expression to `giac`, simplifies it, and converts the result to Sage
  - `fricas` : converts the expression to `fricas`, simplifies it, and converts the result to Sage

See also

simplify_full(), *simplify_trig()*, *simplify_rational()*, *simplify_rectform()* *simplify_factorial()*, *simplify_log()*, *simplify_real()*, *simplify_hypergeometric()*, *canonicalize_radical()*

EXAMPLES:

```
sage: a = var('a'); f = x*sin(2)/(x^a); f
x*sin(2)/x^a
sage: f.simplify()
x^(-a + 1)*sin(2)
```

```
>>> from sage.all import *
>>> a = var('a'); f = x*sin(Integer(2))/(x**a); f
x*sin(2)/x^a
>>> f.simplify()
x^(-a + 1)*sin(2)
```

Some simplifications are quite algorithm-specific:

```
sage: x, t = var("x, t")
sage: ex = cos(t).exponentialize()
sage: ex = ex.subs((sin(t).exponentialize()==x).solve(t)[0])
sage: ex
1/2*I*x + 1/2*I*sqrt(x^2 - 1) + 1/2/(I*x + I*sqrt(x^2 - 1))
sage: ex.simplify()
1/2*I*x + 1/2*I*sqrt(x^2 - 1) + 1/(2*I*x + 2*I*sqrt(x^2 - 1))
sage: ex.simplify(algorithm='sympy')
I*(x^2 + sqrt(x^2 - 1)*x - 1)/(x + sqrt(x^2 - 1))
sage: ex.simplify(algorithm='giac')  # needs giac
I*sqrt(x^2 - 1)
sage: ex.simplify(algorithm='fricas')  # optional - fricas
(I*x^2 + I*sqrt(x^2 - 1)*x - I)/(x + sqrt(x^2 - 1))
```

```
>>> from sage.all import *
>>> x, t = var("x, t")
>>> ex = cos(t).exponentialize()
>>> ex = ex.subs((sin(t).exponentialize()==x).solve(t)[Integer(0)])
>>> ex
1/2*I*x + 1/2*I*sqrt(x^2 - 1) + 1/2/(I*x + I*sqrt(x^2 - 1))
>>> ex.simplify()
1/2*I*x + 1/2*I*sqrt(x^2 - 1) + 1/(2*I*x + 2*I*sqrt(x^2 - 1))
>>> ex.simplify(algorithm='sympy')
I*(x^2 + sqrt(x^2 - 1)*x - 1)/(x + sqrt(x^2 - 1))
>>> ex.simplify(algorithm='giac')  # needs giac
I*sqrt(x^2 - 1)
>>> ex.simplify(algorithm='fricas')  # optional - fricas
(I*x^2 + I*sqrt(x^2 - 1)*x - I)/(x + sqrt(x^2 - 1))
```

simplify_factorial()

Simplify by combining expressions with factorials, and by expanding binomials into factorials.

ALIAS: factorial_simplify and simplify_factorial are the same

EXAMPLES:

Some examples are relatively clear:

```
sage: var('n,k')
(n, k)
sage: f = factorial(n+1)/factorial(n); f
factorial(n + 1)/factorial(n)
sage: f.simplify_factorial()
n + 1
```

```
>>> from sage.all import *
>>> var('n,k')
(n, k)
>>> f = factorial(n+Integer(1))/factorial(n); f
factorial(n + 1)/factorial(n)
>>> f.simplify_factorial()
n + 1
```

```
sage: f = factorial(n)*(n+1); f
(n + 1)*factorial(n)
sage: simplify(f)
(n + 1)*factorial(n)
sage: f.simplify_factorial()
factorial(n + 1)
```

```
>>> from sage.all import *
>>> f = factorial(n)*(n+Integer(1)); f
(n + 1)*factorial(n)
>>> simplify(f)
(n + 1)*factorial(n)
>>> f.simplify_factorial()
factorial(n + 1)
```

```
sage: f = binomial(n, k)*factorial(k)*factorial(n-k); f
binomial(n, k)*factorial(k)*factorial(-k + n)
sage: f.simplify_factorial()
factorial(n)
```

```
>>> from sage.all import *
>>> f = binomial(n, k)*factorial(k)*factorial(n-k); f
binomial(n, k)*factorial(k)*factorial(-k + n)
>>> f.simplify_factorial()
factorial(n)
```

A more complicated example, which needs further processing:

```
sage: f = factorial(x)/factorial(x-2)/2 + factorial(x+1)/factorial(x)/2; f
1/2*factorial(x + 1)/factorial(x) + 1/2*factorial(x)/factorial(x - 2)
sage: g = f.simplify_factorial(); g
1/2*(x - 1)*x + 1/2*x + 1/2
sage: g.simplify_rational()
1/2*x^2 + 1/2
```

```
>>> from sage.all import *
>>> f = factorial(x)/factorial(x-Integer(2))/Integer(2) +␣
↪factorial(x+Integer(1))/factorial(x)/Integer(2); f
1/2*factorial(x + 1)/factorial(x) + 1/2*factorial(x)/factorial(x - 2)
>>> g = f.simplify_factorial(); g
1/2*(x - 1)*x + 1/2*x + 1/2
>>> g.simplify_rational()
1/2*x^2 + 1/2
```

simplify_full()

Apply *simplify_factorial()*, *simplify_rectform()*, *simplify_trig()*, *simplify_rational()*, and then *expand_sum()* to `self` (in that order).

ALIAS: `simplify_full` and `full_simplify` are the same.

EXAMPLES:

```
sage: f = sin(x)^2 + cos(x)^2
sage: f.simplify_full()
1
```

```
>>> from sage.all import *
>>> f = sin(x)**Integer(2) + cos(x)**Integer(2)
>>> f.simplify_full()
1
```

```
sage: f = sin(x/(x^2 + x))
sage: f.simplify_full()
sin(1/(x + 1))
```

```
>>> from sage.all import *
>>> f = sin(x/(x**Integer(2) + x))
>>> f.simplify_full()
sin(1/(x + 1))
```

```
sage: var('n,k')
(n, k)
sage: f = binomial(n,k)*factorial(k)*factorial(n-k)
sage: f.simplify_full()
factorial(n)
```

```
>>> from sage.all import *
>>> var('n,k')
(n, k)
>>> f = binomial(n,k)*factorial(k)*factorial(n-k)
>>> f.simplify_full()
factorial(n)
```

simplify_hypergeometric(*algorithm='maxima'*)

Simplify an expression containing hypergeometric or confluent hypergeometric functions.

INPUT:

- `algorithm` – (default: `'maxima'`) the algorithm to use for for simplification. Implemented are `'maxima'`, which uses Maxima's `hgfred` function, and `'sage'`, which uses an algorithm implemented in the hypergeometric module

ALIAS: *hypergeometric_simplify()* and *simplify_hypergeometric()* are the same

EXAMPLES:

```
sage: hypergeometric((5, 4), (4, 1, 2, 3),
....:                x).simplify_hypergeometric()
1/144*x^2*hypergeometric((), (3, 4), x) +...
1/3*x*hypergeometric((), (2, 3), x) + hypergeometric((), (1, 2), x)
sage: (2*hypergeometric((), (), x)).simplify_hypergeometric()
2*e^x
sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1)  # not tested,␣
↪unstable
....:  .simplify_hypergeometric())
```

(continues on next page)

(continued from previous page)

```
laguerre(-laguerre(-e^x, x), x)
sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1)  # not tested,␣
↪unstable
....:  .simplify_hypergeometric(algorithm='sage'))
hypergeometric((hypergeometric((e^x,), (1,), x),), (1,), x)
sage: hypergeometric_M(1, 3, x).simplify_hypergeometric()
-2*((x + 1)*e^(-x) - 1)*e^x/x^2
sage: (2 * hypergeometric_U(1, 3, x)).simplify_hypergeometric()
2*(x + 1)/x^2
```

```
>>> from sage.all import *
>>> hypergeometric((Integer(5), Integer(4)), (Integer(4), Integer(1),␣
↪Integer(2), Integer(3)),
...                x).simplify_hypergeometric()
1/144*x^2*hypergeometric((), (3, 4), x) +...
1/3*x*hypergeometric((), (2, 3), x) + hypergeometric((), (1, 2), x)
>>> (Integer(2)*hypergeometric((), (), x)).simplify_hypergeometric()
2*e^x
>>> (nest(lambda y: hypergeometric([y], [Integer(1)], x), Integer(3),␣
↪Integer(1))  # not tested, unstable
...  .simplify_hypergeometric())
laguerre(-laguerre(-e^x, x), x)
>>> (nest(lambda y: hypergeometric([y], [Integer(1)], x), Integer(3),␣
↪Integer(1))  # not tested, unstable
...  .simplify_hypergeometric(algorithm='sage'))
hypergeometric((hypergeometric((e^x,), (1,), x),), (1,), x)
>>> hypergeometric_M(Integer(1), Integer(3), x).simplify_hypergeometric()
-2*((x + 1)*e^(-x) - 1)*e^x/x^2
>>> (Integer(2) * hypergeometric_U(Integer(1), Integer(3), x)).simplify_
↪hypergeometric()
2*(x + 1)/x^2
```

simplify_log(*algorithm=None*)

Simplify a (real) symbolic expression that contains logarithms.

The given expression is scanned recursively, transforming subexpressions of the form $a \log(b) + c \log(d)$ into $\log(b^a d^c)$ before simplifying within the `log()`.

The user can specify conditions that a and c must satisfy before this transformation will be performed using the optional parameter `algorithm`.

Warning

This is only safe to call if every variable in the given expression is assumed to be real. The simplification it performs is in general not valid over the complex numbers. For example:

```
sage: x,y = SR.var('x,y')
sage: f = log(x*y) - (log(x) + log(y))
sage: f(x=-1, y=i)
-2*I*pi
sage: f.simplify_log()
0
```

```
>>> from sage.all import *
>>> x,y = SR.var('x,y')
>>> f = log(x*y) - (log(x) + log(y))
>>> f(x=-Integer(1), y=i)
-2*I*pi
>>> f.simplify_log()
0
```

INPUT:

- `self` – expression to be simplified
- `algorithm` – (default: `None`) governs the condition on a and c which must be satisfied to contract expression $a\log(b) + c\log(d)$. Values are
  - `None` (use Maxima default, integers),
  - `'one'` (1 and -1),
  - `'ratios'` (rational numbers),
  - `'constants'` (constants),
  - `'all'` (all expressions).

ALGORITHM:

This uses the Maxima `logcontract()` command.

ALIAS:

`log_simplify()` and *`simplify_log()`* are the same.

EXAMPLES:

```
sage: x,y,t = var('x y t')
```

```
>>> from sage.all import *
>>> x,y,t = var('x y t')
```

Only two first terms are contracted in the following example; the logarithm with coefficient $\frac{1}{2}$ is not contracted:

```
sage: f = log(x)+2*log(y)+1/2*log(t)
sage: f.simplify_log()
log(x*y^2) + 1/2*log(t)
```

```
>>> from sage.all import *
>>> f = log(x)+Integer(2)*log(y)+Integer(1)/Integer(2)*log(t)
>>> f.simplify_log()
log(x*y^2) + 1/2*log(t)
```

To contract all terms in the previous example, we use the `'ratios'` `algorithm`:

```
sage: f.simplify_log(algorithm='ratios')
log(sqrt(t)*x*y^2)
```

```
>>> from sage.all import *
>>> f.simplify_log(algorithm='ratios')
log(sqrt(t)*x*y^2)
```

To contract terms with no coefficient (more precisely, with coefficients 1 and -1), we use the `'one'` algorithm:

```
sage: f = log(x)+2*log(y)-log(t)
sage: f.simplify_log('one')
2*log(y) + log(x/t)
```

```
>>> from sage.all import *
>>> f = log(x)+Integer(2)*log(y)-log(t)
>>> f.simplify_log('one')
2*log(y) + log(x/t)
```

```
sage: f = log(x)+log(y)-1/3*log((x+1))
sage: f.simplify_log()
log(x*y) - 1/3*log(x + 1)

sage: f.simplify_log('ratios')
log(x*y/(x + 1)^(1/3))
```

```
>>> from sage.all import *
>>> f = log(x)+log(y)-Integer(1)/Integer(3)*log((x+Integer(1)))
>>> f.simplify_log()
log(x*y) - 1/3*log(x + 1)

>>> f.simplify_log('ratios')
log(x*y/(x + 1)^(1/3))
```

π is an irrational number; to contract logarithms in the following example we have to set `algorithm` to `'constants'` or `'all'`:

```
sage: f = log(x)+log(y)-pi*log((x+1))
sage: f.simplify_log('constants')
log(x*y/(x + 1)^pi)
```

```
>>> from sage.all import *
>>> f = log(x)+log(y)-pi*log((x+Integer(1)))
>>> f.simplify_log('constants')
log(x*y/(x + 1)^pi)
```

`x*log(9)` is contracted only if `algorithm` is `'all'`:

```
sage: (x*log(9)).simplify_log()
2*x*log(3)
sage: (x*log(9)).simplify_log('all')
log(3^(2*x))
```

```
>>> from sage.all import *
>>> (x*log(Integer(9))).simplify_log()
```

(continues on next page)

(continued from previous page)

```
2*x*log(3)
>>> (x*log(Integer(9))).simplify_log('all')
log(3^(2*x))
```

AUTHORS:

- Robert Marik (11-2009)

simplify_rational(*algorithm='full'*, *map=False*)

Simplify rational expressions.

INPUT:

- `self` – symbolic expression
- `algorithm` – (default: `'full'`) string which switches the algorithm for simplifications. Possible values are
  - `'simple'` (simplify rational functions into quotient of two polynomials)
  - `'full'` (apply repeatedly, if necessary)
  - `'noexpand'` (convert to common denominator and add)
- `map` – boolean (default: `False`); if `True`, the result is an expression whose leading operator is the same as that of the expression `self` but whose subparts are the results of applying simplification rules to the corresponding subparts of the expressions.

ALIAS: *rational_simplify()* and *simplify_rational()* are the same

DETAILS: We call Maxima functions ratsimp, fullratsimp and xthru. If each part of the expression has to be simplified separately, we use Maxima function map.

EXAMPLES:

```
sage: f = sin(x/(x^2 + x))
sage: f
sin(x/(x^2 + x))
sage: f.simplify_rational()
sin(1/(x + 1))
```

```
>>> from sage.all import *
>>> f = sin(x/(x**Integer(2) + x))
>>> f
sin(x/(x^2 + x))
>>> f.simplify_rational()
sin(1/(x + 1))
```

```
sage: f = ((x - 1)^(3/2) - (x + 1)*sqrt(x - 1))/sqrt((x - 1)*(x + 1)); f
-((x + 1)*sqrt(x - 1) - (x - 1)^(3/2))/sqrt((x + 1)*(x - 1))
sage: f.simplify_rational()
-2*sqrt(x - 1)/sqrt(x^2 - 1)
```

```
>>> from sage.all import *
>>> f = ((x - Integer(1))**(Integer(3)/Integer(2)) - (x + Integer(1))*sqrt(x -
↪ Integer(1)))/sqrt((x - Integer(1))*(x + Integer(1))); f
-((x + 1)*sqrt(x - 1) - (x - 1)^(3/2))/sqrt((x + 1)*(x - 1))
```

(continues on next page)

(continued from previous page)

```
>>> f.simplify_rational()
-2*sqrt(x - 1)/sqrt(x^2 - 1)
```

With `map=True` each term in a sum is simplified separately and thus the results are shorter for functions which are combination of rational and nonrational functions. In the following example, we use this option if we want not to combine logarithm and the rational function into one fraction:

```
sage: f = (x^2-1)/(x+1)-ln(x)/(x+2)
sage: f.simplify_rational()
(x^2 + x - log(x) - 2)/(x + 2)
sage: f.simplify_rational(map=True)
x - log(x)/(x + 2) - 1
```

```
>>> from sage.all import *
>>> f = (x**Integer(2)-Integer(1))/(x+Integer(1))-ln(x)/(x+Integer(2))
>>> f.simplify_rational()
(x^2 + x - log(x) - 2)/(x + 2)
>>> f.simplify_rational(map=True)
x - log(x)/(x + 2) - 1
```

Here is an example from the Maxima documentation of where `algorithm='simple'` produces an (possibly useful) intermediate step:

```
sage: y = var('y')
sage: g = (x^(y/2) + 1)^2*(x^(y/2) - 1)^2/(x^y - 1)
sage: g.simplify_rational(algorithm='simple')
(x^(2*y) - 2*x^y + 1)/(x^y - 1)
sage: g.simplify_rational()
x^y - 1
```

```
>>> from sage.all import *
>>> y = var('y')
>>> g = (x**(y/Integer(2)) + Integer(1))**Integer(2)*(x**(y/Integer(2)) -␣
↪Integer(1))**Integer(2)/(x**y - Integer(1))
>>> g.simplify_rational(algorithm='simple')
(x^(2*y) - 2*x^y + 1)/(x^y - 1)
>>> g.simplify_rational()
x^y - 1
```

With option `algorithm='noexpand'` we only convert to common denominators and add. No expansion of products is performed:

```
sage: f = 1/(x+1)+x/(x+2)^2
sage: f.simplify_rational()
(2*x^2 + 5*x + 4)/(x^3 + 5*x^2 + 8*x + 4)
sage: f.simplify_rational(algorithm='noexpand')
((x + 2)^2 + (x + 1)*x)/((x + 2)^2*(x + 1))
```

```
>>> from sage.all import *
>>> f = Integer(1)/(x+Integer(1))+x/(x+Integer(2))**Integer(2)
>>> f.simplify_rational()
(2*x^2 + 5*x + 4)/(x^3 + 5*x^2 + 8*x + 4)
```

(continues on next page)

(continued from previous page)

```
>>> f.simplify_rational(algorithm='noexpand')
((x + 2)^2 + (x + 1)*x)/((x + 2)^2*(x + 1))
```

simplify_real()

Simplify the given expression over the real numbers. This allows the simplification of $\sqrt{x^2}$ into $|x|$ and the contraction of $\log(x) + \log(y)$ into $\log(xy)$.

INPUT:

- `self` – the expression to convert

OUTPUT:

A new expression, equivalent to the original one under the assumption that the variables involved are real.

EXAMPLES:

```
sage: f = sqrt(x^2)
sage: f.simplify_real()
abs(x)
```

```
>>> from sage.all import *
>>> f = sqrt(x**Integer(2))
>>> f.simplify_real()
abs(x)
```

```
sage: y = SR.var('y')
sage: f = log(x) + 2*log(y)
sage: f.simplify_real()
log(x*y^2)
```

```
>>> from sage.all import *
>>> y = SR.var('y')
>>> f = log(x) + Integer(2)*log(y)
>>> f.simplify_real()
log(x*y^2)
```

simplify_rectform(*complexity_measure='string_length'*)

Attempt to simplify this expression by expressing it in the form $a + bi$ where both a and b are real. This transformation is generally not a simplification, so we use the given `complexity_measure` to discard non-simplifications.

INPUT:

- `self` – the expression to simplify
- `complexity_measure` – (default: `sage.symbolic.complexity_measures.string_length`) a function taking a symbolic expression as an argument and returning a measure of that expressions complexity. If `None` is supplied, the simplification will be performed regardless of the result.

OUTPUT:

If the transformation produces a simpler expression (according to `complexity_measure`) then that simpler expression is returned. Otherwise, the original expression is returned.

ALGORITHM:

We first call `rectform()` on the given expression. Then, the supplied complexity measure is used to determine whether or not the result is simpler than the original expression.

EXAMPLES:

The exponential form of $\tan(x)$:

```
sage: f = ( e^(I*x) - e^(-I*x) ) / ( I*e^(I*x) + I*e^(-I*x) )
sage: f.simplify_rectform()
sin(x)/cos(x)
```

```
>>> from sage.all import *
>>> f = ( e**(I*x) - e**(-I*x) ) / ( I*e**(I*x) + I*e**(-I*x) )
>>> f.simplify_rectform()
sin(x)/cos(x)
```

This should not be expanded with Euler's formula since the resulting expression is longer when considered as a string, and the default `complexity_measure` uses string length to determine which expression is simpler:

```
sage: f = e^(I*x)
sage: f.simplify_rectform()
e^(I*x)
```

```
>>> from sage.all import *
>>> f = e**(I*x)
>>> f.simplify_rectform()
e^(I*x)
```

However, if we pass `None` as our complexity measure, it is:

```
sage: f = e^(I*x)
sage: f.simplify_rectform(complexity_measure = None)
cos(x) + I*sin(x)
```

```
>>> from sage.all import *
>>> f = e**(I*x)
>>> f.simplify_rectform(complexity_measure = None)
cos(x) + I*sin(x)
```

simplify_trig(*expand=True*)

Optionally expand and then employ identities such as $\sin(x)^2 + \cos(x)^2 = 1$, $\cosh(x)^2 - \sinh(x)^2 = 1$, $\sin(x)\csc(x) = 1$, or $\tanh(x) = \sinh(x)/\cosh(x)$ to simplify expressions containing tan, sec, etc., to sin, cos, sinh, cosh.

INPUT:

- `self` – symbolic expression
- `expand` – boolean (default: `True`); if `True`, expands trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in `self` first. For best results, `self` should be expanded. See also `expand_trig()` to get more controls on this expansion.

ALIAS: `trig_simplify()` and `simplify_trig()` are the same

EXAMPLES:

```
sage: f = sin(x)^2 + cos(x)^2; f
cos(x)^2 + sin(x)^2
sage: f.simplify()
cos(x)^2 + sin(x)^2
sage: f.simplify_trig()
1
sage: h = sin(x)*csc(x)
sage: h.simplify_trig()
1
sage: k = tanh(x)*cosh(2*x)
sage: k.simplify_trig()
(2*sinh(x)^3 + sinh(x))/cosh(x)
```

```
>>> from sage.all import *
>>> f = sin(x)**Integer(2) + cos(x)**Integer(2); f
cos(x)^2 + sin(x)^2
>>> f.simplify()
cos(x)^2 + sin(x)^2
>>> f.simplify_trig()
1
>>> h = sin(x)*csc(x)
>>> h.simplify_trig()
1
>>> k = tanh(x)*cosh(Integer(2)*x)
>>> k.simplify_trig()
(2*sinh(x)^3 + sinh(x))/cosh(x)
```

In some cases we do not want to expand:

```
sage: f = tan(3*x)
sage: f.simplify_trig()
-(4*cos(x)^2 - 1)*sin(x)/(4*cos(x)*sin(x)^2 - cos(x))
sage: f.simplify_trig(False)
sin(3*x)/cos(3*x)
```

```
>>> from sage.all import *
>>> f = tan(Integer(3)*x)
>>> f.simplify_trig()
-(4*cos(x)^2 - 1)*sin(x)/(4*cos(x)*sin(x)^2 - cos(x))
>>> f.simplify_trig(False)
sin(3*x)/cos(3*x)
```

sin(*hold=False*)

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: sin(x^2 + y^2)
sin(x^2 + y^2)
sage: sin(sage.symbolic.constants.pi)
0
sage: sin(SR(1))
```

(continues on next page)

(continued from previous page)

```
sin(1)
sage: sin(SR(RealField(150)(1)))
0.84147098480789650665250232163029899962256306
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> sin(x**Integer(2) + y**Integer(2))
sin(x^2 + y^2)
>>> sin(sage.symbolic.constants.pi)
0
>>> sin(SR(Integer(1)))
sin(1)
>>> sin(SR(RealField(Integer(150))(Integer(1))))
0.84147098480789650665250232163029899962256306
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(0).sin()
0
sage: SR(0).sin(hold=True)
sin(0)
```

```
>>> from sage.all import *
>>> SR(Integer(0)).sin()
0
>>> SR(Integer(0)).sin(hold=True)
sin(0)
```

This also works using functional notation:

```
sage: sin(0,hold=True)
sin(0)
sage: sin(0)
0
```

```
>>> from sage.all import *
>>> sin(Integer(0),hold=True)
sin(0)
>>> sin(Integer(0))
0
```

To then evaluate again, we use *`unhold()`*:

```
sage: a = SR(0).sin(hold=True); a.unhold()
0
```

```
>>> from sage.all import *
>>> a = SR(Integer(0)).sin(hold=True); a.unhold()
0
```

sinh(*hold=False*)

Return sinh of `self`.

We have $\sinh(x) = (e^x - e^{-x})/2$.

EXAMPLES:

```
sage: x.sinh()
sinh(x)
sage: SR(1).sinh()
sinh(1)
sage: SR(0).sinh()
0
sage: SR(1.0).sinh()
1.17520119364380
sage: maxima('sinh(1.0)')
1.17520119364380...

sinh(1.0000000000000000000000000)
sage: SR(1).sinh().n(90)
1.1752011936438014568823819
sage: SR(RIF(1)).sinh()
1.175201193643802?
```

```
>>> from sage.all import *
>>> x.sinh()
sinh(x)
>>> SR(Integer(1)).sinh()
sinh(1)
>>> SR(Integer(0)).sinh()
0
>>> SR(RealNumber('1.0')).sinh()
1.17520119364380
>>> maxima('sinh(1.0)')
1.17520119364380...

sinh(1.0000000000000000000000000)
>>> SR(Integer(1)).sinh().n(Integer(90))
1.1752011936438014568823819
>>> SR(RIF(Integer(1))).sinh()
1.175201193643802?
```

To prevent automatic evaluation use the `hold` argument:

```
sage: arccosh(x).sinh()
sqrt(x + 1)*sqrt(x - 1)
sage: arccosh(x).sinh(hold=True)
sinh(arccosh(x))
```

```
>>> from sage.all import *
>>> arccosh(x).sinh()
sqrt(x + 1)*sqrt(x - 1)
>>> arccosh(x).sinh(hold=True)
sinh(arccosh(x))
```

This also works using functional notation:

```
sage: sinh(arccosh(x),hold=True)
sinh(arccosh(x))
sage: sinh(arccosh(x))
sqrt(x + 1)*sqrt(x - 1)
```

```
>>> from sage.all import *
>>> sinh(arccosh(x),hold=True)
sinh(arccosh(x))
>>> sinh(arccosh(x))
sqrt(x + 1)*sqrt(x - 1)
```

To then evaluate again, we use `unhold()`:

```
sage: a = arccosh(x).sinh(hold=True); a.simplify()
sqrt(x + 1)*sqrt(x - 1)
```

```
>>> from sage.all import *
>>> a = arccosh(x).sinh(hold=True); a.simplify()
sqrt(x + 1)*sqrt(x - 1)
```

solve(*x*, *multiplicities=False*, *solution_dict=False*, *explicit_solutions=False*, *to_poly_solve=False*, *algorithm=None*, *domain=None*)

Analytically solve the equation `self == 0` or a univariate inequality for the variable x.

> **Warning**
>
> This is not a numerical solver – use `find_root()` to solve for `self == 0` numerically on an interval.

INPUT:

- `x` – variable(s) to solve for
- `multiplicities` – boolean (default: `False`); if `True`, return corresponding multiplicities. This keyword is incompatible with `to_poly_solve=True` and does not make any sense when solving an inequality.
- `solution_dict` – boolean (default: `False`); if `True` or nonzero, return a list of dictionaries containing solutions. Not used when solving an inequality.
- `explicit_solutions` – boolean (default: `False`); require that all roots be explicit rather than implicit. Not used when solving an inequality.
- `to_poly_solve` – boolean (default: `False`) or string; use Maxima's `to_poly_solver` package to search for more possible solutions, but possibly encounter approximate solutions. This keyword is incompatible with `multiplicities=True` and is not used when solving an inequality. Setting `to_poly_solve` to `'force'` omits Maxima's solve command (useful when some solutions of trigonometric equations are lost).

EXAMPLES:

```
sage: z = var('z')
sage: (z^5 - 1).solve(z)
[z == 1/4*sqrt(5) + 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,
```

(continues on next page)

(continued from previous page)

```
 z == -1/4*sqrt(5) + 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,
 z == -1/4*sqrt(5) - 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,
 z == 1/4*sqrt(5) - 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,
 z == 1]

sage: solve((z^3-1)^3, z, multiplicities=True)
([z == 1/2*I*sqrt(3) - 1/2, z == -1/2*I*sqrt(3) - 1/2, z == 1], [3, 3, 3])
```

```
>>> from sage.all import *
>>> z = var('z')
>>> (z**Integer(5) - Integer(1)).solve(z)
[z == 1/4*sqrt(5) + 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,
 z == -1/4*sqrt(5) + 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,
 z == -1/4*sqrt(5) - 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,
 z == 1/4*sqrt(5) - 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,
 z == 1]

>>> solve((z**Integer(3)-Integer(1))**Integer(3), z, multiplicities=True)
([z == 1/2*I*sqrt(3) - 1/2, z == -1/2*I*sqrt(3) - 1/2, z == 1], [3, 3, 3])
```

solve_diophantine(*x=None*, *solution_dict=False*)

Solve a polynomial equation in the integers (a so called Diophantine).

If the argument is just a polynomial expression, equate to zero. If `solution_dict=True`, return a list of dictionaries instead of a list of tuples.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: solve_diophantine(3*x == 4)                                    #␣
↪needs sympy
[]
sage: solve_diophantine(x^2 - 9)                                     #␣
↪needs sympy
[-3, 3]
sage: sorted(solve_diophantine(x^2 + y^2 == 25))                     #␣
↪needs sympy
[(-5, 0), (-4, -3), (-4, 3), (-3, -4), (-3, 4), (0, -5)...
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> solve_diophantine(Integer(3)*x == Integer(4))                    ␣
↪            # needs sympy
[]
>>> solve_diophantine(x**Integer(2) - Integer(9))                    ␣
↪             # needs sympy
[-3, 3]
>>> sorted(solve_diophantine(x**Integer(2) + y**Integer(2) == Integer(25)))  ␣
↪                       # needs sympy
[(-5, 0), (-4, -3), (-4, 3), (-3, -4), (-3, 4), (0, -5)...
```

The function is used when `solve()` is called with all variables assumed integer:

```
sage: assume(x, 'integer')
sage: assume(y, 'integer')
sage: sorted(solve(x*y == 1, (x,y)))                                          #␣
↪needs sympy
[(-1, -1), (1, 1)]
```

```
>>> from sage.all import *
>>> assume(x, 'integer')
>>> assume(y, 'integer')
>>> sorted(solve(x*y == Integer(1), (x,y)))                                   ␣
↪     # needs sympy
[(-1, -1), (1, 1)]
```

You can also pick specific variables, and get the solution as a dictionary:

```
sage: # needs sympy
sage: solve_diophantine(x*y == 10, x)
[-10, -5, -2, -1, 1, 2, 5, 10]
sage: sorted(solve_diophantine(x*y - y == 10, (x,y)))
[(-9, -1), (-4, -2), (-1, -5), (0, -10), (2, 10), (3, 5), (6, 2), (11, 1)]
sage: res = solve_diophantine(x*y - y == 10, solution_dict=True)
sage: sol = [{y: -5, x: -1}, {y: -10, x: 0}, {y: -1, x: -9}, {y: -2, x: -4},
....:        {y: 10, x: 2}, {y: 1, x: 11}, {y: 2, x: 6}, {y: 5, x: 3}]
sage: all(solution in res
....:     for solution in sol) and bool(len(res) == len(sol))
True
```

```
>>> from sage.all import *
>>> # needs sympy
>>> solve_diophantine(x*y == Integer(10), x)
[-10, -5, -2, -1, 1, 2, 5, 10]
>>> sorted(solve_diophantine(x*y - y == Integer(10), (x,y)))
[(-9, -1), (-4, -2), (-1, -5), (0, -10), (2, 10), (3, 5), (6, 2), (11, 1)]
>>> res = solve_diophantine(x*y - y == Integer(10), solution_dict=True)
>>> sol = [{y: -Integer(5), x: -Integer(1)}, {y: -Integer(10), x: Integer(0)},
↪ {y: -Integer(1), x: -Integer(9)}, {y: -Integer(2), x: -Integer(4)},
...        {y: Integer(10), x: Integer(2)}, {y: Integer(1), x: Integer(11)},
↪{y: Integer(2), x: Integer(6)}, {y: Integer(5), x: Integer(3)}]
>>> all(solution in res
...     for solution in sol) and bool(len(res) == len(sol))
True
```

If the solution is parametrized the parameter(s) are not defined, but you can substitute them with specific integer values:

```
sage: # needs sympy
sage: x,y,z = var('x,y,z')
sage: sol = solve_diophantine(x^2-y == 0); sol
(t, t^2)
sage: [(sol[0].subs(t=t),sol[1].subs(t=t)) for t in range(-3,4)]
[(-3, 9), (-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4), (3, 9)]
sage: sol = solve_diophantine(x^2 + y^2 == z^2); sol
(2*p*q, p^2 - q^2, p^2 + q^2)
```

(continues on next page)

(continued from previous page)

```
sage: [(sol[0].subs(p=p,q=q), sol[1].subs(p=p,q=q), sol[2].subs(p=p,q=q))
....:  for p in range(1,4) for q in range(1,4)]
[(2, 0, 2), (4, -3, 5), (6, -8, 10), (4, 3, 5), (8, 0, 8),
 (12, -5, 13), (6, 8, 10), (12, 5, 13), (18, 0, 18)]
```

```
>>> from sage.all import *
>>> # needs sympy
>>> x,y,z = var('x,y,z')
>>> sol = solve_diophantine(x**Integer(2)-y == Integer(0)); sol
(t, t^2)
>>> [(sol[Integer(0)].subs(t=t),sol[Integer(1)].subs(t=t)) for t in range(-
↪Integer(3),Integer(4))]
[(-3, 9), (-2, 4), (-1, 1), (0, 0), (1, 1), (2, 4), (3, 9)]
>>> sol = solve_diophantine(x**Integer(2) + y**Integer(2) == z**Integer(2));␣
↪sol
(2*p*q, p^2 - q^2, p^2 + q^2)
>>> [(sol[Integer(0)].subs(p=p,q=q), sol[Integer(1)].subs(p=p,q=q),␣
↪sol[Integer(2)].subs(p=p,q=q))
...  for p in range(Integer(1),Integer(4)) for q in range(Integer(1),
↪Integer(4))]
[(2, 0, 2), (4, -3, 5), (6, -8, 10), (4, 3, 5), (8, 0, 8),
 (12, -5, 13), (6, 8, 10), (12, 5, 13), (18, 0, 18)]
```

Solve Brahmagupta-Pell equations:

```
sage: sol = sorted(solve_diophantine(x^2 - 2*y^2 == 1), key=str); sol       #␣
↪needs sympy
[(-sqrt(2)*(2*sqrt(2) + 3)^t + sqrt(2)*(-2*sqrt(2) + 3)^t
   - 3/2*(2*sqrt(2) + 3)^t - 3/2*(-2*sqrt(2) + 3)^t,...
sage: [(sol[1][0].subs(t=t).simplify_full(),                                #␣
↪needs sympy
....:   sol[1][1].subs(t=t).simplify_full()) for t in range(-1,5)]
[(1, 0), (3, -2), (17, -12), (99, -70), (577, -408), (3363, -2378)]
```

```
>>> from sage.all import *
>>> sol = sorted(solve_diophantine(x**Integer(2) - Integer(2)*y**Integer(2)␣
↪== Integer(1)), key=str); sol       # needs sympy
[(-sqrt(2)*(2*sqrt(2) + 3)^t + sqrt(2)*(-2*sqrt(2) + 3)^t
   - 3/2*(2*sqrt(2) + 3)^t - 3/2*(-2*sqrt(2) + 3)^t,...
>>> [(sol[Integer(1)][Integer(0)].subs(t=t).simplify_full(),               ␣
↪               # needs sympy
...   sol[Integer(1)][Integer(1)].subs(t=t).simplify_full()) for t in range(-
↪Integer(1),Integer(5))]
[(1, 0), (3, -2), (17, -12), (99, -70), (577, -408), (3363, -2378)]
```

See also

http://docs.sympy.org/latest/modules/solvers/diophantine.html

sqrt(*hold=False*)

Return the square root of this expression.

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: SR(2).sqrt()
sqrt(2)
sage: (x^2+y^2).sqrt()
sqrt(x^2 + y^2)
sage: (x^2).sqrt()
sqrt(x^2)
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> SR(Integer(2)).sqrt()
sqrt(2)
>>> (x**Integer(2)+y**Integer(2)).sqrt()
sqrt(x^2 + y^2)
>>> (x**Integer(2)).sqrt()
sqrt(x^2)
```

Immediate simplifications are applied:

```
sage: sqrt(x^2)
sqrt(x^2)
sage: x = SR.symbol('x', domain='real')
sage: sqrt(x^2)
abs(x)
sage: forget()
sage: assume(x<0)
sage: sqrt(x^2)
-x
sage: sqrt(x^4)
x^2
sage: forget()
sage: x = SR.symbol('x', domain='real')
sage: sqrt(x^4)
x^2
sage: sqrt(sin(x)^2)
abs(sin(x))
sage: sqrt((x+1)^2)
abs(x + 1)
sage: forget()
sage: assume(x<0)
sage: sqrt((x-1)^2)
-x + 1
sage: forget()
```

```
>>> from sage.all import *
>>> sqrt(x**Integer(2))
sqrt(x^2)
>>> x = SR.symbol('x', domain='real')
>>> sqrt(x**Integer(2))
```

(continues on next page)

(continued from previous page)

```
abs(x)
>>> forget()
>>> assume(x<Integer(0))
>>> sqrt(x**Integer(2))
-x
>>> sqrt(x**Integer(4))
x^2
>>> forget()
>>> x = SR.symbol('x', domain='real')
>>> sqrt(x**Integer(4))
x^2
>>> sqrt(sin(x)**Integer(2))
abs(sin(x))
>>> sqrt((x+Integer(1))**Integer(2))
abs(x + 1)
>>> forget()
>>> assume(x<Integer(0))
>>> sqrt((x-Integer(1))**Integer(2))
-x + 1
>>> forget()
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(4).sqrt()
2
sage: SR(4).sqrt(hold=True)
sqrt(4)
```

```
>>> from sage.all import *
>>> SR(Integer(4)).sqrt()
2
>>> SR(Integer(4)).sqrt(hold=True)
sqrt(4)
```

To then evaluate again, we use *`unhold()`*:

```
sage: a = SR(4).sqrt(hold=True); a.unhold()
2
```

```
>>> from sage.all import *
>>> a = SR(Integer(4)).sqrt(hold=True); a.unhold()
2
```

To use this parameter in functional notation, you must coerce to the symbolic ring:

```
sage: sqrt(SR(4),hold=True)
sqrt(4)
sage: sqrt(4,hold=True)
Traceback (most recent call last):
...
TypeError: ..._do_sqrt() got an unexpected keyword argument 'hold'
```

```
>>> from sage.all import *
>>> sqrt(SR(Integer(4)),hold=True)
sqrt(4)
>>> sqrt(Integer(4),hold=True)
Traceback (most recent call last):
...
TypeError: ..._do_sqrt() got an unexpected keyword argument 'hold'
```

step(*hold=False*)

Return the value of the unit step function, which is 0 for negative x, 1 for 0, and 1 for positive x.

See also

sage.functions.generalized.FunctionUnitStep

EXAMPLES:

```
sage: x = var('x')
sage: SR(1.5).step()
1
sage: SR(0).step()
1
sage: SR(-1/2).step()
0
sage: SR(float(-1)).step()
0
```

```
>>> from sage.all import *
>>> x = var('x')
>>> SR(RealNumber('1.5')).step()
1
>>> SR(Integer(0)).step()
1
>>> SR(-Integer(1)/Integer(2)).step()
0
>>> SR(float(-Integer(1))).step()
0
```

Using the `hold` parameter it is possible to prevent automatic evaluation:

```
sage: SR(2).step()
1
sage: SR(2).step(hold=True)
unit_step(2)
```

```
>>> from sage.all import *
>>> SR(Integer(2)).step()
1
>>> SR(Integer(2)).step(hold=True)
unit_step(2)
```

subs(**args*, ***kwds*)

Substitute the given subexpressions in this expression.

EXAMPLES:

```
sage: var('x,y,z,a,b,c,d,f,g')
(x, y, z, a, b, c, d, f, g)
sage: w0 = SR.wild(0); w1 = SR.wild(1)
sage: t = a^2 + b^2 + (x+y)^3
```

```
>>> from sage.all import *
>>> var('x,y,z,a,b,c,d,f,g')
(x, y, z, a, b, c, d, f, g)
>>> w0 = SR.wild(Integer(0)); w1 = SR.wild(Integer(1))
>>> t = a**Integer(2) + b**Integer(2) + (x+y)**Integer(3)
```

Substitute with keyword arguments (works only with symbols):

```
sage: t.subs(a=c)
(x + y)^3 + b^2 + c^2
sage: t.subs(b=19, x=z)
(y + z)^3 + a^2 + 361
```

```
>>> from sage.all import *
>>> t.subs(a=c)
(x + y)^3 + b^2 + c^2
>>> t.subs(b=Integer(19), x=z)
(y + z)^3 + a^2 + 361
```

Substitute with a dictionary argument:

```
sage: t.subs({a^2: c})
(x + y)^3 + b^2 + c

sage: t.subs({w0^2: w0^3})
a^3 + b^3 + (x + y)^3
```

```
>>> from sage.all import *
>>> t.subs({a**Integer(2): c})
(x + y)^3 + b^2 + c

>>> t.subs({w0**Integer(2): w0**Integer(3)})
a^3 + b^3 + (x + y)^3
```

Substitute with one or more relational expressions:

```
sage: t.subs(w0^2 == w0^3)
a^3 + b^3 + (x + y)^3

sage: t.subs(w0 == w0^2)
a^8 + b^8 + (x^2 + y^2)^6

sage: t.subs(a == b, b == c)
(x + y)^3 + b^2 + c^2
```

```
>>> from sage.all import *
>>> t.subs(w0**Integer(2) == w0**Integer(3))
a^3 + b^3 + (x + y)^3

>>> t.subs(w0 == w0**Integer(2))
a^8 + b^8 + (x^2 + y^2)^6

>>> t.subs(a == b, b == c)
(x + y)^3 + b^2 + c^2
```

Any number of arguments is accepted:

```
sage: t.subs(a=b, b=c)
(x + y)^3 + b^2 + c^2

sage: t.subs({a:b}, b=c)
(x + y)^3 + b^2 + c^2

sage: t.subs([x == 3, y == 2], a == 2, {b:3})
138
```

```
>>> from sage.all import *
>>> t.subs(a=b, b=c)
(x + y)^3 + b^2 + c^2

>>> t.subs({a:b}, b=c)
(x + y)^3 + b^2 + c^2

>>> t.subs([x == Integer(3), y == Integer(2)], a == Integer(2), {b:Integer(3)}
↪)
138
```

It can even accept lists of lists:

```
sage: eqn1 = (a*x + b*y == 0)
sage: eqn2 = (1 + y == 0)
sage: soln = solve([eqn1, eqn2], [x, y])
sage: soln
[[x == b/a, y == -1]]
sage: f = x + y
sage: f.subs(soln)
b/a - 1
```

```
>>> from sage.all import *
>>> eqn1 = (a*x + b*y == Integer(0))
>>> eqn2 = (Integer(1) + y == Integer(0))
>>> soln = solve([eqn1, eqn2], [x, y])
>>> soln
[[x == b/a, y == -1]]
>>> f = x + y
>>> f.subs(soln)
b/a - 1
```

Duplicate assignments will throw an error:

```
sage: t.subs({a:b}, a=c)
Traceback (most recent call last):
...
ValueError: duplicate substitution for a, got values b and c

sage: t.subs([x == 1], a = 1, b = 2, x = 2)
Traceback (most recent call last):
...
ValueError: duplicate substitution for x, got values 1 and 2
```

```
>>> from sage.all import *
>>> t.subs({a:b}, a=c)
Traceback (most recent call last):
...
ValueError: duplicate substitution for a, got values b and c

>>> t.subs([x == Integer(1)], a = Integer(1), b = Integer(2), x = Integer(2))
Traceback (most recent call last):
...
ValueError: duplicate substitution for x, got values 1 and 2
```

All substitutions are performed at the same time:

```
sage: t.subs({a:b, b:c})
(x + y)^3 + b^2 + c^2
```

```
>>> from sage.all import *
>>> t.subs({a:b, b:c})
(x + y)^3 + b^2 + c^2
```

Substitutions are done term by term, in other words Sage is not able to identify partial sums in a substitution (see Issue #18396):

```
sage: f = x + x^2 + x^4
sage: f.subs(x = y)
y^4 + y^2 + y
sage: f.subs(x^2 == y)             # one term is fine
x^4 + x + y
sage: f.subs(x + x^2 == y)         # partial sum does not work
x^4 + x^2 + x
sage: f.subs(x + x^2 + x^4 == y)   # whole sum is fine
y
```

```
>>> from sage.all import *
>>> f = x + x**Integer(2) + x**Integer(4)
>>> f.subs(x = y)
y^4 + y^2 + y
>>> f.subs(x**Integer(2) == y)             # one term is fine
x^4 + x + y
>>> f.subs(x + x**Integer(2) == y)         # partial sum does not work
x^4 + x^2 + x
```

(continues on next page)

(continued from previous page)

```
>>> f.subs(x + x**Integer(2) + x**Integer(4) == y)    # whole sum is fine
y
```

Note that it is the very same behavior as in Maxima:

```
sage: E = 'x^4 + x^2 + x'
sage: subs = [('x','y'), ('x^2','y'), ('x^2+x','y'), ('x^4+x^2+x','y')]

sage: cmd = '{}, {}={}'
sage: for s1,s2 in subs:
....:     maxima.eval(cmd.format(E, s1, s2))
'y^4+y^2+y'
'y+x^4+x'
'x^4+x^2+x'
'y'
```

```
>>> from sage.all import *
>>> E = 'x^4 + x^2 + x'
>>> subs = [('x','y'), ('x^2','y'), ('x^2+x','y'), ('x^4+x^2+x','y')]

>>> cmd = '{}, {}={}'
>>> for s1,s2 in subs:
...     maxima.eval(cmd.format(E, s1, s2))
'y^4+y^2+y'
'y+x^4+x'
'x^4+x^2+x'
'y'
```

Or as in Maple:

```
sage: cmd = 'subs({}={}, {})'                # optional - maple
sage: for s1,s2 in subs:                     # optional - maple
....:     maple.eval(cmd.format(s1,s2, E))
'y^4+y^2+y'
'x^4+x+y'
'x^4+x^2+x'
'y'
```

```
>>> from sage.all import *
>>> cmd = 'subs({}={}, {})'                # optional - maple
>>> for s1,s2 in subs:                     # optional - maple
...     maple.eval(cmd.format(s1,s2, E))
'y^4+y^2+y'
'x^4+x+y'
'x^4+x^2+x'
'y'
```

But Mathematica does something different on the third example:

```
sage: cmd = '{} /. {} -> {}'                    # optional - mathematica
sage: for s1,s2 in subs:                        # optional - mathematica
....:     mathematica.eval(cmd.format(E,s1,s2))
```

(continues on next page)

(continued from previous page)

```
     2    4
y + y  + y
     4
x + x  + y
 4
x  + y
y
```

```
>>> from sage.all import *
>>> cmd = '{} /. {} -> {}'                    # optional - mathematica
>>> for s1,s2 in subs:                        # optional - mathematica
...     mathematica.eval(cmd.format(E,s1,s2))
     2    4
y + y  + y
     4
x + x  + y
 4
x  + y
y
```

The same, with formatting more suitable for cut and paste:

```
sage: for s1,s2 in subs:                        # optional - mathematica
....:     mathematica(cmd.format(E,s1,s2))
y + y^2 + y^4
x + x^4 + y
x^4 + y
y
```

```
>>> from sage.all import *
>>> for s1,s2 in subs:                        # optional - mathematica
...     mathematica(cmd.format(E,s1,s2))
y + y^2 + y^4
x + x^4 + y
x^4 + y
y
```

Warning

Unexpected results may occur if the left-hand side of some substitution is not just a single variable (or is a "wildcard" variable). For example, the result of `cos(cos(cos(x))).subs({cos(x) : x})` is `x`, because the substitution is applied repeatedly. Such repeated substitutions (and pattern-matching code that may be somewhat unpredictable) are disabled only in the basic case where the left-hand side of every substitution is a variable. In particular, although the result of `(x^2).subs({x : sqrt(x)})` is `x`, the result of `(x^2).subs({x : sqrt(x), y^2 : y})` is `sqrt(x)`, because repeated substitution is enabled by the presence of the expression `y^2` in the left-hand side of one of the substitutions, even though that particular substitution does not get applied.

substitute_function(**args*, ***kwds*)

Substitute the given functions by their replacements in this expression.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: foo = function('foo'); bar = function('bar')
sage: f = foo(x) + 1/foo(pi*y)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> foo = function('foo'); bar = function('bar')
>>> f = foo(x) + Integer(1)/foo(pi*y)
```

Substitute with a dictionary:

```
sage: f.substitute_function({foo: bar})
1/bar(pi*y) + bar(x)
sage: f.substitute_function({foo(x): bar(x)})
1/bar(pi*y) + bar(x)
```

```
>>> from sage.all import *
>>> f.substitute_function({foo: bar})
1/bar(pi*y) + bar(x)
>>> f.substitute_function({foo(x): bar(x)})
1/bar(pi*y) + bar(x)
```

If the function expression to be substituted includes its arguments, the right hand side can be an arbitrary symbolic expression:

```
sage: f.substitute_function({foo(x): x^2})
x^2 + 1/(pi^2*y^2)
```

```
>>> from sage.all import *
>>> f.substitute_function({foo(x): x**Integer(2)})
x^2 + 1/(pi^2*y^2)
```

Substitute with keyword arguments (works only if no function arguments are given):

```
sage: f.substitute_function(foo=bar)
1/bar(pi*y) + bar(x)
```

```
>>> from sage.all import *
>>> f.substitute_function(foo=bar)
1/bar(pi*y) + bar(x)
```

Substitute with a relational expression:

```
sage: f.substitute_function(foo(x)==bar(x))
1/bar(pi*y) + bar(x)
sage: f.substitute_function(foo(x)==bar(x+1))
1/bar(pi*y + 1) + bar(x + 1)
```

```
>>> from sage.all import *
>>> f.substitute_function(foo(x)==bar(x))
1/bar(pi*y) + bar(x)
```

(continues on next page)

(continued from previous page)

```
>>> f.substitute_function(foo(x)==bar(x+Integer(1)))
1/bar(pi*y + 1) + bar(x + 1)
```

All substitutions are performed at the same time:

```
sage: g = foo(x) + 1/bar(pi*y)
sage: g.substitute_function({foo: bar, bar: foo})
1/foo(pi*y) + bar(x)
```

```
>>> from sage.all import *
>>> g = foo(x) + Integer(1)/bar(pi*y)
>>> g.substitute_function({foo: bar, bar: foo})
1/foo(pi*y) + bar(x)
```

Any number of arguments is accepted:

```
sage: g.substitute_function({foo: bar}, bar(x) == x^2)
1/(pi^2*y^2) + bar(x)
```

```
>>> from sage.all import *
>>> g.substitute_function({foo: bar}, bar(x) == x**Integer(2))
1/(pi^2*y^2) + bar(x)
```

As well as lists of substitutions:

```
sage: g.substitute_function([foo(x) == 1, bar(x) == x])
1/(pi*y) + 1
```

```
>>> from sage.all import *
>>> g.substitute_function([foo(x) == Integer(1), bar(x) == x])
1/(pi*y) + 1
```

Alternative syntax:

```
sage: g.substitute_function(foo, bar)
1/bar(pi*y) + bar(x)
```

```
>>> from sage.all import *
>>> g.substitute_function(foo, bar)
1/bar(pi*y) + bar(x)
```

Duplicate assignments will throw an error:

```
sage: g.substitute_function({foo:bar}, foo(x) == x^2)
Traceback (most recent call last):
...
ValueError: duplicate substitution for foo, got values bar and x |--> x^2

sage: g.substitute_function([foo(x) == x^2], foo = bar)
Traceback (most recent call last):
...
ValueError: duplicate substitution for foo, got values x |--> x^2 and bar
```

```
>>> from sage.all import *
>>> g.substitute_function({foo:bar}, foo(x) == x**Integer(2))
Traceback (most recent call last):
...
ValueError: duplicate substitution for foo, got values bar and x |--> x^2

>>> g.substitute_function([foo(x) == x**Integer(2)], foo = bar)
Traceback (most recent call last):
...
ValueError: duplicate substitution for foo, got values x |--> x^2 and bar
```

substitution_delayed(*pattern*, *replacement*)

Replace all occurrences of pattern by the result of replacement.

In contrast to *subs()*, the pattern may contains wildcards and the replacement can depend on the particular term matched by the pattern.

INPUT:

- `pattern` – an *Expression*, usually containing wildcards
- `replacement` – a function; its argument is a dictionary mapping the wildcard occurring in `pattern` to the actual values. If it returns `None`, this occurrence of `pattern` is not replaced. Otherwise, it is replaced by the output of `replacement`.

OUTPUT: an *Expression*

EXAMPLES:

```
sage: var('x y')
(x, y)
sage: w0 = SR.wild(0)
sage: sqrt(1 + 2*x + x^2).substitution_delayed(
....:     sqrt(w0), lambda d: sqrt(factor(d[w0]))
....: )
sqrt((x + 1)^2)
sage: def r(d):
....:    if x not in d[w0].variables():
....:        return cos(d[w0])
sage: (sin(x^2 + x) + sin(y^2 + y)).substitution_delayed(sin(w0), r)
cos(y^2 + y) + sin(x^2 + x)
```

```
>>> from sage.all import *
>>> var('x y')
(x, y)
>>> w0 = SR.wild(Integer(0))
>>> sqrt(Integer(1) + Integer(2)*x + x**Integer(2)).substitution_delayed(
...     sqrt(w0), lambda d: sqrt(factor(d[w0]))
... )
sqrt((x + 1)^2)
>>> def r(d):
...    if x not in d[w0].variables():
...        return cos(d[w0])
>>> (sin(x**Integer(2) + x) + sin(y**Integer(2) + y)).substitution_
↪delayed(sin(w0), r)
cos(y^2 + y) + sin(x^2 + x)
```

See also

match()

subtract_from_both_sides(*x*)

Return a relation obtained by subtracting x from both sides of this relation.

EXAMPLES:

```
sage: eqn = x*sin(x)*sqrt(3) + sqrt(2) > cos(sin(x))
sage: eqn.subtract_from_both_sides(sqrt(2))
sqrt(3)*x*sin(x) > -sqrt(2) + cos(sin(x))
sage: eqn.subtract_from_both_sides(cos(sin(x)))
sqrt(3)*x*sin(x) + sqrt(2) - cos(sin(x)) > 0
```

```
>>> from sage.all import *
>>> eqn = x*sin(x)*sqrt(Integer(3)) + sqrt(Integer(2)) > cos(sin(x))
>>> eqn.subtract_from_both_sides(sqrt(Integer(2)))
sqrt(3)*x*sin(x) > -sqrt(2) + cos(sin(x))
>>> eqn.subtract_from_both_sides(cos(sin(x)))
sqrt(3)*x*sin(x) + sqrt(2) - cos(sin(x)) > 0
```

sum(**args*, ***kwds*)

Return the symbolic sum $\sum_{v=a}^{b}$ self.

with respect to the variable v with endpoints a and b.

INPUT:

- v – a variable or variable name
- a – lower endpoint of the sum
- b – upper endpoint of the sum
- algorithm – (default: 'maxima') one of
  - 'maxima' – use Maxima (the default)
  - 'maple' – (optional) use Maple
  - 'mathematica' – (optional) use Mathematica
  - 'giac' – (optional) use Giac
  - 'sympy' – use SymPy

EXAMPLES:

```
sage: k, n = var('k,n')
sage: k.sum(k, 1, n).factor()
1/2*(n + 1)*n
```

```
>>> from sage.all import *
>>> k, n = var('k,n')
>>> k.sum(k, Integer(1), n).factor()
1/2*(n + 1)*n
```

```
sage: (1/k^4).sum(k, 1, oo)
1/90*pi^4
```

```
>>> from sage.all import *
>>> (Integer(1)/k**Integer(4)).sum(k, Integer(1), oo)
1/90*pi^4
```

```
sage: (1/k^5).sum(k, 1, oo)
zeta(5)
```

```
>>> from sage.all import *
>>> (Integer(1)/k**Integer(5)).sum(k, Integer(1), oo)
zeta(5)
```

A well known binomial identity:

```
sage: assume(n>=0)
sage: binomial(n,k).sum(k, 0, n)
2^n
```

```
>>> from sage.all import *
>>> assume(n>=Integer(0))
>>> binomial(n,k).sum(k, Integer(0), n)
2^n
```

And some truncations thereof:

```
sage: binomial(n,k).sum(k,1,n)
2^n - 1
sage: binomial(n,k).sum(k,2,n)
2^n - n - 1
sage: binomial(n,k).sum(k,0,n-1)
2^n - 1
sage: binomial(n,k).sum(k,1,n-1)
2^n - 2
```

```
>>> from sage.all import *
>>> binomial(n,k).sum(k,Integer(1),n)
2^n - 1
>>> binomial(n,k).sum(k,Integer(2),n)
2^n - n - 1
>>> binomial(n,k).sum(k,Integer(0),n-Integer(1))
2^n - 1
>>> binomial(n,k).sum(k,Integer(1),n-Integer(1))
2^n - 2
```

The binomial theorem:

```
sage: x, y = var('x, y')
sage: (binomial(n,k) * x^k * y^(n-k)).sum(k, 0, n)
(x + y)^n
```

```
>>> from sage.all import *
>>> x, y = var('x, y')
>>> (binomial(n,k) * x**k * y**(n-k)).sum(k, Integer(0), n)
(x + y)^n
```

```
sage: (k * binomial(n, k)).sum(k, 1, n)
2^(n - 1)*n
```

```
>>> from sage.all import *
>>> (k * binomial(n, k)).sum(k, Integer(1), n)
2^(n - 1)*n
```

```
sage: ((-1)^k*binomial(n,k)).sum(k, 0, n)
0
```

```
>>> from sage.all import *
>>> ((-Integer(1))**k*binomial(n,k)).sum(k, Integer(0), n)
0
```

```
sage: (2^(-k)/(k*(k+1))).sum(k, 1, oo)
-log(2) + 1
```

```
>>> from sage.all import *
>>> (Integer(2)**(-k)/(k*(k+Integer(1)))).sum(k, Integer(1), oo)
-log(2) + 1
```

Summing a hypergeometric term:

```
sage: (binomial(n, k) * factorial(k) / factorial(n+1+k)).sum(k, 0, n)
1/2*sqrt(pi)/factorial(n + 1/2)
```

```
>>> from sage.all import *
>>> (binomial(n, k) * factorial(k) / factorial(n+Integer(1)+k)).sum(k,␣
↪Integer(0), n)
1/2*sqrt(pi)/factorial(n + 1/2)
```

We check a well known identity:

```
sage: bool((k^3).sum(k, 1, n) == k.sum(k, 1, n)^2)
True
```

```
>>> from sage.all import *
>>> bool((k**Integer(3)).sum(k, Integer(1), n) == k.sum(k, Integer(1),␣
↪n)**Integer(2))
True
```

A geometric sum:

```
sage: a, q = var('a, q')
sage: (a*q^k).sum(k, 0, n)
(a*q^(n + 1) - a)/(q - 1)
```

```
>>> from sage.all import *
>>> a, q = var('a, q')
>>> (a*q**k).sum(k, Integer(0), n)
(a*q^(n + 1) - a)/(q - 1)
```

The geometric series:

```
sage: assume(abs(q) < 1)
sage: (a*q^k).sum(k, 0, oo)
-a/(q - 1)
```

```
>>> from sage.all import *
>>> assume(abs(q) < Integer(1))
>>> (a*q**k).sum(k, Integer(0), oo)
-a/(q - 1)
```

A divergent geometric series. Do not forget to $forget$ your assumptions:

```
sage: forget()
sage: assume(q > 1)
sage: (a*q^k).sum(k, 0, oo)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
```

```
>>> from sage.all import *
>>> forget()
>>> assume(q > Integer(1))
>>> (a*q**k).sum(k, Integer(0), oo)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
```

This summation only Mathematica can perform:

```
sage: (1/(1+k^2)).sum(k, -oo, oo, algorithm = 'mathematica')     # optional -␣
↪mathematica
pi*coth(pi)
```

```
>>> from sage.all import *
>>> (Integer(1)/(Integer(1)+k**Integer(2))).sum(k, -oo, oo, algorithm =
↪'mathematica')     # optional - mathematica
pi*coth(pi)
```

Use Giac to perform this summation:

```
sage: # needs giac
sage: (sum(1/(1+k^2), k, -oo, oo, algorithm = 'giac')).factor()
pi*(e^(2*pi) + 1)/((e^pi + 1)*(e^pi - 1))
```

```
>>> from sage.all import *
>>> # needs giac
```

(continues on next page)

(continued from previous page)

```
>>> (sum(Integer(1)/(Integer(1)+k**Integer(2)), k, -oo, oo, algorithm = 'giac
↪')).factor()
pi*(e^(2*pi) + 1)/((e^pi + 1)*(e^pi - 1))
```

Use Maple as a backend for summation:

```
sage: (binomial(n,k)*x^k).sum(k, 0, n, algorithm = 'maple')      # optional -␣
↪maple
(x + 1)^n
```

```
>>> from sage.all import *
>>> (binomial(n,k)*x**k).sum(k, Integer(0), n, algorithm = 'maple')      #␣
↪optional - maple
(x + 1)^n
```

Note

1. Sage can currently only understand a subset of the output of Maxima, Maple and Mathematica, so even if the chosen backend can perform the summation the result might not be convertible into a usable Sage expression.

tan(*hold=False*)

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: tan(x^2 + y^2)
tan(x^2 + y^2)
sage: tan(sage.symbolic.constants.pi/2)
Infinity
sage: tan(SR(1))
tan(1)
sage: tan(SR(RealField(150)(1)))
1.5574077246549022305069748074583601730872508
```

```
>>> from sage.all import *
>>> var('x, y')
(x, y)
>>> tan(x**Integer(2) + y**Integer(2))
tan(x^2 + y^2)
>>> tan(sage.symbolic.constants.pi/Integer(2))
Infinity
>>> tan(SR(Integer(1)))
tan(1)
>>> tan(SR(RealField(Integer(150))(Integer(1))))
1.5574077246549022305069748074583601730872508
```

To prevent automatic evaluation use the `hold` argument:

```
sage: (pi/12).tan()
-sqrt(3) + 2
sage: (pi/12).tan(hold=True)
tan(1/12*pi)
```

```
>>> from sage.all import *
>>> (pi/Integer(12)).tan()
-sqrt(3) + 2
>>> (pi/Integer(12)).tan(hold=True)
tan(1/12*pi)
```

This also works using functional notation:

```
sage: tan(pi/12,hold=True)
tan(1/12*pi)
sage: tan(pi/12)
-sqrt(3) + 2
```

```
>>> from sage.all import *
>>> tan(pi/Integer(12),hold=True)
tan(1/12*pi)
>>> tan(pi/Integer(12))
-sqrt(3) + 2
```

To then evaluate again, we use `unhold()`:

```
sage: a = (pi/12).tan(hold=True); a.unhold()
-sqrt(3) + 2
```

```
>>> from sage.all import *
>>> a = (pi/Integer(12)).tan(hold=True); a.unhold()
-sqrt(3) + 2
```

tanh(*hold=False*)

Return tanh of `self`.

We have $\tanh(x) = \sinh(x)/\cosh(x)$.

EXAMPLES:

```
sage: x.tanh()
tanh(x)
sage: SR(1).tanh()
tanh(1)
sage: SR(0).tanh()
0
sage: SR(1.0).tanh()
0.761594155955765
sage: maxima('tanh(1.0)')
0.7615941559557649
sage: plot(lambda x: SR(x).tanh(), -1, 1)                              #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> x.tanh()
tanh(x)
>>> SR(Integer(1)).tanh()
tanh(1)
>>> SR(Integer(0)).tanh()
0
>>> SR(RealNumber('1.0')).tanh()
0.761594155955765
>>> maxima('tanh(1.0)')
0.7615941559557649
>>> plot(lambda x: SR(x).tanh(), -Integer(1), Integer(1))                    ␣
↪               # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

To prevent automatic evaluation use the `hold` argument:

```
sage: arcsinh(x).tanh()
x/sqrt(x^2 + 1)
sage: arcsinh(x).tanh(hold=True)
tanh(arcsinh(x))
```

```
>>> from sage.all import *
>>> arcsinh(x).tanh()
x/sqrt(x^2 + 1)
>>> arcsinh(x).tanh(hold=True)
tanh(arcsinh(x))
```

This also works using functional notation:

```
sage: tanh(arcsinh(x), hold=True)
tanh(arcsinh(x))
sage: tanh(arcsinh(x))
x/sqrt(x^2 + 1)
```

```
>>> from sage.all import *
>>> tanh(arcsinh(x), hold=True)
tanh(arcsinh(x))
>>> tanh(arcsinh(x))
x/sqrt(x^2 + 1)
```

To then evaluate again, we use *unhold()*:

```
sage: a = arcsinh(x).tanh(hold=True); a.unhold()
x/sqrt(x^2 + 1)
```

```
>>> from sage.all import *
>>> a = arcsinh(x).tanh(hold=True); a.unhold()
x/sqrt(x^2 + 1)
```

taylor(**args*)

Expand this symbolic expression in a truncated Taylor or Laurent series in the variable v around the point a, containing terms through $(x-a)^n$. Functions in more variables is also supported.

INPUT:

- `*args` – the following notation is supported
  - `x`, `a`, `n` – variable, point, degree
  - `(x, a)`, `(y, b)`, `n` – variables with points, degree of polynomial

EXAMPLES:

```
sage: var('a, x, z')
(a, x, z)
sage: taylor(a*log(z), z, 2, 3)
1/24*a*(z - 2)^3 - 1/8*a*(z - 2)^2 + 1/2*a*(z - 2) + a*log(2)
```

```
>>> from sage.all import *
>>> var('a, x, z')
(a, x, z)
>>> taylor(a*log(z), z, Integer(2), Integer(3))
1/24*a*(z - 2)^3 - 1/8*a*(z - 2)^2 + 1/2*a*(z - 2) + a*log(2)
```

```
sage: taylor(sqrt(sin(x) + a*x + 1), x, 0, 3)
1/48*(3*a^3 + 9*a^2 + 9*a - 1)*x^3 - 1/8*(a^2 + 2*a + 1)*x^2 + 1/2*(a + 1)*x␣
↪+ 1
```

```
>>> from sage.all import *
>>> taylor(sqrt(sin(x) + a*x + Integer(1)), x, Integer(0), Integer(3))
1/48*(3*a^3 + 9*a^2 + 9*a - 1)*x^3 - 1/8*(a^2 + 2*a + 1)*x^2 + 1/2*(a + 1)*x␣
↪+ 1
```

```
sage: taylor(sqrt(x + 1), x, 0, 5)
7/256*x^5 - 5/128*x^4 + 1/16*x^3 - 1/8*x^2 + 1/2*x + 1
```

```
>>> from sage.all import *
>>> taylor(sqrt(x + Integer(1)), x, Integer(0), Integer(5))
7/256*x^5 - 5/128*x^4 + 1/16*x^3 - 1/8*x^2 + 1/2*x + 1
```

```
sage: taylor(1/log(x + 1), x, 0, 3)
-19/720*x^3 + 1/24*x^2 - 1/12*x + 1/x + 1/2
```

```
>>> from sage.all import *
>>> taylor(Integer(1)/log(x + Integer(1)), x, Integer(0), Integer(3))
-19/720*x^3 + 1/24*x^2 - 1/12*x + 1/x + 1/2
```

```
sage: taylor(cos(x) - sec(x), x, 0, 5)
-1/6*x^4 - x^2
```

```
>>> from sage.all import *
>>> taylor(cos(x) - sec(x), x, Integer(0), Integer(5))
-1/6*x^4 - x^2
```

```
sage: taylor((cos(x) - sec(x))^3, x, 0, 9)
-1/2*x^8 - x^6
```

```
>>> from sage.all import *
>>> taylor((cos(x) - sec(x))**Integer(3), x, Integer(0), Integer(9))
-1/2*x^8 - x^6
```

```
sage: taylor(1/(cos(x) - sec(x))^3, x, 0, 5)
-15377/7983360*x^4 - 6767/604800*x^2 + 11/120/x^2 + 1/2/x^4 - 1/x^6 - 347/
↪15120
```

```
>>> from sage.all import *
>>> taylor(Integer(1)/(cos(x) - sec(x))**Integer(3), x, Integer(0),␣
↪Integer(5))
-15377/7983360*x^4 - 6767/604800*x^2 + 11/120/x^2 + 1/2/x^4 - 1/x^6 - 347/
↪15120
```

test_relation(*ntests=20*, *domain=None*, *proof=True*)

Test this relation at several random values, attempting to find a contradiction. If this relation has no variables, it will also test this relation after casting into the domain.

Because the interval fields never return false positives, we can be assured that if `True` or `False` is returned (and proof is `False`) then the answer is correct.

INPUT:

- `ntests` – (default: 20) the number of iterations to run
- `domain` – (optional) the domain from which to draw the random values defaults to `CIF` for equality testing and `RIF` for order testing
- `proof` – (default: `True`) if `False` and the domain is an interval field, regard overlapping (potentially equal) intervals as equal, and return `True` if all tests succeeded.

OUTPUT: boolean or `NotImplemented`, meaning

- `True` – this relation holds in the domain and has no variables
- `False` – a contradiction was found
- `NotImplemented` – no contradiction found

EXAMPLES:

```
sage: (3 < pi).test_relation()
True
sage: (0 >= pi).test_relation()
False
sage: (exp(pi) - pi).n()
19.9990999791895
sage: (exp(pi) - pi == 20).test_relation()
False
sage: (sin(x)^2 + cos(x)^2 == 1).test_relation()
NotImplemented
sage: (sin(x)^2 + cos(x)^2 == 1).test_relation(proof=False)
True
sage: (x == 1).test_relation()
False
sage: var('x,y')
(x, y)
```

(continues on next page)

(continued from previous page)

```
sage: (x < y).test_relation()
False
```

```
>>> from sage.all import *
>>> (Integer(3) < pi).test_relation()
True
>>> (Integer(0) >= pi).test_relation()
False
>>> (exp(pi) - pi).n()
19.9990999791895
>>> (exp(pi) - pi == Integer(20)).test_relation()
False
>>> (sin(x)**Integer(2) + cos(x)**Integer(2) == Integer(1)).test_relation()
NotImplemented
>>> (sin(x)**Integer(2) + cos(x)**Integer(2) == Integer(1)).test_
↪relation(proof=False)
True
>>> (x == Integer(1)).test_relation()
False
>>> var('x,y')
(x, y)
>>> (x < y).test_relation()
False
```

to_gamma()

Convert factorial, binomial, and Pochhammer symbol expressions to their gamma function equivalents.

EXAMPLES:

```
sage: m,n = var('m n', domain='integer')
sage: factorial(n).to_gamma()
gamma(n + 1)
sage: binomial(m,n).to_gamma()
gamma(m + 1)/(gamma(m - n + 1)*gamma(n + 1))
```

```
>>> from sage.all import *
>>> m,n = var('m n', domain='integer')
>>> factorial(n).to_gamma()
gamma(n + 1)
>>> binomial(m,n).to_gamma()
gamma(m + 1)/(gamma(m - n + 1)*gamma(n + 1))
```

trailing_coeff(*s*)

Return the trailing coefficient of s in `self`, i.e., the coefficient of the smallest power of s in `self`.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.trailing_coefficient(x)
```

(continues on next page)

(continued from previous page)

```
2*sin(x*y)
sage: f.trailing_coefficient(y)
x
sage: f.trailing_coefficient(sin(x*y))
a*x + x*y + x/y + 100
```

```
>>> from sage.all import *
>>> var('x,y,a')
(x, y, a)
>>> f = Integer(100) + a*x + x**Integer(3)*sin(x*y) + x*y + x/y +␣
↪Integer(2)*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
>>> f.trailing_coefficient(x)
2*sin(x*y)
>>> f.trailing_coefficient(y)
x
>>> f.trailing_coefficient(sin(x*y))
a*x + x*y + x/y + 100
```

trailing_coefficient(*s*)

Return the trailing coefficient of s in `self`, i.e., the coefficient of the smallest power of s in `self`.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.trailing_coefficient(x)
2*sin(x*y)
sage: f.trailing_coefficient(y)
x
sage: f.trailing_coefficient(sin(x*y))
a*x + x*y + x/y + 100
```

```
>>> from sage.all import *
>>> var('x,y,a')
(x, y, a)
>>> f = Integer(100) + a*x + x**Integer(3)*sin(x*y) + x*y + x/y +␣
↪Integer(2)*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
>>> f.trailing_coefficient(x)
2*sin(x*y)
>>> f.trailing_coefficient(y)
x
>>> f.trailing_coefficient(sin(x*y))
a*x + x*y + x/y + 100
```

trig_expand(*full=False*, *half_angles=False*, *plus=True*, *times=True*)

Expand trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in `self`.

For best results, `self` should already be expanded.

INPUT:

- `full` – boolean (default: `False`); to enhance user control of simplification, this function expands only one level at a time by default, expanding sums of angles or multiple angles. To obtain full expansion into sines and cosines immediately, set the optional parameter full to `True`.
- `half_angles` – boolean (default: `False`); if `True`, causes half-angles to be simplified away
- `plus` – boolean (default: `True`); controls the sum rule. Expansion of sums (e.g. $\sin(x + y)$) will take place only if `plus` is `True`.
- `times` – boolean (default: `True`); controls the product rule, expansion of products (e.g. $\sin(2x)$) will take place only if `times` is `True`.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: sin(5*x).expand_trig()
5*cos(x)^4*sin(x) - 10*cos(x)^2*sin(x)^3 + sin(x)^5
sage: cos(2*x + var('y')).expand_trig()
cos(2*x)*cos(y) - sin(2*x)*sin(y)
```

```
>>> from sage.all import *
>>> sin(Integer(5)*x).expand_trig()
5*cos(x)^4*sin(x) - 10*cos(x)^2*sin(x)^3 + sin(x)^5
>>> cos(Integer(2)*x + var('y')).expand_trig()
cos(2*x)*cos(y) - sin(2*x)*sin(y)
```

We illustrate various options to this function:

```
sage: f = sin(sin(3*cos(2*x))*x)
sage: f.expand_trig()
sin((3*cos(cos(2*x))^2*sin(cos(2*x)) - sin(cos(2*x))^3)*x)
sage: f.expand_trig(full=True)
sin((3*(cos(cos(x)^2)*cos(sin(x)^2)
      + sin(cos(x)^2)*sin(sin(x)^2))^2*(cos(sin(x)^2)*sin(cos(x)^2)
                                       - cos(cos(x)^2)*sin(sin(x)^2))
      - (cos(sin(x)^2)*sin(cos(x)^2) - cos(cos(x)^2)*sin(sin(x)^2))^3)*x)
sage: sin(2*x).expand_trig(times=False)
sin(2*x)
sage: sin(2*x).expand_trig(times=True)
2*cos(x)*sin(x)
sage: sin(2 + x).expand_trig(plus=False)
sin(x + 2)
sage: sin(2 + x).expand_trig(plus=True)
cos(x)*sin(2) + cos(2)*sin(x)
sage: sin(x/2).expand_trig(half_angles=False)
sin(1/2*x)
sage: sin(x/2).expand_trig(half_angles=True)
(-1)^floor(1/2*x/pi)*sqrt(-1/2*cos(x) + 1/2)
```

```
>>> from sage.all import *
>>> f = sin(sin(Integer(3)*cos(Integer(2)*x))*x)
>>> f.expand_trig()
sin((3*cos(cos(2*x))^2*sin(cos(2*x)) - sin(cos(2*x))^3)*x)
>>> f.expand_trig(full=True)
sin((3*(cos(cos(x)^2)*cos(sin(x)^2)
```

(continues on next page)

(continued from previous page)

```
      + sin(cos(x)^2)*sin(sin(x)^2))^2*(cos(sin(x)^2)*sin(cos(x)^2)
                                    - cos(cos(x)^2)*sin(sin(x)^2))
      - (cos(sin(x)^2)*sin(cos(x)^2) - cos(cos(x)^2)*sin(sin(x)^2))^3)*x)
>>> sin(Integer(2)*x).expand_trig(times=False)
sin(2*x)
>>> sin(Integer(2)*x).expand_trig(times=True)
2*cos(x)*sin(x)
>>> sin(Integer(2) + x).expand_trig(plus=False)
sin(x + 2)
>>> sin(Integer(2) + x).expand_trig(plus=True)
cos(x)*sin(2) + cos(2)*sin(x)
>>> sin(x/Integer(2)).expand_trig(half_angles=False)
sin(1/2*x)
>>> sin(x/Integer(2)).expand_trig(half_angles=True)
(-1)^floor(1/2*x/pi)*sqrt(-1/2*cos(x) + 1/2)
```

If the expression contains terms which are factored, we expand first:

```
sage: (x, k1, k2) = var('x, k1, k2')
sage: cos((k1-k2)*x).expand().expand_trig()
cos(k1*x)*cos(k2*x) + sin(k1*x)*sin(k2*x)
```

```
>>> from sage.all import *
>>> (x, k1, k2) = var('x, k1, k2')
>>> cos((k1-k2)*x).expand().expand_trig()
cos(k1*x)*cos(k2*x) + sin(k1*x)*sin(k2*x)
```

ALIAS:

trig_expand() and *expand_trig()* are the same

trig_reduce(*var=None*)

Combine products and powers of trigonometric and hyperbolic sin's and cos's of x into those of multiples of x. It also tries to eliminate these functions when they occur in denominators.

INPUT:

- `self` – a symbolic expression
- `var` – (default: `None`) the variable which is used for these transformations. If not specified, all variables are used.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: y = var('y')
sage: f = sin(x)*cos(x)^3+sin(y)^2
sage: f.reduce_trig()
-1/2*cos(2*y) + 1/8*sin(4*x) + 1/4*sin(2*x) + 1/2
```

```
>>> from sage.all import *
>>> y = var('y')
>>> f = sin(x)*cos(x)**Integer(3)+sin(y)**Integer(2)
```

(continues on next page)

(continued from previous page)

```
>>> f.reduce_trig()
-1/2*cos(2*y) + 1/8*sin(4*x) + 1/4*sin(2*x) + 1/2
```

To reduce only the expressions involving x we use optional parameter:

```
sage: f.reduce_trig(x)
sin(y)^2 + 1/8*sin(4*x) + 1/4*sin(2*x)
```

```
>>> from sage.all import *
>>> f.reduce_trig(x)
sin(y)^2 + 1/8*sin(4*x) + 1/4*sin(2*x)
```

ALIAS: *trig_reduce()* and *reduce_trig()* are the same

trig_simplify(*expand=True*)

Optionally expand and then employ identities such as $\sin(x)^2 + \cos(x)^2 = 1$, $\cosh(x)^2 - \sinh(x)^2 = 1$, $\sin(x)\csc(x) = 1$, or $\tanh(x) = \sinh(x)/\cosh(x)$ to simplify expressions containing tan, sec, etc., to sin, cos, sinh, cosh.

INPUT:

- `self` – symbolic expression
- `expand` – boolean (default: `True`); if `True`, expands trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in `self` first. For best results, `self` should be expanded. See also *expand_trig()* to get more controls on this expansion.

ALIAS: *trig_simplify()* and *simplify_trig()* are the same

EXAMPLES:

```
sage: f = sin(x)^2 + cos(x)^2; f
cos(x)^2 + sin(x)^2
sage: f.simplify()
cos(x)^2 + sin(x)^2
sage: f.simplify_trig()
1
sage: h = sin(x)*csc(x)
sage: h.simplify_trig()
1
sage: k = tanh(x)*cosh(2*x)
sage: k.simplify_trig()
(2*sinh(x)^3 + sinh(x))/cosh(x)
```

```
>>> from sage.all import *
>>> f = sin(x)**Integer(2) + cos(x)**Integer(2); f
cos(x)^2 + sin(x)^2
>>> f.simplify()
cos(x)^2 + sin(x)^2
>>> f.simplify_trig()
1
>>> h = sin(x)*csc(x)
>>> h.simplify_trig()
1
>>> k = tanh(x)*cosh(Integer(2)*x)
```

(continues on next page)

(continued from previous page)

```
>>> k.simplify_trig()
(2*sinh(x)^3 + sinh(x))/cosh(x)
```

In some cases we do not want to expand:

```
sage: f = tan(3*x)
sage: f.simplify_trig()
-(4*cos(x)^2 - 1)*sin(x)/(4*cos(x)*sin(x)^2 - cos(x))
sage: f.simplify_trig(False)
sin(3*x)/cos(3*x)
```

```
>>> from sage.all import *
>>> f = tan(Integer(3)*x)
>>> f.simplify_trig()
-(4*cos(x)^2 - 1)*sin(x)/(4*cos(x)*sin(x)^2 - cos(x))
>>> f.simplify_trig(False)
sin(3*x)/cos(3*x)
```

truncate()

Given a power series or expression, return the corresponding expression without the big oh.

INPUT:

- `self` – a series as output by the *series()* command

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: f = sin(x)/x^2
sage: f.truncate()
sin(x)/x^2
sage: f.series(x,7)
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
sage: f.series(x,7).truncate()
-1/5040*x^5 + 1/120*x^3 - 1/6*x + 1/x
sage: f.series(x==1,3).truncate().expand()
-2*x^2*cos(1) + 5/2*x^2*sin(1) + 5*x*cos(1) - 7*x*sin(1) - 3*cos(1) + 11/
↪2*sin(1)
```

```
>>> from sage.all import *
>>> f = sin(x)/x**Integer(2)
>>> f.truncate()
sin(x)/x^2
>>> f.series(x,Integer(7))
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
>>> f.series(x,Integer(7)).truncate()
-1/5040*x^5 + 1/120*x^3 - 1/6*x + 1/x
>>> f.series(x==Integer(1),Integer(3)).truncate().expand()
-2*x^2*cos(1) + 5/2*x^2*sin(1) + 5*x*cos(1) - 7*x*sin(1) - 3*cos(1) + 11/
↪2*sin(1)
```

unhold(*exclude=None*)

Evaluates any held operations (with the `hold` keyword) in the expression

INPUT:

- `self` – an expression with held operations
- `exclude` – (default: `None`) a list of operators to exclude from evaluation. Excluding arithmetic operators does not yet work (see Issue #10169).

OUTPUT:

A new expression with held operations, except those in `exclude`, evaluated

EXAMPLES:

```
sage: a = exp(I * pi, hold=True)
sage: a
e^(I*pi)
sage: a.unhold()
-1
sage: b = x.add(x, hold=True)
sage: b
x + x
sage: b.unhold()
2*x
sage: (a + b).unhold()
2*x - 1
sage: c = (x.mul(x, hold=True)).add(x.mul(x, hold=True), hold=True)
sage: c
x*x + x*x
sage: c.unhold()
2*x^2
sage: sin(tan(0, hold=True), hold=True).unhold()
0
sage: sin(tan(0, hold=True), hold=True).unhold(exclude=[sin])
sin(0)
sage: (e^sgn(0, hold=True)).unhold()
1
sage: (e^sgn(0, hold=True)).unhold(exclude=[exp])
e^0
sage: log(3).unhold()
log(3)
```

```
>>> from sage.all import *
>>> a = exp(I * pi, hold=True)
>>> a
e^(I*pi)
>>> a.unhold()
-1
>>> b = x.add(x, hold=True)
>>> b
x + x
>>> b.unhold()
2*x
>>> (a + b).unhold()
2*x - 1
>>> c = (x.mul(x, hold=True)).add(x.mul(x, hold=True), hold=True)
>>> c
```

(continues on next page)

(continued from previous page)

```
x*x + x*x
>>> c.unhold()
2*x^2
>>> sin(tan(Integer(0), hold=True), hold=True).unhold()
0
>>> sin(tan(Integer(0), hold=True), hold=True).unhold(exclude=[sin])
sin(0)
>>> (e**sgn(Integer(0), hold=True)).unhold()
1
>>> (e**sgn(Integer(0), hold=True)).unhold(exclude=[exp])
e^0
>>> log(Integer(3)).unhold()
log(3)
```

unit(*s*)

Return the unit of this expression when considered as a polynomial in `s`.

See also `content()`, `primitive_part()`, and `unit_content_primitive()`.

INPUT:

- `s` – a symbolic expression

OUTPUT:

The unit part of a polynomial as a symbolic expression. It is defined as the sign of the leading coefficient.

EXAMPLES:

```
sage: (2*x+4).unit(x)
1
sage: (-2*x+1).unit(x)
-1
sage: (2*x+1/2).unit(x)
1
sage: var('y')
y
sage: (2*x - 4*sin(y)).unit(sin(y))
-1
```

```
>>> from sage.all import *
>>> (Integer(2)*x+Integer(4)).unit(x)
1
>>> (-Integer(2)*x+Integer(1)).unit(x)
-1
>>> (Integer(2)*x+Integer(1)/Integer(2)).unit(x)
1
>>> var('y')
y
>>> (Integer(2)*x - Integer(4)*sin(y)).unit(sin(y))
-1
```

unit_content_primitive(*s*)

Return the factorization into unit, content, and primitive part.

INPUT:

- `s` – a symbolic expression, usually a symbolic variable. The whole symbolic expression `self` will be considered as a univariate polynomial in `s`.

OUTPUT:

A triple (unit, content, primitive polynomial)` containing the *unit*, *content*, and *primitive polynomial*. Their product equals `self`.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: ex = 9*x^3*y+3*y
sage: ex.unit_content_primitive(x)
(1, 3*y, 3*x^3 + 1)
sage: ex.unit_content_primitive(y)
(1, 9*x^3 + 3, y)
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> ex = Integer(9)*x**Integer(3)*y+Integer(3)*y
>>> ex.unit_content_primitive(x)
(1, 3*y, 3*x^3 + 1)
>>> ex.unit_content_primitive(y)
(1, 9*x^3 + 3, y)
```

variables()

Return sorted tuple of variables that occur in this expression.

EXAMPLES:

```
sage: (x,y,z) = var('x,y,z')
sage: (x+y).variables()
(x, y)
sage: (2*x).variables()
(x,)
sage: (x^y).variables()
(x, y)
sage: sin(x+y^z).variables()
(x, y, z)
```

```
>>> from sage.all import *
>>> (x,y,z) = var('x,y,z')
>>> (x+y).variables()
(x, y)
>>> (Integer(2)*x).variables()
(x,)
>>> (x**y).variables()
(x, y)
>>> sin(x+y**z).variables()
(x, y, z)
```

zeta(*hold=False*)

EXAMPLES:

```
sage: x, y = var('x, y')
sage: (x/y).zeta()
zeta(x/y)
sage: SR(2).zeta()
1/6*pi^2
sage: SR(3).zeta()
zeta(3)
sage: SR(CDF(0,1)).zeta()  # abs tol 1e-16                                    #␣
↪needs sage.libs.pari
0.0033002236853241 - 0.4181554491413217*I
sage: CDF(0,1).zeta()  # abs tol 1e-16                                        #␣
↪needs sage.libs.pari
0.0033002236853241 - 0.4181554491413217*I
sage: plot(lambda x: SR(x).zeta(), -10,10).show(ymin=-3, ymax=3)               #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> x, y = var('x, y')
>>> (x/y).zeta()
zeta(x/y)
>>> SR(Integer(2)).zeta()
1/6*pi^2
>>> SR(Integer(3)).zeta()
zeta(3)
>>> SR(CDF(Integer(0),Integer(1))).zeta()  # abs tol 1e-16                     ␣
↪               # needs sage.libs.pari
0.0033002236853241 - 0.4181554491413217*I
>>> CDF(Integer(0),Integer(1)).zeta()  # abs tol 1e-16                         ␣
↪               # needs sage.libs.pari
0.0033002236853241 - 0.4181554491413217*I
>>> plot(lambda x: SR(x).zeta(), -Integer(10),Integer(10)).show(ymin=-
↪Integer(3), ymax=Integer(3))            # needs sage.plot
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(2).zeta(hold=True)
zeta(2)
```

```
>>> from sage.all import *
>>> SR(Integer(2)).zeta(hold=True)
zeta(2)
```

This also works using functional notation:

```
sage: zeta(2, hold=True)
zeta(2)
sage: zeta(2)
1/6*pi^2
```

```
>>> from sage.all import *
>>> zeta(Integer(2), hold=True)
zeta(2)
```

(continues on next page)

(continued from previous page)

```
>>> zeta(Integer(2))
1/6*pi^2
```

To then evaluate again, we use `unhold()`:

```
sage: a = SR(2).zeta(hold=True); a.unhold()
1/6*pi^2
```

```
>>> from sage.all import *
>>> a = SR(Integer(2)).zeta(hold=True); a.unhold()
1/6*pi^2
```

class sage.symbolic.expression.**ExpressionIterator**

Bases: `object`

class sage.symbolic.expression.**OperandsWrapper**

Bases: `SageObject`

Operands wrapper for symbolic expressions.

EXAMPLES:

```
sage: x,y,z = var('x,y,z')
sage: e = x + x*y + z^y + 3*y*z; e
x*y + 3*y*z + x + z^y
sage: e.op[1]
3*y*z
sage: e.op[1,1]
z
sage: e.op[-1]
z^y
sage: e.op[1:]
[3*y*z, x, z^y]
sage: e.op[:2]
[x*y, 3*y*z]
sage: e.op[-2:]
[x, z^y]
sage: e.op[:-2]
[x*y, 3*y*z]
sage: e.op[-5]
Traceback (most recent call last):
...
IndexError: operand index out of range, got -5, expect between -4 and 3
sage: e.op[5]
Traceback (most recent call last):
...
IndexError: operand index out of range, got 5, expect between -4 and 3
sage: e.op[1,1,0]
Traceback (most recent call last):
...
TypeError: expressions containing only a numeric coefficient, constant or symbol␣
↪have no operands
sage: e.op[:1.5]
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: slice indices must be integers or None or have an __index__ method
sage: e.op[:2:1.5]
Traceback (most recent call last):
...
ValueError: step value must be an integer
```

```
>>> from sage.all import *
>>> x,y,z = var('x,y,z')
>>> e = x + x*y + z**y + Integer(3)*y*z; e
x*y + 3*y*z + x + z^y
>>> e.op[Integer(1)]
3*y*z
>>> e.op[Integer(1),Integer(1)]
z
>>> e.op[-Integer(1)]
z^y
>>> e.op[Integer(1):]
[3*y*z, x, z^y]
>>> e.op[:Integer(2)]
[x*y, 3*y*z]
>>> e.op[-Integer(2):]
[x, z^y]
>>> e.op[:-Integer(2)]
[x*y, 3*y*z]
>>> e.op[-Integer(5)]
Traceback (most recent call last):
...
IndexError: operand index out of range, got -5, expect between -4 and 3
>>> e.op[Integer(5)]
Traceback (most recent call last):
...
IndexError: operand index out of range, got 5, expect between -4 and 3
>>> e.op[Integer(1),Integer(1),Integer(0)]
Traceback (most recent call last):
...
TypeError: expressions containing only a numeric coefficient, constant or symbol␣
↪have no operands
>>> e.op[:RealNumber('1.5')]
Traceback (most recent call last):
...
TypeError: slice indices must be integers or None or have an __index__ method
>>> e.op[:Integer(2):RealNumber('1.5')]
Traceback (most recent call last):
...
ValueError: step value must be an integer
```

class sage.symbolic.expression.**PynacConstant**

Bases: object

expression()

Return this constant as an Expression.

EXAMPLES:

```
sage: from sage.symbolic.expression import PynacConstant
sage: f = PynacConstant('foo', 'foo', 'real')
sage: f + 2
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: '<class 'sage.symbolic.
↪expression.PynacConstant'>' and 'Integer Ring'

sage: foo = f.expression(); foo
foo
sage: foo + 2
foo + 2
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import PynacConstant
>>> f = PynacConstant('foo', 'foo', 'real')
>>> f + Integer(2)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: '<class 'sage.symbolic.
↪expression.PynacConstant'>' and 'Integer Ring'

>>> foo = f.expression(); foo
foo
>>> foo + Integer(2)
foo + 2
```

name()

Return the name of this constant.

EXAMPLES:

```
sage: from sage.symbolic.expression import PynacConstant
sage: f = PynacConstant('foo', 'foo', 'real')
sage: f.name()
'foo'
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import PynacConstant
>>> f = PynacConstant('foo', 'foo', 'real')
>>> f.name()
'foo'
```

serial()

Return the underlying Pynac serial for this constant.

EXAMPLES:

```
sage: from sage.symbolic.expression import PynacConstant
sage: f = PynacConstant('foo', 'foo', 'real')
sage: f.serial()  #random
15
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import PynacConstant
>>> f = PynacConstant('foo', 'foo', 'real')
>>> f.serial()  #random
15
```

class sage.symbolic.expression.**SubstitutionMap**

Bases: SageObject

apply_to(*expr*, *options*)

Apply the substitution to a symbolic expression.

EXAMPLES:

```
sage: from sage.symbolic.expression import make_map
sage: subs = make_map({x:x+1})
sage: subs.apply_to(x^2, 0)
(x + 1)^2
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import make_map
>>> subs = make_map({x:x+Integer(1)})
>>> subs.apply_to(x**Integer(2), Integer(0))
(x + 1)^2
```

class sage.symbolic.expression.**SymbolicSeries**

Bases: *Expression*

Trivial constructor.

EXAMPLES:

```
sage: loads(dumps((x+x^3).series(x,2)))
1*x + Order(x^2)
```

```
>>> from sage.all import *
>>> loads(dumps((x+x**Integer(3)).series(x,Integer(2))))
1*x + Order(x^2)
```

coefficients(*x=None*, *sparse=True*)

Return the coefficients of this symbolic series as a list of pairs.

INPUT:

- `x` – (optional) variable
- `sparse` – boolean (default: `True`); if `False` return a list with as much entries as the order of the series

OUTPUT: depending on the value of `sparse`,

- A list of pairs `(expr, n)`, where `expr` is a symbolic expression and `n` is a power (`sparse=True`, default)
- A list of expressions where the `n`-th element is the coefficient of `x^n` when `self` is seen as polynomial in `x` (`sparse=False`).

EXAMPLES:

```
sage: s = (1/(1-x)).series(x,6); s
1 + 1*x + 1*x^2 + 1*x^3 + 1*x^4 + 1*x^5 + Order(x^6)
sage: s.coefficients()
[[1, 0], [1, 1], [1, 2], [1, 3], [1, 4], [1, 5]]
sage: s.coefficients(x, sparse=False)
[1, 1, 1, 1, 1, 1]
sage: x,y = var("x,y")
sage: s = (1/(1-y*x-x)).series(x,3); s
1 + (y + 1)*x + ((y + 1)^2)*x^2 + Order(x^3)
sage: s.coefficients(x, sparse=False)
[1, y + 1, (y + 1)^2]
```

```
>>> from sage.all import *
>>> s = (Integer(1)/(Integer(1)-x)).series(x,Integer(6)); s
1 + 1*x + 1*x^2 + 1*x^3 + 1*x^4 + 1*x^5 + Order(x^6)
>>> s.coefficients()
[[1, 0], [1, 1], [1, 2], [1, 3], [1, 4], [1, 5]]
>>> s.coefficients(x, sparse=False)
[1, 1, 1, 1, 1, 1]
>>> x,y = var("x,y")
>>> s = (Integer(1)/(Integer(1)-y*x-x)).series(x,Integer(3)); s
1 + (y + 1)*x + ((y + 1)^2)*x^2 + Order(x^3)
>>> s.coefficients(x, sparse=False)
[1, y + 1, (y + 1)^2]
```

default_variable()

Return the expansion variable of this symbolic series.

EXAMPLES:

```
sage: s = (1/(1-x)).series(x,3); s
1 + 1*x + 1*x^2 + Order(x^3)
sage: s.default_variable()
x
```

```
>>> from sage.all import *
>>> s = (Integer(1)/(Integer(1)-x)).series(x,Integer(3)); s
1 + 1*x + 1*x^2 + Order(x^3)
>>> s.default_variable()
x
```

is_terminating_series()

Return `True` if the series is without order term.

A series is terminating if it can be represented exactly, without requiring an order term. You can explicitly request terminating series by setting the order to positive infinity.

OUTPUT: boolean; `True` if the series has no order term

EXAMPLES:

```
sage: (x^5+x^2+1).series(x, +oo)
1 + 1*x^2 + 1*x^5
sage: (x^5+x^2+1).series(x,+oo).is_terminating_series()
```

(continues on next page)

(continued from previous page)

```
True
sage: SR(5).is_terminating_series()
False
sage: exp(x).series(x,10).is_terminating_series()
False
```

```
>>> from sage.all import *
>>> (x**Integer(5)+x**Integer(2)+Integer(1)).series(x, +oo)
1 + 1*x^2 + 1*x^5
>>> (x**Integer(5)+x**Integer(2)+Integer(1)).series(x,+oo).is_terminating_
↪series()
True
>>> SR(Integer(5)).is_terminating_series()
False
>>> exp(x).series(x,Integer(10)).is_terminating_series()
False
```

power_series(*base_ring*)

Return the algebraic power series associated to this symbolic series.

The coefficients must be coercible to the base ring.

EXAMPLES:

```
sage: ex = (gamma(1-x)).series(x,3); ex
1 + euler_gamma*x + (1/2*euler_gamma^2 + 1/12*pi^2)*x^2 + Order(x^3)
sage: g = ex.power_series(SR); g
1 + euler_gamma*x + (1/2*euler_gamma^2 + 1/12*pi^2)*x^2 + O(x^3)
sage: g.parent()
Power Series Ring in x over Symbolic Ring
```

```
>>> from sage.all import *
>>> ex = (gamma(Integer(1)-x)).series(x,Integer(3)); ex
1 + euler_gamma*x + (1/2*euler_gamma^2 + 1/12*pi^2)*x^2 + Order(x^3)
>>> g = ex.power_series(SR); g
1 + euler_gamma*x + (1/2*euler_gamma^2 + 1/12*pi^2)*x^2 + O(x^3)
>>> g.parent()
Power Series Ring in x over Symbolic Ring
```

truncate()

Given a power series or expression, return the corresponding expression without the big oh.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: f = sin(x)/x^2
sage: f.truncate()
sin(x)/x^2
sage: f.series(x,7)
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
sage: f.series(x,7).truncate()
-1/5040*x^5 + 1/120*x^3 - 1/6*x + 1/x
```

(continues on next page)

(continued from previous page)

```
sage: f.series(x==1,3).truncate().expand()
-2*x^2*cos(1) + 5/2*x^2*sin(1) + 5*x*cos(1) - 7*x*sin(1) - 3*cos(1) + 11/
↪2*sin(1)
```

```
>>> from sage.all import *
>>> f = sin(x)/x**Integer(2)
>>> f.truncate()
sin(x)/x^2
>>> f.series(x,Integer(7))
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
>>> f.series(x,Integer(7)).truncate()
-1/5040*x^5 + 1/120*x^3 - 1/6*x + 1/x
>>> f.series(x==Integer(1),Integer(3)).truncate().expand()
-2*x^2*cos(1) + 5/2*x^2*sin(1) + 5*x*cos(1) - 7*x*sin(1) - 3*cos(1) + 11/
↪2*sin(1)
```

`sage.symbolic.expression.`**`call_registered_function`**(*serial*, *nargs*, *args*, *hold*, *allow_numeric_result*, *result_parent*)

Call a function registered with Pynac (GiNaC).

INPUT:

- `serial` – serial number of the function
- `nargs` – declared number of args (0 is variadic)
- `args` – list of arguments to pass to the function; each must be an *`Expression`*
- `hold` – whether to leave the call unevaluated
- `allow_numeric_result` – if `True`, keep numeric results numeric; if `False`, make all results symbolic expressions
- `result_parent` – an instance of `SymbolicRing`

EXAMPLES:

```
sage: from sage.symbolic.expression import find_registered_function, call_
↪registered_function
sage: s_arctan = find_registered_function('arctan', 1)
sage: call_registered_function(s_arctan, 1, [SR(1)], False, True, SR)
1/4*pi
sage: call_registered_function(s_arctan, 1, [SR(1)], True, True, SR)
arctan(1)
sage: call_registered_function(s_arctan, 1, [SR(0)], False, True, SR)
0
sage: call_registered_function(s_arctan, 1, [SR(0)], False, True, SR).parent()
Integer Ring
sage: call_registered_function(s_arctan, 1, [SR(0)], False, False, SR).parent()
Symbolic Ring
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import find_registered_function, call_
↪registered_function
>>> s_arctan = find_registered_function('arctan', Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> call_registered_function(s_arctan, Integer(1), [SR(Integer(1))], False, True,␣
↪SR)
1/4*pi
>>> call_registered_function(s_arctan, Integer(1), [SR(Integer(1))], True, True,␣
↪SR)
arctan(1)
>>> call_registered_function(s_arctan, Integer(1), [SR(Integer(0))], False, True,␣
↪SR)
0
>>> call_registered_function(s_arctan, Integer(1), [SR(Integer(0))], False, True,␣
↪SR).parent()
Integer Ring
>>> call_registered_function(s_arctan, Integer(1), [SR(Integer(0))], False, False,
↪ SR).parent()
Symbolic Ring
```

sage.symbolic.expression.**doublefactorial**(*n*)

The double factorial combinatorial function:

n!! == n * (n-2) * (n-4) * … * ({1|2}) with 0!! == (-1)!! == 1.

INPUT:

- `n` – integer `>= 1`

EXAMPLES:

```
sage: from sage.symbolic.expression import doublefactorial
sage: doublefactorial(-1)
1
sage: doublefactorial(0)
1
sage: doublefactorial(1)
1
sage: doublefactorial(5)
15
sage: doublefactorial(20)
3715891200
sage: prod( [20,18,..,2] )
3715891200
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import doublefactorial
>>> doublefactorial(-Integer(1))
1
>>> doublefactorial(Integer(0))
1
>>> doublefactorial(Integer(1))
1
>>> doublefactorial(Integer(5))
15
>>> doublefactorial(Integer(20))
3715891200
```

(continues on next page)

(continued from previous page)

```
>>> prod( (ellipsis_range(Integer(20),Integer(18),Ellipsis,Integer(2))) )
3715891200
```

sage.symbolic.expression.**find_registered_function**(*name*, *nargs*)

Look up a function registered with Pynac (GiNaC).

Raise a `ValueError` if the function is not registered.

OUTPUT: serial number of the function, for use in *call_registered_function()*

EXAMPLES:

```
sage: from sage.symbolic.expression import find_registered_function
sage: find_registered_function('arctan', 1)  # random
19
sage: find_registered_function('archenemy', 1)
Traceback (most recent call last):
...
ValueError: cannot find GiNaC function with name archenemy and 1 arguments
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import find_registered_function
>>> find_registered_function('arctan', Integer(1))  # random
19
>>> find_registered_function('archenemy', Integer(1))
Traceback (most recent call last):
...
ValueError: cannot find GiNaC function with name archenemy and 1 arguments
```

sage.symbolic.expression.**get_fn_serial**()

Return the overall size of the Pynac function registry which corresponds to the last serial value plus one.

EXAMPLES:

```
sage: from sage.symbolic.expression import get_fn_serial
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: get_fn_serial() > 125
True
sage: print(get_sfunction_from_serial(get_fn_serial()))
None
sage: get_sfunction_from_serial(get_fn_serial() - 1) is not None
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import get_fn_serial
>>> from sage.symbolic.function import get_sfunction_from_serial
>>> get_fn_serial() > Integer(125)
True
>>> print(get_sfunction_from_serial(get_fn_serial()))
None
>>> get_sfunction_from_serial(get_fn_serial() - Integer(1)) is not None
True
```

sage.symbolic.expression.**get_ginac_serial**()

Number of C++ level functions defined by GiNaC. (Defined mainly for testing.)

EXAMPLES:

```
sage: sage.symbolic.expression.get_ginac_serial() >= 35
True
```

```
>>> from sage.all import *
>>> sage.symbolic.expression.get_ginac_serial() >= Integer(35)
True
```

sage.symbolic.expression.**get_sfunction_from_hash**(*myhash*)

Return an already created `SymbolicFunction` given the hash.

EXAMPLES:

```
sage: from sage.symbolic.expression import get_sfunction_from_hash
sage: get_sfunction_from_hash(1)  # random
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import get_sfunction_from_hash
>>> get_sfunction_from_hash(Integer(1))  # random
```

sage.symbolic.expression.**get_sfunction_from_serial**(*serial*)

Return an already created `SymbolicFunction` given the serial.

These are stored in the dictionary `sfunction_serial_dict`.

EXAMPLES:

```
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: get_sfunction_from_serial(65) #random
f
```

```
>>> from sage.all import *
>>> from sage.symbolic.function import get_sfunction_from_serial
>>> get_sfunction_from_serial(Integer(65)) #random
f
```

class sage.symbolic.expression.**hold_class**

Bases: `object`

Instances of this class can be used with Python *with*.

EXAMPLES:

```
sage: with hold:
....:     tan(1/12*pi)
....:
tan(1/12*pi)
sage: tan(1/12*pi)
-sqrt(3) + 2
sage: with hold:
....:     2^5
```

(continues on next page)

(continued from previous page)

```
....:
32
sage: with hold:
....:     SR(2)^5
....:
2^5
sage: with hold:
....:     t=tan(1/12*pi)
....:
sage: t
tan(1/12*pi)
sage: t.unhold()
-sqrt(3) + 2
```

```
>>> from sage.all import *
>>> with hold:
...     tan(Integer(1)/Integer(12)*pi)
....:
tan(1/12*pi)
>>> tan(Integer(1)/Integer(12)*pi)
-sqrt(3) + 2
>>> with hold:
...     Integer(2)**Integer(5)
....:
32
>>> with hold:
...     SR(Integer(2))**Integer(5)
....:
2^5
>>> with hold:
...     t=tan(Integer(1)/Integer(12)*pi)
....:
>>> t
tan(1/12*pi)
>>> t.unhold()
-sqrt(3) + 2
```

start()

Start a hold context.

EXAMPLES:

```
sage: hold.start()
sage: SR(2)^5
2^5
sage: hold.stop()
sage: SR(2)^5
32
```

```
>>> from sage.all import *
>>> hold.start()
>>> SR(Integer(2))**Integer(5)
```

(continues on next page)

(continued from previous page)

```
2^5
>>> hold.stop()
>>> SR(Integer(2))**Integer(5)
32
```

stop()

Stop any hold context.

EXAMPLES:

```
sage: hold.start()
sage: SR(2)^5
2^5
sage: hold.stop()
sage: SR(2)^5
32
```

```
>>> from sage.all import *
>>> hold.start()
>>> SR(Integer(2))**Integer(5)
2^5
>>> hold.stop()
>>> SR(Integer(2))**Integer(5)
32
```

sage.symbolic.expression.**init_function_table**()

Initialize the function pointer table in Pynac. This must be called before Pynac is used; otherwise, there will be segfaults.

sage.symbolic.expression.**init_pynac_I**()

Initialize the numeric `I` object in pynac. We use the generator of `QQ(i)`.

EXAMPLES:

```
sage: from sage.symbolic.constants import I as symbolic_I
sage: symbolic_I
I
sage: symbolic_I^2
-1
```

```
>>> from sage.all import *
>>> from sage.symbolic.constants import I as symbolic_I
>>> symbolic_I
I
>>> symbolic_I**Integer(2)
-1
```

Note that conversions to real fields will give `TypeError`:

```
sage: float(symbolic_I)
Traceback (most recent call last):
...
TypeError: unable to simplify to float approximation
```

(continues on next page)

(continued from previous page)

```
sage: gp(symbolic_I)
I
sage: RR(symbolic_I)
Traceback (most recent call last):
...
TypeError: unable to convert '1.00000000000000*I' to a real number
```

```
>>> from sage.all import *
>>> float(symbolic_I)
Traceback (most recent call last):
...
TypeError: unable to simplify to float approximation
>>> gp(symbolic_I)
I
>>> RR(symbolic_I)
Traceback (most recent call last):
...
TypeError: unable to convert '1.00000000000000*I' to a real number
```

We can convert to complex fields:

```
sage: C = ComplexField(200); C
Complex Field with 200 bits of precision
sage: C(symbolic_I)
1.0000000000000000000000000000000000000000000000000000000000*I
sage: symbolic_I._complex_mpfr_field_(ComplexField(53))
1.00000000000000*I

sage: symbolic_I._complex_double_(CDF)
1.0*I
sage: CDF(symbolic_I)
1.0*I

sage: z = symbolic_I + symbolic_I; z
2*I
sage: C(z)
2.0000000000000000000000000000000000000000000000000000000000*I
sage: 1e8*symbolic_I
1.00000000000000e8*I

sage: complex(symbolic_I)
1j

sage: QQbar(symbolic_I)
I

sage: abs(symbolic_I)
1

sage: symbolic_I.minpoly()
x^2 + 1
sage: maxima(2*symbolic_I)
```

(continues on next page)

(continued from previous page)

```
2*%i
```

```
>>> from sage.all import *
>>> C = ComplexField(Integer(200)); C
Complex Field with 200 bits of precision
>>> C(symbolic_I)
1.0000000000000000000000000000000000000000000000000000000000*I
>>> symbolic_I._complex_mpfr_field_(ComplexField(Integer(53)))
1.00000000000000*I

>>> symbolic_I._complex_double_(CDF)
1.0*I
>>> CDF(symbolic_I)
1.0*I

>>> z = symbolic_I + symbolic_I; z
2*I
>>> C(z)
2.0000000000000000000000000000000000000000000000000000000000*I
>>> RealNumber('1e8')*symbolic_I
1.00000000000000e8*I

>>> complex(symbolic_I)
1j

>>> QQbar(symbolic_I)
I

>>> abs(symbolic_I)
1

>>> symbolic_I.minpoly()
x^2 + 1
>>> maxima(Integer(2)*symbolic_I)
2*%i
```

sage.symbolic.expression.**is_SymbolicEquation**(*x*)

Return `True` if `x` is a symbolic equation.

This function is deprecated.

EXAMPLES:

The following two examples are symbolic equations:

```
sage: from sage.symbolic.expression import is_SymbolicEquation
sage: is_SymbolicEquation(sin(x) == x)
doctest:warning...
DeprecationWarning: is_SymbolicEquation is deprecated; use
'isinstance(x, sage.structure.element.Expression) and x.is_relational()' instead
See https://github.com/sagemath/sage/issues/35505 for details.
True
sage: is_SymbolicEquation(sin(x) < x)
```

(continues on next page)

(continued from previous page)

```
True
sage: is_SymbolicEquation(x)
False
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import is_SymbolicEquation
>>> is_SymbolicEquation(sin(x) == x)
doctest:warning...
DeprecationWarning: is_SymbolicEquation is deprecated; use
'isinstance(x, sage.structure.element.Expression) and x.is_relational()' instead
See https://github.com/sagemath/sage/issues/35505 for details.
True
>>> is_SymbolicEquation(sin(x) < x)
True
>>> is_SymbolicEquation(x)
False
```

This is not, since `2==3` evaluates to the boolean `False`:

```
sage: is_SymbolicEquation(2 == 3)
False
```

```
>>> from sage.all import *
>>> is_SymbolicEquation(Integer(2) == Integer(3))
False
```

However here since both 2 and 3 are coerced to be symbolic, we obtain a symbolic equation:

```
sage: is_SymbolicEquation(SR(2) == SR(3))
True
```

```
>>> from sage.all import *
>>> is_SymbolicEquation(SR(Integer(2)) == SR(Integer(3)))
True
```

`sage.symbolic.expression.`**`make_map`**`(`*subs_dict*`)`

Construct a new substitution map.

OUTPUT: a new `SubstitutionMap` for doctesting

EXAMPLES:

```
sage: from sage.symbolic.expression import make_map
sage: make_map({x:x+1})
SubsMap
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import make_map
>>> make_map({x:x+Integer(1)})
SubsMap
```

`sage.symbolic.expression.`**`math_sorted`**`(`*expressions*`)`

Sort a list of symbolic numbers in the “Mathematics” order.

INPUT:

- `expressions` – list/tuple/iterable of symbolic expressions, or something that can be converted to one

OUTPUT:

The list sorted by ascending (real) value. If an entry does not define a real value (or plus/minus infinity), or if the comparison is not known, a `ValueError` is raised.

EXAMPLES:

```
sage: from sage.symbolic.expression import math_sorted
sage: math_sorted([SR(1), SR(e), SR(pi), sqrt(2)])
[1, sqrt(2), e, pi]
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import math_sorted
>>> math_sorted([SR(Integer(1)), SR(e), SR(pi), sqrt(Integer(2))])
[1, sqrt(2), e, pi]
```

`sage.symbolic.expression.`**`mixed_order`**(*lhs*, *rhs*)

Comparison in the mixed order

INPUT:

- `lhs`, `rhs` – two symbolic expressions or something that can be converted to one

OUTPUT:

Either -1, 0, or $+1$ indicating the comparison. An exception is raised if the arguments cannot be converted into the symbolic ring.

EXAMPLES:

```
sage: from sage.symbolic.expression import mixed_order
sage: mixed_order(1, oo)
-1
sage: mixed_order(e, oo)
-1
sage: mixed_order(pi, oo)
-1
sage: mixed_order(1, sqrt(2))
-1
sage: mixed_order(x + x^2, x*(x+1))
-1
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import mixed_order
>>> mixed_order(Integer(1), oo)
-1
>>> mixed_order(e, oo)
-1
>>> mixed_order(pi, oo)
-1
>>> mixed_order(Integer(1), sqrt(Integer(2)))
-1
>>> mixed_order(x + x**Integer(2), x*(x+Integer(1)))
-1
```

Check that Issue #12967 is fixed:

```
sage: mixed_order(SR(oo), sqrt(2))
1
```

```
>>> from sage.all import *
>>> mixed_order(SR(oo), sqrt(Integer(2)))
1
```

Ensure that Issue #32185 is fixed:

```
sage: mixed_order(pi, 0)
1
sage: mixed_order(golden_ratio, 0)
1
sage: mixed_order(log2, 0)
1
```

```
>>> from sage.all import *
>>> mixed_order(pi, Integer(0))
1
>>> mixed_order(golden_ratio, Integer(0))
1
>>> mixed_order(log2, Integer(0))
1
```

sage.symbolic.expression.**mixed_sorted**(*expressions*)

Sort a list of symbolic numbers in the "Mixed" order.

INPUT:

- `expressions` – list/tuple/iterable of symbolic expressions, or something that can be converted to one

OUTPUT:

In the list the numeric values are sorted by ascending (real) value, and the expressions with variables according to print order. If an entry does not define a real value (or plus/minus infinity), or if the comparison is not known, a `ValueError` is raised.

EXAMPLES:

```
sage: from sage.symbolic.expression import mixed_sorted
sage: mixed_sorted([SR(1), SR(e), SR(pi), sqrt(2), x, sqrt(x), sin(1/x)])
[1, sqrt(2), e, pi, sin(1/x), sqrt(x), x]
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import mixed_sorted
>>> mixed_sorted([SR(Integer(1)), SR(e), SR(pi), sqrt(Integer(2)), x, sqrt(x),␣
↪sin(Integer(1)/x)])
[1, sqrt(2), e, pi, sin(1/x), sqrt(x), x]
```

sage.symbolic.expression.**new_Expression**(*parent*, *x*)

Convert `x` into the symbolic expression ring `parent`.

This is the element constructor.

EXAMPLES:

```
sage: a = SR(-3/4); a
-3/4
sage: type(a)
<class 'sage.symbolic.expression.Expression'>
sage: a.parent()
Symbolic Ring
sage: K.<a> = QuadraticField(-3)                                          #␣
↪needs sage.rings.number_field
sage: a + sin(x)                                                          #␣
↪needs sage.rings.number_field
I*sqrt(3) + sin(x)
sage: x = var('x'); y0,y1 = PolynomialRing(ZZ,2,'y').gens()
sage: x+y0/y1
x + y0/y1
sage: x.subs(x=y0/y1)
y0/y1
sage: x + int(1)
x + 1
```

```
>>> from sage.all import *
>>> a = SR(-Integer(3)/Integer(4)); a
-3/4
>>> type(a)
<class 'sage.symbolic.expression.Expression'>
>>> a.parent()
Symbolic Ring
>>> K = QuadraticField(-Integer(3), names=('a',)); (a,) = K._first_ngens(1)#␣
↪needs sage.rings.number_field
>>> a + sin(x)                                                          #␣
↪needs sage.rings.number_field
I*sqrt(3) + sin(x)
>>> x = var('x'); y0,y1 = PolynomialRing(ZZ,Integer(2),'y').gens()
>>> x+y0/y1
x + y0/y1
>>> x.subs(x=y0/y1)
y0/y1
>>> x + int(Integer(1))
x + 1
```

sage.symbolic.expression.**new_Expression_from_pyobject**(*parent*, *x*, *force=True*, *recursive=True*)

Wrap the given Python object in a symbolic expression even if it cannot be coerced to the Symbolic Ring.

INPUT:

- `parent` – a symbolic ring
- `x` – a Python object
- `force` – boolean (default: `True`); if `True`, the Python object is taken as is without attempting coercion or list traversal
- `recursive` – boolean (default: `True`); disables recursive traversal of lists

EXAMPLES:

```
sage: t = SR._force_pyobject(QQ); t    # indirect doctest
Rational Field
sage: type(t)
<class 'sage.symbolic.expression.Expression'>

sage: from sage.symbolic.expression import new_Expression_from_pyobject
sage: t = new_Expression_from_pyobject(SR, 17); t
17
sage: type(t)
<class 'sage.symbolic.expression.Expression'>

sage: t2 = new_Expression_from_pyobject(SR, t, False); t2
17
sage: t2 is t
True

sage: tt = new_Expression_from_pyobject(SR, t, True); tt
17
sage: tt is t
False
```

```
>>> from sage.all import *
>>> t = SR._force_pyobject(QQ); t    # indirect doctest
Rational Field
>>> type(t)
<class 'sage.symbolic.expression.Expression'>

>>> from sage.symbolic.expression import new_Expression_from_pyobject
>>> t = new_Expression_from_pyobject(SR, Integer(17)); t
17
>>> type(t)
<class 'sage.symbolic.expression.Expression'>

>>> t2 = new_Expression_from_pyobject(SR, t, False); t2
17
>>> t2 is t
True

>>> tt = new_Expression_from_pyobject(SR, t, True); tt
17
>>> tt is t
False
```

sage.symbolic.expression.**new_Expression_symbol**(*parent*, *name=None*, *latex_name=None*, *domain=None*)

Look up or create a symbol.

EXAMPLES:

```
sage: t0 = SR.symbol("t0")
sage: t0.conjugate()
conjugate(t0)
```

(continues on next page)

(continued from previous page)

```
sage: t1 = SR.symbol("t1", domain='real')
sage: t1.conjugate()
t1

sage: t0.abs()
abs(t0)

sage: t0_2 = SR.symbol("t0", domain='positive')
sage: t0_2.abs()
t0
sage: bool(t0_2 == t0)
True
sage: t0.conjugate()
t0

sage: SR.symbol() # temporary variable
symbol...
```

```
>>> from sage.all import *
>>> t0 = SR.symbol("t0")
>>> t0.conjugate()
conjugate(t0)

>>> t1 = SR.symbol("t1", domain='real')
>>> t1.conjugate()
t1

>>> t0.abs()
abs(t0)

>>> t0_2 = SR.symbol("t0", domain='positive')
>>> t0_2.abs()
t0
>>> bool(t0_2 == t0)
True
>>> t0.conjugate()
t0

>>> SR.symbol() # temporary variable
symbol...
```

sage.symbolic.expression.**new_Expression_wild**(*parent*, *n=0*)

Return the n-th wild-card for pattern matching and substitution.

INPUT:

- `parent` – a symbolic ring
- `n` – nonnegative integer

OUTPUT: n-th wildcard expression

EXAMPLES:

```
sage: x,y = var('x,y')
sage: w0 = SR.wild(0); w1 = SR.wild(1)
sage: pattern = sin(x)*w0*w1^2; pattern
$1^2*$0*sin(x)
sage: f = atan(sin(x)*3*x^2); f
arctan(3*x^2*sin(x))
sage: f.has(pattern)
True
sage: f.subs(pattern == x^2)
arctan(x^2)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> w0 = SR.wild(Integer(0)); w1 = SR.wild(Integer(1))
>>> pattern = sin(x)*w0*w1**Integer(2); pattern
$1^2*$0*sin(x)
>>> f = atan(sin(x)*Integer(3)*x**Integer(2)); f
arctan(3*x^2*sin(x))
>>> f.has(pattern)
True
>>> f.subs(pattern == x**Integer(2))
arctan(x^2)
```

sage.symbolic.expression.**normalize_index_for_doctests**(*arg*, *nops*)

Wrapper function to test `normalize_index`.

sage.symbolic.expression.**paramset_from_Expression**(*e*)

EXAMPLES:

```
sage: from sage.symbolic.expression import paramset_from_Expression
sage: f = function('f')
sage: paramset_from_Expression(f(x).diff(x))
[0]
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import paramset_from_Expression
>>> f = function('f')
>>> paramset_from_Expression(f(x).diff(x))
[0]
```

sage.symbolic.expression.**print_order**(*lhs*, *rhs*)

Comparison in the print order

INPUT:

- `lhs`, `rhs` – two symbolic expressions or something that can be converted to one

OUTPUT:

Either -1, 0, or $+1$ indicating the comparison. An exception is raised if the arguments cannot be converted into the symbolic ring.

EXAMPLES:

```
sage: from sage.symbolic.expression import print_order
sage: print_order(1, oo)
1
sage: print_order(e, oo)
-1
sage: print_order(pi, oo)
1
sage: print_order(1, sqrt(2))
1
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import print_order
>>> print_order(Integer(1), oo)
1
>>> print_order(e, oo)
-1
>>> print_order(pi, oo)
1
>>> print_order(Integer(1), sqrt(Integer(2)))
1
```

Check that Issue #12967 is fixed:

```
sage: print_order(SR(oo), sqrt(2))
1
```

```
>>> from sage.all import *
>>> print_order(SR(oo), sqrt(Integer(2)))
1
```

sage.symbolic.expression.**print_sorted**(*expressions*)

Sort a list in print order.

INPUT:

- `expressions` – list/tuple/iterable of symbolic expressions, or something that can be converted to one

OUTPUT: the list sorted by `print_order()`

EXAMPLES:

```
sage: from sage.symbolic.expression import print_sorted
sage: print_sorted([SR(1), SR(e), SR(pi), sqrt(2)])
[e, sqrt(2), pi, 1]
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import print_sorted
>>> print_sorted([SR(Integer(1)), SR(e), SR(pi), sqrt(Integer(2))])
[e, sqrt(2), pi, 1]
```

sage.symbolic.expression.**py_atan2_for_doctests**(*x*, *y*)

Wrapper function to test py_atan2.

sage.symbolic.expression.**py_denom_for_doctests**(*n*)

This function is used to test py_denom().

EXAMPLES:

```
sage: from sage.symbolic.expression import py_denom_for_doctests
sage: py_denom_for_doctests(2/3)
3
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_denom_for_doctests
>>> py_denom_for_doctests(Integer(2)/Integer(3))
3
```

sage.symbolic.expression.**py_eval_infinity_for_doctests**()

This function tests py_eval_infinity.

sage.symbolic.expression.**py_eval_neg_infinity_for_doctests**()

This function tests py_eval_neg_infinity.

sage.symbolic.expression.**py_eval_unsigned_infinity_for_doctests**()

This function tests py_eval_unsigned_infinity.

sage.symbolic.expression.**py_exp_for_doctests**(*x*)

This function tests py_exp.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_exp_for_doctests
sage: py_exp_for_doctests(CC(2))
7.38905609893065
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_exp_for_doctests
>>> py_exp_for_doctests(CC(Integer(2)))
7.38905609893065
```

sage.symbolic.expression.**py_factorial_py**(*x*)

This function is a python wrapper around py_factorial(). This wrapper is needed when we override the eval() method for GiNaC's factorial function in sage.functions.other.Function_factorial.

sage.symbolic.expression.**py_float_for_doctests**(*n*, *kwds*)

This function is for testing py_float.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_float_for_doctests
sage: py_float_for_doctests(pi, {'parent':RealField(80)})
3.1415926535897932384626
sage: py_float_for_doctests(I, {'parent':RealField(80)})
1.0000000000000000000000*I
sage: py_float_for_doctests(I, {'parent':float})
1j
sage: py_float_for_doctests(pi, {'parent':complex})
(3.141592653589793+0j)
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_float_for_doctests
>>> py_float_for_doctests(pi, {'parent':RealField(Integer(80))})
3.1415926535897932384626
>>> py_float_for_doctests(I, {'parent':RealField(Integer(80))})
1.0000000000000000000000*I
>>> py_float_for_doctests(I, {'parent':float})
1j
>>> py_float_for_doctests(pi, {'parent':complex})
(3.141592653589793+0j)
```

sage.symbolic.expression.**py_imag_for_doctests**(*x*)

Used for doctesting py_imag.

sage.symbolic.expression.**py_is_cinteger_for_doctest**(*x*)

Return `True` if pynac should treat this object as an element of $\mathbf{Z}(i)$.

sage.symbolic.expression.**py_is_crational_for_doctest**(*x*)

Return `True` if pynac should treat this object as an element of $\mathbf{Q}(i)$.

sage.symbolic.expression.**py_is_integer_for_doctests**(*x*)

Used internally for doctesting purposes.

sage.symbolic.expression.**py_latex_fderivative_for_doctests**(*id*, *params*, *args*)

Used internally for writing doctests for certain cdef'd functions.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_latex_fderivative_for_doctests as␣
↪py_latex_fderivative, get_ginac_serial, get_fn_serial

sage: var('x,y,z')
(x, y, z)
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: foo = function('foo', nargs=2)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_fderivative(i, (0, 1, 0, 1), (x, y^z))
\mathrm{D}_{0, 1, 0, 1}\left({\rm foo}\right)\left(x, y^{z}\right)
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_latex_fderivative_for_doctests as py_
↪latex_fderivative, get_ginac_serial, get_fn_serial

>>> var('x,y,z')
(x, y, z)
>>> from sage.symbolic.function import get_sfunction_from_serial
>>> foo = function('foo', nargs=Integer(2))
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...   if get_sfunction_from_serial(i) == foo: break
```

(continues on next page)

(continued from previous page)

```
>>> get_sfunction_from_serial(i) == foo
True
>>> py_latex_fderivative(i, (Integer(0), Integer(1), Integer(0), Integer(1)), (x,␣
↪y**z))
\mathrm{D}_{0, 1, 0, 1}\left({\rm foo}\right)\left(x, y^{z}\right)
```

Test latex_name:

```
sage: foo = function('foo', nargs=2, latex_name=r'\mathrm{bar}')
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_fderivative(i, (0, 1, 0, 1), (x, y^z))
\mathrm{D}_{0, 1, 0, 1}\left(\mathrm{bar}\right)\left(x, y^{z}\right)
```

```
>>> from sage.all import *
>>> foo = function('foo', nargs=Integer(2), latex_name=r'\mathrm{bar}')
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...   if get_sfunction_from_serial(i) == foo: break

>>> get_sfunction_from_serial(i) == foo
True
>>> py_latex_fderivative(i, (Integer(0), Integer(1), Integer(0), Integer(1)), (x,␣
↪y**z))
\mathrm{D}_{0, 1, 0, 1}\left(\mathrm{bar}\right)\left(x, y^{z}\right)
```

Test custom func:

```
sage: def my_print(self, *args): return "func_with_args(" + ', '.join(map(repr,␣
↪args)) +')'
sage: foo = function('foo', nargs=2, print_latex_func=my_print)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_fderivative(i, (0, 1, 0, 1), (x, y^z))
\mathrm{D}_{0, 1, 0, 1}func_with_args(x, y^z)
```

```
>>> from sage.all import *
>>> def my_print(self, *args): return "func_with_args(" + ', '.join(map(repr,␣
↪args)) +')'
>>> foo = function('foo', nargs=Integer(2), print_latex_func=my_print)
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...   if get_sfunction_from_serial(i) == foo: break

>>> get_sfunction_from_serial(i) == foo
True
>>> py_latex_fderivative(i, (Integer(0), Integer(1), Integer(0), Integer(1)), (x,␣
↪y**z))
```

(continues on next page)

(continued from previous page)

```
\mathrm{D}_{0, 1, 0, 1}func_with_args(x, y^z)
```

sage.symbolic.expression.**py_latex_function_pystring**(*id*, *args*, *fname_paren=False*)

Return a string with the latex representation of the symbolic function specified by the given id applied to args.

See documentation of py_print_function_pystring for more information.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_latex_function_pystring, get_ginac_
↪serial, get_fn_serial
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: var('x,y,z')
(x, y, z)
sage: foo = function('foo', nargs=2)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_function_pystring(i, (x,y^z))
'{\\rm foo}\\left(x, y^{z}\\right)'
sage: py_latex_function_pystring(i, (x,y^z), True)
'\\left({\\rm foo}\\right)\\left(x, y^{z}\\right)'
sage: py_latex_function_pystring(i, (int(0),x))
'{\\rm foo}\\left(0, x\\right)'
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_latex_function_pystring, get_ginac_
↪serial, get_fn_serial
>>> from sage.symbolic.function import get_sfunction_from_serial
>>> var('x,y,z')
(x, y, z)
>>> foo = function('foo', nargs=Integer(2))
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...   if get_sfunction_from_serial(i) == foo: break

>>> get_sfunction_from_serial(i) == foo
True
>>> py_latex_function_pystring(i, (x,y**z))
'{\\rm foo}\\left(x, y^{z}\\right)'
>>> py_latex_function_pystring(i, (x,y**z), True)
'\\left({\\rm foo}\\right)\\left(x, y^{z}\\right)'
>>> py_latex_function_pystring(i, (int(Integer(0)),x))
'{\\rm foo}\\left(0, x\\right)'
```

Test latex_name:

```
sage: foo = function('foo', nargs=2, latex_name=r'\mathrm{bar}')
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
```

(continues on next page)

(continued from previous page)

```
True
sage: py_latex_function_pystring(i, (x,y^z))
'\\mathrm{bar}\\left(x, y^{z}\\right)'
```

```
>>> from sage.all import *
>>> foo = function('foo', nargs=Integer(2), latex_name=r'\mathrm{bar}')
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...   if get_sfunction_from_serial(i) == foo: break

>>> get_sfunction_from_serial(i) == foo
True
>>> py_latex_function_pystring(i, (x,y**z))
'\\mathrm{bar}\\left(x, y^{z}\\right)'
```

Test custom func:

```
sage: def my_print(self, *args): return "my args are: " + ', '.join(map(repr,␣
↪args))
sage: foo = function('foo', nargs=2, print_latex_func=my_print)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_latex_function_pystring(i, (x,y^z))
'my args are: x, y^z'
```

```
>>> from sage.all import *
>>> def my_print(self, *args): return "my args are: " + ', '.join(map(repr, args))
>>> foo = function('foo', nargs=Integer(2), print_latex_func=my_print)
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...   if get_sfunction_from_serial(i) == foo: break

>>> get_sfunction_from_serial(i) == foo
True
>>> py_latex_function_pystring(i, (x,y**z))
'my args are: x, y^z'
```

sage.symbolic.expression.**py_latex_variable_for_doctests**(*x*)

Internal function used so we can doctest a certain cdef'd method.

EXAMPLES:

```
sage: sage.symbolic.expression.py_latex_variable_for_doctests('x')
x
sage: sage.symbolic.expression.py_latex_variable_for_doctests('sigma')
\sigma
```

```
>>> from sage.all import *
>>> sage.symbolic.expression.py_latex_variable_for_doctests('x')
x
```

(continues on next page)

(continued from previous page)

```
>>> sage.symbolic.expression.py_latex_variable_for_doctests('sigma')
\sigma
```

sage.symbolic.expression.**py_lgamma_for_doctests**(*x*)

This function tests py_lgamma.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_lgamma_for_doctests
sage: py_lgamma_for_doctests(CC(I))
-0.650923199301856 - 1.87243664726243*I
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_lgamma_for_doctests
>>> py_lgamma_for_doctests(CC(I))
-0.650923199301856 - 1.87243664726243*I
```

sage.symbolic.expression.**py_li2_for_doctests**(*x*)

This function is a python wrapper so py_psi2 can be tested. The real tests are in the docstring for py_psi2.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_li2_for_doctests
sage: py_li2_for_doctests(-1.1)
-0.890838090262283
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_li2_for_doctests
>>> py_li2_for_doctests(-RealNumber('1.1'))
-0.890838090262283
```

sage.symbolic.expression.**py_li_for_doctests**(*x*, *n*, *parent*)

This function is a python wrapper so py_li can be tested. The real tests are in the docstring for py_li.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_li_for_doctests
sage: py_li_for_doctests(0,2,float)
0.000000000000000
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_li_for_doctests
>>> py_li_for_doctests(Integer(0),Integer(2),float)
0.000000000000000
```

sage.symbolic.expression.**py_log_for_doctests**(*x*)

This function tests py_log.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_log_for_doctests
sage: py_log_for_doctests(CC(e))
1.00000000000000
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_log_for_doctests
>>> py_log_for_doctests(CC(e))
1.00000000000000
```

sage.symbolic.expression.**py_mod_for_doctests**(*x*, *n*)

This function is a python wrapper so py_mod can be tested. The real tests are in the docstring for py_mod.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_mod_for_doctests
sage: py_mod_for_doctests(5, 2)
1
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_mod_for_doctests
>>> py_mod_for_doctests(Integer(5), Integer(2))
1
```

sage.symbolic.expression.**py_numer_for_doctests**(*n*)

This function is used to test py_numer().

EXAMPLES:

```
sage: from sage.symbolic.expression import py_numer_for_doctests
sage: py_numer_for_doctests(2/3)
2
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_numer_for_doctests
>>> py_numer_for_doctests(Integer(2)/Integer(3))
2
```

sage.symbolic.expression.**py_print_fderivative_for_doctests**(*id*, *params*, *args*)

Used for testing a cdef'd function.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_print_fderivative_for_doctests as␣
↪py_print_fderivative, get_ginac_serial, get_fn_serial
sage: var('x,y,z')
(x, y, z)
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: foo = function('foo', nargs=2)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:     if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_print_fderivative(i, (0, 1, 0, 1), (x, y^z))
D[0, 1, 0, 1](foo)(x, y^z)
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_print_fderivative_for_doctests as py_
↪print_fderivative, get_ginac_serial, get_fn_serial
>>> var('x,y,z')
(x, y, z)
>>> from sage.symbolic.function import get_sfunction_from_serial
>>> foo = function('foo', nargs=Integer(2))
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...     if get_sfunction_from_serial(i) == foo: break

>>> get_sfunction_from_serial(i) == foo
True
>>> py_print_fderivative(i, (Integer(0), Integer(1), Integer(0), Integer(1)), (x,␣
↪y**z))
D[0, 1, 0, 1](foo)(x, y^z)
```

Test custom print function:

```
sage: def my_print(self, *args): return "func_with_args(" + ', '.join(map(repr,␣
↪args)) +')'
sage: foo = function('foo', nargs=2, print_func=my_print)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:     if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_print_fderivative(i, (0, 1, 0, 1), (x, y^z))
D[0, 1, 0, 1]func_with_args(x, y^z)
```

```
>>> from sage.all import *
>>> def my_print(self, *args): return "func_with_args(" + ', '.join(map(repr,␣
↪args)) +')'
>>> foo = function('foo', nargs=Integer(2), print_func=my_print)
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...     if get_sfunction_from_serial(i) == foo: break

>>> get_sfunction_from_serial(i) == foo
True
>>> py_print_fderivative(i, (Integer(0), Integer(1), Integer(0), Integer(1)), (x,␣
↪y**z))
D[0, 1, 0, 1]func_with_args(x, y^z)
```

sage.symbolic.expression.**py_print_function_pystring**(*id*, *args*, *fname_paren=False*)

Return a string with the representation of the symbolic function specified by the given id applied to args.

INPUT:

- `id` – serial number of the corresponding symbolic function
- `params` – set of parameter numbers with respect to which to take the derivative
- `args` – arguments of the function

EXAMPLES:

```
sage: from sage.symbolic.expression import py_print_function_pystring, get_ginac_
↪serial, get_fn_serial
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: var('x,y,z')
(x, y, z)
sage: foo = function('foo', nargs=2)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_print_function_pystring(i, (x,y))
'foo(x, y)'
sage: py_print_function_pystring(i, (x,y), True)
'(foo)(x, y)'
sage: def my_print(self, *args): return "my args are: " + ', '.join(map(repr,␣
↪args))
sage: foo = function('foo', nargs=2, print_func=my_print)
sage: for i in range(get_ginac_serial(), get_fn_serial()):
....:   if get_sfunction_from_serial(i) == foo: break

sage: get_sfunction_from_serial(i) == foo
True
sage: py_print_function_pystring(i, (x,y))
'my args are: x, y'
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_print_function_pystring, get_ginac_
↪serial, get_fn_serial
>>> from sage.symbolic.function import get_sfunction_from_serial
>>> var('x,y,z')
(x, y, z)
>>> foo = function('foo', nargs=Integer(2))
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...   if get_sfunction_from_serial(i) == foo: break

>>> get_sfunction_from_serial(i) == foo
True
>>> py_print_function_pystring(i, (x,y))
'foo(x, y)'
>>> py_print_function_pystring(i, (x,y), True)
'(foo)(x, y)'
>>> def my_print(self, *args): return "my args are: " + ', '.join(map(repr, args))
>>> foo = function('foo', nargs=Integer(2), print_func=my_print)
>>> for i in range(get_ginac_serial(), get_fn_serial()):
...   if get_sfunction_from_serial(i) == foo: break

>>> get_sfunction_from_serial(i) == foo
True
>>> py_print_function_pystring(i, (x,y))
'my args are: x, y'
```

sage.symbolic.expression.**py_psi2_for_doctests**(n, x)

This function is a python wrapper so py_psi2 can be tested. The real tests are in the docstring for py_psi2.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_psi2_for_doctests
sage: py_psi2_for_doctests(1, 2)
0.644934066848226
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_psi2_for_doctests
>>> py_psi2_for_doctests(Integer(1), Integer(2))
0.644934066848226
```

sage.symbolic.expression.**py_psi_for_doctests**(*x*)

This function is a python wrapper so py_psi can be tested. The real tests are in the docstring for py_psi.

EXAMPLES:

```
sage: from sage.symbolic.expression import py_psi_for_doctests
sage: py_psi_for_doctests(2)
0.422784335098467
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_psi_for_doctests
>>> py_psi_for_doctests(Integer(2))
0.422784335098467
```

sage.symbolic.expression.**py_real_for_doctests**(*x*)

Used for doctesting py_real.

sage.symbolic.expression.**py_stieltjes_for_doctests**(*x*)

This function is for testing py_stieltjes().

EXAMPLES:

```
sage: from sage.symbolic.expression import py_stieltjes_for_doctests
sage: py_stieltjes_for_doctests(0.0)
0.577215664901533
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_stieltjes_for_doctests
>>> py_stieltjes_for_doctests(RealNumber('0.0'))
0.577215664901533
```

sage.symbolic.expression.**py_tgamma_for_doctests**(*x*)

This function is for testing py_tgamma().

sage.symbolic.expression.**py_zeta_for_doctests**(*x*)

This function is for testing py_zeta().

EXAMPLES:

```
sage: from sage.symbolic.expression import py_zeta_for_doctests
sage: py_zeta_for_doctests(CC.0)
0.00330022368532410 - 0.418155449141322*I
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import py_zeta_for_doctests
>>> py_zeta_for_doctests(CC.gen(0))
0.00330022368532410 - 0.418155449141322*I
```

`sage.symbolic.expression.`**`register_or_update_function`**(*self*, *name*, *latex_name*, *nargs*, *evalf_params_first*, *update*)

Register the function `self` with Pynac (GiNaC).

OUTPUT: serial number of the function, for use in `call_registered_function()`

EXAMPLES:

```
sage: from sage.symbolic.function import BuiltinFunction
sage: class Archosaurian(BuiltinFunction):
....:     def __init__(self):
....:         BuiltinFunction.__init__(self, 'archsaur', nargs=1)
....:     def _eval_(self, x):
....:         return x * exp(x)
sage: archsaur = Archosaurian()  # indirect doctest
sage: archsaur(2)
2*e^2
```

```
>>> from sage.all import *
>>> from sage.symbolic.function import BuiltinFunction
>>> class Archosaurian(BuiltinFunction):
...     def __init__(self):
...         BuiltinFunction.__init__(self, 'archsaur', nargs=Integer(1))
...     def _eval_(self, x):
...         return x * exp(x)
>>> archsaur = Archosaurian()  # indirect doctest
>>> archsaur(Integer(2))
2*e^2
```

`sage.symbolic.expression.`**`restore_op_wrapper`**(*expr*)

`sage.symbolic.expression.`**`solve_diophantine`**(*f*, **args*, ***kwds*)

Solve a Diophantine equation.

The argument, if not given as symbolic equation, is set equal to zero. It can be given in any form that can be converted to symbolic. Please see `Expression.solve_diophantine()` for a detailed synopsis.

EXAMPLES:

```
sage: R.<a,b> = PolynomialRing(ZZ); R
Multivariate Polynomial Ring in a, b over Integer Ring
sage: solve_diophantine(a^2 - 3*b^2 + 1)
[]
sage: sorted(solve_diophantine(a^2 - 3*b^2 + 2), key=str)
[(-1/2*sqrt(3)*(sqrt(3) + 2)^t + 1/2*sqrt(3)*(-sqrt(3) + 2)^t - 1/2*(sqrt(3) + 2)^
↪t - 1/2*(-sqrt(3) + 2)^t,
  -1/6*sqrt(3)*(sqrt(3) + 2)^t + 1/6*sqrt(3)*(-sqrt(3) + 2)^t - 1/2*(sqrt(3) + 2)^
↪t - 1/2*(-sqrt(3) + 2)^t),
  (1/2*sqrt(3)*(sqrt(3) + 2)^t - 1/2*sqrt(3)*(-sqrt(3) + 2)^t + 1/2*(sqrt(3) + 2)^
```

(continues on next page)

(continued from previous page)

```
↪t + 1/2*(-sqrt(3) + 2)^t,
   1/6*sqrt(3)*(sqrt(3) + 2)^t - 1/6*sqrt(3)*(-sqrt(3) + 2)^t + 1/2*(sqrt(3) + 2)^
↪t + 1/2*(-sqrt(3) + 2)^t)]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(ZZ, names=('a', 'b',)); (a, b,) = R._first_ngens(2); R
Multivariate Polynomial Ring in a, b over Integer Ring
>>> solve_diophantine(a**Integer(2) - Integer(3)*b**Integer(2) + Integer(1))
[]
>>> sorted(solve_diophantine(a**Integer(2) - Integer(3)*b**Integer(2) +␣
↪Integer(2)), key=str)
[(-1/2*sqrt(3)*(sqrt(3) + 2)^t + 1/2*sqrt(3)*(-sqrt(3) + 2)^t - 1/2*(sqrt(3) + 2)^
↪t - 1/2*(-sqrt(3) + 2)^t,
  -1/6*sqrt(3)*(sqrt(3) + 2)^t + 1/6*sqrt(3)*(-sqrt(3) + 2)^t - 1/2*(sqrt(3) + 2)^
↪t - 1/2*(-sqrt(3) + 2)^t),
  (1/2*sqrt(3)*(sqrt(3) + 2)^t - 1/2*sqrt(3)*(-sqrt(3) + 2)^t + 1/2*(sqrt(3) + 2)^
↪t + 1/2*(-sqrt(3) + 2)^t,
   1/6*sqrt(3)*(sqrt(3) + 2)^t - 1/6*sqrt(3)*(-sqrt(3) + 2)^t + 1/2*(sqrt(3) + 2)^
↪t + 1/2*(-sqrt(3) + 2)^t)]
```

`sage.symbolic.expression.`**`test_binomial`**`(n, k)`

The Binomial coefficients. It computes the binomial coefficients. For integer n and k and positive n this is the number of ways of choosing k objects from n distinct objects. If n is negative, the formula binomial(n,k) == (-1)^k*binomial(k-n-1,k) is used to compute the result.

INPUT:

- `n`, `k` – integers, with `k >= 0`

OUTPUT: integer

EXAMPLES:

```
sage: import sage.symbolic.expression
sage: sage.symbolic.expression.test_binomial(5,2)
10
sage: sage.symbolic.expression.test_binomial(-5,3)
-35
sage: -sage.symbolic.expression.test_binomial(3-(-5)-1, 3)
-35
```

```
>>> from sage.all import *
>>> import sage.symbolic.expression
>>> sage.symbolic.expression.test_binomial(Integer(5),Integer(2))
10
>>> sage.symbolic.expression.test_binomial(-Integer(5),Integer(3))
-35
>>> -sage.symbolic.expression.test_binomial(Integer(3)-(-Integer(5))-Integer(1),␣
↪Integer(3))
-35
```

`sage.symbolic.expression.`**`tolerant_is_symbol`**`(a)`

Utility function to test if something is a symbol.

Returns False for arguments that do not have an is_symbol attribute. Returns the result of calling the is_symbol method otherwise.

EXAMPLES:

```
sage: from sage.symbolic.expression import tolerant_is_symbol
sage: tolerant_is_symbol(var("x"))
True
sage: tolerant_is_symbol(None)
False
sage: None.is_symbol()
Traceback (most recent call last):
...
AttributeError: 'NoneType' object has no attribute 'is_symbol'...
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import tolerant_is_symbol
>>> tolerant_is_symbol(var("x"))
True
>>> tolerant_is_symbol(None)
False
>>> None.is_symbol()
Traceback (most recent call last):
...
AttributeError: 'NoneType' object has no attribute 'is_symbol'...
```

sage.symbolic.expression.**unpack_operands**(*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression import unpack_operands
sage: t = SR._force_pyobject((1, 2, x, x+1, x+2))
sage: unpack_operands(t)
(1, 2, x, x + 1, x + 2)
sage: type(unpack_operands(t))
<... 'tuple'>
sage: list(map(type, unpack_operands(t)))
[<class 'sage.rings.integer.Integer'>, <class 'sage.rings.integer.Integer'>,
↪<class 'sage.symbolic.expression.Expression'>, <class 'sage.symbolic.expression.
↪Expression'>, <class 'sage.symbolic.expression.Expression'>]
sage: u = SR._force_pyobject((t, x^2))
sage: unpack_operands(u)
((1, 2, x, x + 1, x + 2), x^2)
sage: type(unpack_operands(u)[0])
<... 'tuple'>
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression import unpack_operands
>>> t = SR._force_pyobject((Integer(1), Integer(2), x, x+Integer(1),␣
↪x+Integer(2)))
>>> unpack_operands(t)
(1, 2, x, x + 1, x + 2)
>>> type(unpack_operands(t))
<... 'tuple'>
```

(continues on next page)

(continued from previous page)

```
>>> list(map(type, unpack_operands(t)))
[<class 'sage.rings.integer.Integer'>, <class 'sage.rings.integer.Integer'>,
↪<class 'sage.symbolic.expression.Expression'>, <class 'sage.symbolic.expression.
↪Expression'>, <class 'sage.symbolic.expression.Expression'>]
>>> u = SR._force_pyobject((t, x**Integer(2)))
>>> unpack_operands(u)
((1, 2, x, x + 1, x + 2), x^2)
>>> type(unpack_operands(u)[Integer(0)])
<... 'tuple'>
```

2.2 Callable Symbolic Expressions

EXAMPLES:

When you do arithmetic with:

```
sage: f(x, y, z) = sin(x+y+z)
sage: g(x, y) = y + 2*x
sage: f + g
(x, y, z) |--> 2*x + y + sin(x + y + z)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y,z"); f = symbolic_expression(sin(x+y+z)).function(x,y,z)
>>> __tmp__=var("x,y"); g = symbolic_expression(y + Integer(2)*x).function(x,y)
>>> f + g
(x, y, z) |--> 2*x + y + sin(x + y + z)
```

```
sage: f(x, y, z) = sin(x+y+z)
sage: g(w, t) = cos(w - t)
sage: f + g
(t, w, x, y, z) |--> cos(-t + w) + sin(x + y + z)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y,z"); f = symbolic_expression(sin(x+y+z)).function(x,y,z)
>>> __tmp__=var("w,t"); g = symbolic_expression(cos(w - t)).function(w,t)
>>> f + g
(t, w, x, y, z) |--> cos(-t + w) + sin(x + y + z)
```

```
sage: f(x, y, t) = y*(x^2-t)
sage: g(x, y, w) = x + y - cos(w)
sage: f*g
(x, y, t, w) |--> (x^2 - t)*(x + y - cos(w))*y
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y,t"); f = symbolic_expression(y*(x**Integer(2)-t)).function(x,y,t)
>>> __tmp__=var("x,y,w"); g = symbolic_expression(x + y - cos(w)).function(x,y,w)
>>> f*g
(x, y, t, w) |--> (x^2 - t)*(x + y - cos(w))*y
```

```
sage: f(x,y, t) = x+y
sage: g(x, y, w) = w + t
sage: f + g
(x, y, t, w) |--> t + w + x + y
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y,t"); f = symbolic_expression(x+y).function(x,y,t)
>>> __tmp__=var("x,y,w"); g = symbolic_expression(w + t).function(x,y,w)
>>> f + g
(x, y, t, w) |--> t + w + x + y
```

class sage.symbolic.callable.**CallableSymbolicExpressionFunctor**(*arguments*)

Bases: ConstructionFunctor

A functor which produces a CallableSymbolicExpressionRing from the SymbolicRing.

EXAMPLES:

```
sage: from sage.symbolic.callable import CallableSymbolicExpressionFunctor
sage: x,y = var('x,y')
sage: f = CallableSymbolicExpressionFunctor((x,y)); f
CallableSymbolicExpressionFunctor(x, y)
sage: f(SR)
Callable function ring with arguments (x, y)

sage: loads(dumps(f))
CallableSymbolicExpressionFunctor(x, y)
```

```
>>> from sage.all import *
>>> from sage.symbolic.callable import CallableSymbolicExpressionFunctor
>>> x,y = var('x,y')
>>> f = CallableSymbolicExpressionFunctor((x,y)); f
CallableSymbolicExpressionFunctor(x, y)
>>> f(SR)
Callable function ring with arguments (x, y)

>>> loads(dumps(f))
CallableSymbolicExpressionFunctor(x, y)
```

arguments()

EXAMPLES:

```
sage: from sage.symbolic.callable import CallableSymbolicExpressionFunctor
sage: x,y = var('x,y')
sage: a = CallableSymbolicExpressionFunctor((x,y))
sage: a.arguments()
(x, y)
```

```
>>> from sage.all import *
>>> from sage.symbolic.callable import CallableSymbolicExpressionFunctor
>>> x,y = var('x,y')
>>> a = CallableSymbolicExpressionFunctor((x,y))
```

(continues on next page)

(continued from previous page)

```
>>> a.arguments()
(x, y)
```

merge(*other*)

EXAMPLES:

```
sage: from sage.symbolic.callable import CallableSymbolicExpressionFunctor
sage: x,y = var('x,y')
sage: a = CallableSymbolicExpressionFunctor((x,))
sage: b = CallableSymbolicExpressionFunctor((y,))
sage: a.merge(b)
CallableSymbolicExpressionFunctor(x, y)
```

```
>>> from sage.all import *
>>> from sage.symbolic.callable import CallableSymbolicExpressionFunctor
>>> x,y = var('x,y')
>>> a = CallableSymbolicExpressionFunctor((x,))
>>> b = CallableSymbolicExpressionFunctor((y,))
>>> a.merge(b)
CallableSymbolicExpressionFunctor(x, y)
```

unify_arguments(*x*)

Take the variable list from another `CallableSymbolicExpression` object and compare it with the current `CallableSymbolicExpression` object's variable list, combining them according to the following rules:

Let `a` be `self`'s variable list, let `b` be `y`'s variable list.

1. If `a == b`, then the variable lists are identical, so return that variable list.
2. If `a` $\neq$ `b`, then check if the first n items in `a` are the first n items in `b`, or vice versa. If so, return a list with these n items, followed by the remaining items in `a` and `b` sorted together in alphabetical order.

Note

When used for arithmetic between `CallableSymbolicExpression`'s, these rules ensure that the set of `CallableSymbolicExpression`'s will have certain properties. In particular, it ensures that the set is a *commutative* ring, i.e., the order of the input variables is the same no matter in which order arithmetic is done.

INPUT:

- `x` – a `CallableSymbolicExpression`

OUTPUT: a tuple of variables

EXAMPLES:

```
sage: from sage.symbolic.callable import CallableSymbolicExpressionFunctor
sage: x,y = var('x,y')
sage: a = CallableSymbolicExpressionFunctor((x,))
sage: b = CallableSymbolicExpressionFunctor((y,))
sage: a.unify_arguments(b)
(x, y)
```

```
>>> from sage.all import *
>>> from sage.symbolic.callable import CallableSymbolicExpressionFunctor
>>> x,y = var('x,y')
>>> a = CallableSymbolicExpressionFunctor((x,))
>>> b = CallableSymbolicExpressionFunctor((y,))
>>> a.unify_arguments(b)
(x, y)
```

AUTHORS:

- Bobby Moretti: thanks to William Stein for the rules

class sage.symbolic.callable.**CallableSymbolicExpressionRingFactory**

Bases: UniqueFactory

create_key(*args*, *check=True*)

EXAMPLES:

```
sage: x,y = var('x,y')
sage: CallableSymbolicExpressionRing.create_key((x,y))
(x, y)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> CallableSymbolicExpressionRing.create_key((x,y))
(x, y)
```

create_object(*version*, *key*, ***extra_args*)

Return a CallableSymbolicExpressionRing given a version and a key.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: CallableSymbolicExpressionRing.create_object(0, (x, y))
Callable function ring with arguments (x, y)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> CallableSymbolicExpressionRing.create_object(Integer(0), (x, y))
Callable function ring with arguments (x, y)
```

class sage.symbolic.callable.**CallableSymbolicExpressionRing_class**(*arguments*)

Bases: *SymbolicRing*, CallableSymbolicExpressionRing

EXAMPLES:

We verify that coercion works in the case where `x` is not an instance of SymbolicExpression, but its parent is still the SymbolicRing:

```
sage: f(x) = 1
sage: f*e
x |--> e
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(Integer(1)).function(x)
>>> f*e
x |--> e
```

args()

Return the arguments of `self`.

The order that the variables appear in `self.arguments()` is the order that is used in evaluating the elements of `self`.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f(x,y) = 2*x+y
sage: f.parent().arguments()
(x, y)
sage: f(y,x) = 2*x+y
sage: f.parent().arguments()
(y, x)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> __tmp__=var("x,y"); f = symbolic_expression(Integer(2)*x+y).function(x,y)
>>> f.parent().arguments()
(x, y)
>>> __tmp__=var("y,x"); f = symbolic_expression(Integer(2)*x+y).function(y,x)
>>> f.parent().arguments()
(y, x)
```

arguments()

Return the arguments of `self`.

The order that the variables appear in `self.arguments()` is the order that is used in evaluating the elements of `self`.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f(x,y) = 2*x+y
sage: f.parent().arguments()
(x, y)
sage: f(y,x) = 2*x+y
sage: f.parent().arguments()
(y, x)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> __tmp__=var("x,y"); f = symbolic_expression(Integer(2)*x+y).function(x,y)
>>> f.parent().arguments()
(x, y)
>>> __tmp__=var("y,x"); f = symbolic_expression(Integer(2)*x+y).function(y,x)
>>> f.parent().arguments()
(y, x)
```

construction()

EXAMPLES:

```
sage: f(x,y) = x^2 + y
sage: f.parent().construction()
(CallableSymbolicExpressionFunctor(x, y), Symbolic Ring)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(x**Integer(2) + y).function(x,
↪y)
>>> f.parent().construction()
(CallableSymbolicExpressionFunctor(x, y), Symbolic Ring)
```

2.3 Constructor for symbolic expressions

sage.calculus.expr.**symbolic_expression**(*x*)

Create a symbolic expression or vector of symbolic expressions from x.

INPUT:

- x - an object

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: a = symbolic_expression(3/2); a
3/2
sage: type(a)
<class 'sage.symbolic.expression.Expression'>
sage: R.<x> = QQ[]; type(x)
<class 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint
↪'>
sage: a = symbolic_expression(2*x^2 + 3); a
2*x^2 + 3
sage: type(a)
<class 'sage.symbolic.expression.Expression'>
sage: from sage.structure.element import Expression
sage: isinstance(a, Expression)
True
sage: a in SR
True
sage: a.parent()
Symbolic Ring
```

```
>>> from sage.all import *
>>> a = symbolic_expression(Integer(3)/Integer(2)); a
3/2
>>> type(a)
<class 'sage.symbolic.expression.Expression'>
>>> R = QQ['x']; (x,) = R._first_ngens(1); type(x)
<class 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint
↪'>
```

(continues on next page)

(continued from previous page)

```
>>> a = symbolic_expression(Integer(2)*x**Integer(2) + Integer(3)); a
2*x^2 + 3
>>> type(a)
<class 'sage.symbolic.expression.Expression'>
>>> from sage.structure.element import Expression
>>> isinstance(a, Expression)
True
>>> a in SR
True
>>> a.parent()
Symbolic Ring
```

Note that equations exist in the symbolic ring:

```
sage: # needs sage.schemes
sage: E = EllipticCurve('15a'); E
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 10*x - 10 over Rational␣
↪Field
sage: symbolic_expression(E)
x*y + y^2 + y == x^3 + x^2 - 10*x - 10
sage: symbolic_expression(E) in SR
True
```

```
>>> from sage.all import *
>>> # needs sage.schemes
>>> E = EllipticCurve('15a'); E
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 10*x - 10 over Rational␣
↪Field
>>> symbolic_expression(E)
x*y + y^2 + y == x^3 + x^2 - 10*x - 10
>>> symbolic_expression(E) in SR
True
```

If x is a list or tuple, create a vector of symbolic expressions:

```
sage: v = symbolic_expression([x,1]); v
(x, 1)
sage: v.base_ring()
Symbolic Ring
sage: v = symbolic_expression((x,1)); v
(x, 1)
sage: v.base_ring()
Symbolic Ring
sage: v = symbolic_expression((3,1)); v
(3, 1)
sage: v.base_ring()
Symbolic Ring
sage: E = EllipticCurve('15a'); E
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 10*x - 10 over Rational␣
↪Field
sage: v = symbolic_expression([E,E]); v
(x*y + y^2 + y == x^3 + x^2 - 10*x - 10, x*y + y^2 + y == x^3 + x^2 - 10*x - 10)
```

(continues on next page)

(continued from previous page)

```
sage: v.base_ring()
Symbolic Ring
```

```
>>> from sage.all import *
>>> v = symbolic_expression([x,Integer(1)]); v
(x, 1)
>>> v.base_ring()
Symbolic Ring
>>> v = symbolic_expression((x,Integer(1))); v
(x, 1)
>>> v.base_ring()
Symbolic Ring
>>> v = symbolic_expression((Integer(3),Integer(1))); v
(3, 1)
>>> v.base_ring()
Symbolic Ring
>>> E = EllipticCurve('15a'); E
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 10*x - 10 over Rational␣
↪Field
>>> v = symbolic_expression([E,E]); v
(x*y + y^2 + y == x^3 + x^2 - 10*x - 10, x*y + y^2 + y == x^3 + x^2 - 10*x - 10)
>>> v.base_ring()
Symbolic Ring
```

Likewise, if `x` is a vector, create a vector of symbolic expressions:

```
sage: u = vector([1, 2, 3])
sage: v = symbolic_expression(u); v
(1, 2, 3)
sage: v.parent()
Vector space of dimension 3 over Symbolic Ring
```

```
>>> from sage.all import *
>>> u = vector([Integer(1), Integer(2), Integer(3)])
>>> v = symbolic_expression(u); v
(1, 2, 3)
>>> v.parent()
Vector space of dimension 3 over Symbolic Ring
```

If `x` is a list or tuple of lists/tuples/vectors, create a matrix of symbolic expressions:

```
sage: M = symbolic_expression([[1, x, x^2], (x, x^2, x^3), vector([x^2, x^3, x^
↪4])]); M
[  1   x x^2]
[  x x^2 x^3]
[x^2 x^3 x^4]
sage: M.parent()
Full MatrixSpace of 3 by 3 dense matrices over Symbolic Ring
```

```
>>> from sage.all import *
>>> M = symbolic_expression([[Integer(1), x, x**Integer(2)], (x, x**Integer(2),␣
↪x**Integer(3)), vector([x**Integer(2), x**Integer(3), x**Integer(4)])]); M
```

(continues on next page)

(continued from previous page)

```
[  1   x x^2]
[  x x^2 x^3]
[x^2 x^3 x^4]
>>> M.parent()
Full MatrixSpace of 3 by 3 dense matrices over Symbolic Ring
```

If `x` is a matrix, create a matrix of symbolic expressions:

```
sage: A = matrix([[1, 2, 3], [4, 5, 6]])
sage: B = symbolic_expression(A); B
[1 2 3]
[4 5 6]
sage: B.parent()
Full MatrixSpace of 2 by 3 dense matrices over Symbolic Ring
```

```
>>> from sage.all import *
>>> A = matrix([[Integer(1), Integer(2), Integer(3)], [Integer(4), Integer(5),
→Integer(6)]])
>>> B = symbolic_expression(A); B
[1 2 3]
[4 5 6]
>>> B.parent()
Full MatrixSpace of 2 by 3 dense matrices over Symbolic Ring
```

If `x` is a function, for example defined by a `lambda` expression, create a symbolic function:

```
sage: f = symbolic_expression(lambda z: z^2 + 1); f
z |--> z^2 + 1
sage: f.parent()
Callable function ring with argument z
sage: f(7)
50
```

```
>>> from sage.all import *
>>> f = symbolic_expression(lambda z: z**Integer(2) + Integer(1)); f
z |--> z^2 + 1
>>> f.parent()
Callable function ring with argument z
>>> f(Integer(7))
50
```

If `x` is a list or tuple of functions, or if `x` is a function that returns a list or tuple, create a callable symbolic vector:

```
sage: symbolic_expression([lambda mu, nu: mu^2 + nu^2, lambda mu, nu: mu^2 - nu^
→2])
(mu, nu) |--> (mu^2 + nu^2, mu^2 - nu^2)
sage: f = symbolic_expression(lambda uwu: [1, uwu, uwu^2]); f
uwu |--> (1, uwu, uwu^2)
sage: f.parent()
Vector space of dimension 3 over Callable function ring with argument uwu
sage: f(5)
(1, 5, 25)
```

(continues on next page)

(continued from previous page)

```
sage: f(5).parent()
Vector space of dimension 3 over Symbolic Ring
```

```
>>> from sage.all import *
>>> symbolic_expression([lambda mu, nu: mu**Integer(2) + nu**Integer(2), lambda␣
↪mu, nu: mu**Integer(2) - nu**Integer(2)])
(mu, nu) |--> (mu^2 + nu^2, mu^2 - nu^2)
>>> f = symbolic_expression(lambda uwu: [Integer(1), uwu, uwu**Integer(2)]); f
uwu |--> (1, uwu, uwu^2)
>>> f.parent()
Vector space of dimension 3 over Callable function ring with argument uwu
>>> f(Integer(5))
(1, 5, 25)
>>> f(Integer(5)).parent()
Vector space of dimension 3 over Symbolic Ring
```

2.4 Assumptions

The `GenericDeclaration` class provides assumptions about a symbol or function in verbal form. Such assumptions can be made using the *assume()* function in this module, which also can take any relation of symbolic expressions as argument. Use *forget()* to clear all assumptions. Creating a variable with a specific domain is equivalent with making an assumption about it.

There is only rudimentary support for consistency and satisfiability checking in Sage. Assumptions are used both in Maxima and Pynac to support or refine some computations. In the following we show how to make and query assumptions. Please see the respective modules for more practical examples.

In addition to the global *assumptions()* database, *assuming()* creates reusable, stackable context managers allowing for temporary updates of the database for evaluation of a (block of) statements.

EXAMPLES:

The default domain of a symbolic variable is the complex plane:

```
sage: var('x')
x
sage: x.is_real()
False
sage: assume(x,'real')
sage: x.is_real()
True
sage: forget()
sage: x.is_real()
False
```

```
>>> from sage.all import *
>>> var('x')
x
>>> x.is_real()
False
>>> assume(x,'real')
>>> x.is_real()
```

(continues on next page)

(continued from previous page)

```
True
>>> forget()
>>> x.is_real()
False
```

Here is the list of acceptable features:

```
sage: ", ".join(map(str, maxima("features")._sage_()))
'integer, noninteger, even, odd, rational, irrational, real, imaginary,
complex, analytic, increasing, decreasing, oddfun, evenfun, posfun,
constant, commutative, lassociative, rassociative, symmetric,
antisymmetric, integervalued'
```

```
>>> from sage.all import *
>>> ", ".join(map(str, maxima("features")._sage_()))
'integer, noninteger, even, odd, rational, irrational, real, imaginary,
complex, analytic, increasing, decreasing, oddfun, evenfun, posfun,
constant, commutative, lassociative, rassociative, symmetric,
antisymmetric, integervalued'
```

Set positive domain using a relation:

```
sage: assume(x>0)
sage: x.is_positive()
True
sage: x.is_real()
True
sage: assumptions()
[x > 0]
```

```
>>> from sage.all import *
>>> assume(x>Integer(0))
>>> x.is_positive()
True
>>> x.is_real()
True
>>> assumptions()
[x > 0]
```

Assumptions also affect operations that do not use Maxima:

```
sage: forget()
sage: assume(x, 'even')
sage: assume(x, 'real')
sage: (-1)^x
1
sage: (-gamma(pi))^x
gamma(pi)^x
sage: binomial(2*x, x).is_integer()
True
```

```
>>> from sage.all import *
>>> forget()
>>> assume(x, 'even')
>>> assume(x, 'real')
>>> (-Integer(1))**x
1
>>> (-gamma(pi))**x
gamma(pi)^x
>>> binomial(Integer(2)*x, x).is_integer()
True
```

Assumptions are added and in some cases checked for consistency:

```
sage: assume(x>0)
sage: assume(x<0)
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
sage: forget()
```

```
>>> from sage.all import *
>>> assume(x>Integer(0))
>>> assume(x<Integer(0))
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
>>> forget()
```

class `sage.symbolic.assumptions.`**`GenericDeclaration`**`(`*var*, *assumption*`)`

Bases: `UniqueRepresentation`

This class represents generic assumptions, such as a variable being an integer or a function being increasing. It passes such information to Maxima's declare (wrapped in a context so it is able to forget) and to Pynac.

INPUT:

- `var` – the variable about which assumptions are being made
- `assumption` – string containing a Maxima feature, either user defined or in the list given by `maxima('features')`

EXAMPLES:

```
sage: from sage.symbolic.assumptions import GenericDeclaration
sage: decl = GenericDeclaration(x, 'integer')
sage: decl.assume()
sage: sin(x*pi)
0
sage: decl.forget()
sage: sin(x*pi)
sin(pi*x)
sage: sin(x*pi).simplify()
sin(pi*x)
```

```
>>> from sage.all import *
>>> from sage.symbolic.assumptions import GenericDeclaration
>>> decl = GenericDeclaration(x, 'integer')
>>> decl.assume()
>>> sin(x*pi)
0
>>> decl.forget()
>>> sin(x*pi)
sin(pi*x)
>>> sin(x*pi).simplify()
sin(pi*x)
```

Here is the list of acceptable features:

```
sage: ", ".join(map(str, maxima("features")._sage_()))
'integer, noninteger, even, odd, rational, irrational, real, imaginary,
complex, analytic, increasing, decreasing, oddfun, evenfun, posfun,
constant, commutative, lassociative, rassociative, symmetric,
antisymmetric, integervalued'
```

```
>>> from sage.all import *
>>> ", ".join(map(str, maxima("features")._sage_()))
'integer, noninteger, even, odd, rational, irrational, real, imaginary,
complex, analytic, increasing, decreasing, oddfun, evenfun, posfun,
constant, commutative, lassociative, rassociative, symmetric,
antisymmetric, integervalued'
```

Test unique representation behavior:

```
sage: GenericDeclaration(x, 'integer') is GenericDeclaration(SR.var("x"), 'integer
↪')
True
```

```
>>> from sage.all import *
>>> GenericDeclaration(x, 'integer') is GenericDeclaration(SR.var("x"), 'integer')
True
```

assume()

Make this assumption.

contradicts(*soln*)

Return `True` if this assumption is violated by the given variable assignment(s).

INPUT:

- `soln` – either a dictionary with variables as keys or a symbolic relation with a variable on the left hand side

EXAMPLES:

```
sage: from sage.symbolic.assumptions import GenericDeclaration
sage: GenericDeclaration(x, 'integer').contradicts(x==4)
False
sage: GenericDeclaration(x, 'integer').contradicts(x==4.0)
```

(continues on next page)

(continued from previous page)

```
False
sage: GenericDeclaration(x, 'integer').contradicts(x==4.5)
True
sage: GenericDeclaration(x, 'integer').contradicts(x==sqrt(17))
True
sage: GenericDeclaration(x, 'noninteger').contradicts(x==sqrt(17))
False
sage: GenericDeclaration(x, 'noninteger').contradicts(x==17)
True
sage: GenericDeclaration(x, 'even').contradicts(x==3)
True
sage: GenericDeclaration(x, 'complex').contradicts(x==3)
False
sage: GenericDeclaration(x, 'imaginary').contradicts(x==3)
True
sage: GenericDeclaration(x, 'imaginary').contradicts(x==I)
False

sage: var('y,z')
(y, z)
sage: GenericDeclaration(x, 'imaginary').contradicts(x==y+z)
False

sage: GenericDeclaration(x, 'rational').contradicts(y==pi)
False
sage: GenericDeclaration(x, 'rational').contradicts(x==pi)
True
sage: GenericDeclaration(x, 'irrational').contradicts(x!=pi)
False
sage: GenericDeclaration(x, 'rational').contradicts({x: pi, y: pi})
True
sage: GenericDeclaration(x, 'rational').contradicts({z: pi, y: pi})
False
```

```
>>> from sage.all import *
>>> from sage.symbolic.assumptions import GenericDeclaration
>>> GenericDeclaration(x, 'integer').contradicts(x==Integer(4))
False
>>> GenericDeclaration(x, 'integer').contradicts(x==RealNumber('4.0'))
False
>>> GenericDeclaration(x, 'integer').contradicts(x==RealNumber('4.5'))
True
>>> GenericDeclaration(x, 'integer').contradicts(x==sqrt(Integer(17)))
True
>>> GenericDeclaration(x, 'noninteger').contradicts(x==sqrt(Integer(17)))
False
>>> GenericDeclaration(x, 'noninteger').contradicts(x==Integer(17))
True
>>> GenericDeclaration(x, 'even').contradicts(x==Integer(3))
True
>>> GenericDeclaration(x, 'complex').contradicts(x==Integer(3))
False
```

(continues on next page)

(continued from previous page)

```
>>> GenericDeclaration(x, 'imaginary').contradicts(x==Integer(3))
True
>>> GenericDeclaration(x, 'imaginary').contradicts(x==I)
False

>>> var('y,z')
(y, z)
>>> GenericDeclaration(x, 'imaginary').contradicts(x==y+z)
False

>>> GenericDeclaration(x, 'rational').contradicts(y==pi)
False
>>> GenericDeclaration(x, 'rational').contradicts(x==pi)
True
>>> GenericDeclaration(x, 'irrational').contradicts(x!=pi)
False
>>> GenericDeclaration(x, 'rational').contradicts({x: pi, y: pi})
True
>>> GenericDeclaration(x, 'rational').contradicts({z: pi, y: pi})
False
```

forget()

Forget this assumption.

has(*arg*)

Check if this assumption contains the argument `arg`.

EXAMPLES:

```
sage: from sage.symbolic.assumptions import GenericDeclaration as GDecl
sage: var('y')
y
sage: d = GDecl(x, 'integer')
sage: d.has(x)
True
sage: d.has(y)
False
```

```
>>> from sage.all import *
>>> from sage.symbolic.assumptions import GenericDeclaration as GDecl
>>> var('y')
y
>>> d = GDecl(x, 'integer')
>>> d.has(x)
True
>>> d.has(y)
False
```

`sage.symbolic.assumptions.`**assume**(**args*)

Make the given assumptions.

INPUT:

- `*args` – a variable-length sequence of assumptions, each consisting of:

  - any number of symbolic inequalities, like `0 < x, x < 1`
  - a subsequence of variable names, followed by some property that should be assumed for those variables; for example, `x, y, z, 'integer'` would assume that each of `x`, `y`, and `z` are integer variables, and `x, 'odd'` would assume that `x` is odd (as opposed to even).

  The two types can be combined, but a symbolic inequality cannot appear in the middle of a list of variables.

OUTPUT: if everything goes as planned, there is no output

If you assume something that is not one of the two forms above, then an `AttributeError` is raised as we try to call its `assume` method.

If you make inconsistent assumptions (for example, that `x` is both even and odd), then a `ValueError` is raised.

Warning

Do not use Python's chained comparison notation in assumptions. Python literally translates the expression `0 < x < 1` to `(0 < x) and (x < 1)`, but the value of `bool(0 < x)` is `False` when `x` is a symbolic variable. Therefore, by the definition of Python's logical "and" operator, the entire expression is equal to `0 < x`.

EXAMPLES:

Assumptions are typically used to ensure certain relations are evaluated as true that are not true in general.

Here, we verify that for $x > 0$, $\sqrt{x^2} = x$:

```
sage: assume(x > 0)
sage: bool(sqrt(x^2) == x)
True
```

```
>>> from sage.all import *
>>> assume(x > Integer(0))
>>> bool(sqrt(x**Integer(2)) == x)
True
```

This will be assumed in the current Sage session until forgotten:

```
sage: bool(sqrt(x^2) == x)
True
sage: forget()
sage: bool(sqrt(x^2) == x)
False
```

```
>>> from sage.all import *
>>> bool(sqrt(x**Integer(2)) == x)
True
>>> forget()
>>> bool(sqrt(x**Integer(2)) == x)
False
```

Another major use case is in taking certain integrals and limits where the answers may depend on some sign condition:

```
sage: var('x, n')
(x, n)
```

(continues on next page)

(continued from previous page)

```
sage: assume(n+1>0)
sage: integral(x^n,x)
x^(n + 1)/(n + 1)
sage: forget()
```

```
>>> from sage.all import *
>>> var('x, n')
(x, n)
>>> assume(n+Integer(1)>Integer(0))
>>> integral(x**n,x)
x^(n + 1)/(n + 1)
>>> forget()
```

```
sage: var('q, a, k')
(q, a, k)
sage: assume(q > 1)
sage: sum(a*q^k, k, 0, oo)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
sage: forget()
sage: assume(abs(q) < 1)
sage: sum(a*q^k, k, 0, oo)
-a/(q - 1)
sage: forget()
```

```
>>> from sage.all import *
>>> var('q, a, k')
(q, a, k)
>>> assume(q > Integer(1))
>>> sum(a*q**k, k, Integer(0), oo)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
>>> forget()
>>> assume(abs(q) < Integer(1))
>>> sum(a*q**k, k, Integer(0), oo)
-a/(q - 1)
>>> forget()
```

An integer constraint:

```
sage: n,P,r,r2 = SR.var('n, P, r, r2')
sage: assume(n, 'integer')
sage: c = P*e^(r*n)
sage: d = P*(1+r2)^n
sage: solve(c==d,r2)
[r2 == e^r - 1]
sage: forget()
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> n,P,r,r2 = SR.var('n, P, r, r2')
>>> assume(n, 'integer')
>>> c = P*e**(r*n)
>>> d = P*(Integer(1)+r2)**n
>>> solve(c==d,r2)
[r2 == e^r - 1]
>>> forget()
```

Simplifying certain well-known identities works as well:

```
sage: n = SR.var('n')
sage: assume(n, 'integer')
sage: sin(n*pi)
0
sage: forget()
sage: sin(n*pi).simplify()
sin(pi*n)
```

```
>>> from sage.all import *
>>> n = SR.var('n')
>>> assume(n, 'integer')
>>> sin(n*pi)
0
>>> forget()
>>> sin(n*pi).simplify()
sin(pi*n)
```

Instead of using chained comparison notation, each relationship should be passed as a separate assumption:

```
sage: x = SR.var('x')
sage: assume(0 < x, x < 1) # instead of assume(0 < x < 1)
sage: assumptions()
[0 < x, x < 1]
sage: forget()
```

```
>>> from sage.all import *
>>> x = SR.var('x')
>>> assume(Integer(0) < x, x < Integer(1)) # instead of assume(0 < x < 1)
>>> assumptions()
[0 < x, x < 1]
>>> forget()
```

If you make inconsistent or meaningless assumptions, Sage will let you know:

```
sage: assume(x<0)
sage: assume(x>0)
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
sage: assume(x<1)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: Assumption is redundant
sage: assumptions()
[x < 0]
sage: forget()
sage: assume(x,'even')
sage: assume(x,'odd')
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
sage: forget()
```

```
>>> from sage.all import *
>>> assume(x<Integer(0))
>>> assume(x>Integer(0))
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
>>> assume(x<Integer(1))
Traceback (most recent call last):
...
ValueError: Assumption is redundant
>>> assumptions()
[x < 0]
>>> forget()
>>> assume(x,'even')
>>> assume(x,'odd')
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
>>> forget()
```

You can also use assumptions to evaluate simple truth values:

```
sage: x, y, z = var('x, y, z')
sage: assume(x>=y,y>=z,z>=x)
sage: bool(x==z)
True
sage: bool(z<x)
False
sage: bool(z>y)
False
sage: bool(y==z)
True
sage: forget()
sage: assume(x>=1,x<=1)
sage: bool(x==1)
True
sage: bool(x>1)
False
sage: forget()
```

```
>>> from sage.all import *
>>> x, y, z = var('x, y, z')
>>> assume(x>=y,y>=z,z>=x)
>>> bool(x==z)
True
>>> bool(z<x)
False
>>> bool(z>y)
False
>>> bool(y==z)
True
>>> forget()
>>> assume(x>=Integer(1),x<=Integer(1))
>>> bool(x==Integer(1))
True
>>> bool(x>Integer(1))
False
>>> forget()
```

class sage.symbolic.assumptions.**assuming**(**args*, ***kwds*)

Bases: `object`

Temporarily modify assumptions.

Create a context manager in which temporary assumptions are added (or substituted) to the current assumptions set.

The set of possible assumptions and declarations is the same as for *assume()*.

This can be useful in interactive mode to discover the assumptions necessary to a given integration, or the exact solution to a system of equations.

It can also be used to explore the branches of a `cases()` expression.

As with *assume()*, it is an error to add an assumption either redundant or inconsistent with the current assumption set (unless `replace=True` is used). See examples.

INPUT:

- `*args` – assumptions (same format as for *assume()*)
- `replace` – boolean (default: `False`); specifies whether the new assumptions are added to (default) or replace (if `replace=True`) the current assumption set

OUTPUT:

A context manager useable in a `with` statement (see examples).

EXAMPLES:

Basic functionality : inside a `with assuming:()` block, Sage uses the updated assumptions database. After exit, the original database is restored.

```
sage: var("x")
x
sage: forget(assumptions())
sage: solve(x^2 == 4,x)
[x == -2, x == 2]
sage: with assuming(x > 0):
```

(continues on next page)

(continued from previous page)

```
....:     solve(x^2 == 4,x)
[x == 2]
sage: assumptions()
[]
```

```
>>> from sage.all import *
>>> var("x")
x
>>> forget(assumptions())
>>> solve(x**Integer(2) == Integer(4),x)
[x == -2, x == 2]
>>> with assuming(x > Integer(0)):
...     solve(x**Integer(2) == Integer(4),x)
[x == 2]
>>> assumptions()
[]
```

The local assumptions can be stacked. We can use this functionality to discover incrementally the assumptions necessary to a given calculation (and by the way, to check that Sage's default integrator (Maxima's, that is), sometimes nitpicks for naught).

```
sage: var("y,k,theta")
(y, k, theta)
sage: dgamma(y,k,theta)=y^(k-1)*e^(-y/theta)/(theta^k*gamma(k))
sage: integrate(dgamma(y,k,theta),y,0,oo)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints;␣
↪using the 'assume' command before evaluation *may* help (example of legal␣
↪syntax is 'assume(theta>0)', see `assume?` for more details)
Is theta positive or negative?
sage: a1=assuming(theta>0)
sage: with a1:integrate(dgamma(y,k,theta),y,0,oo)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints;␣
↪using the 'assume' command before evaluation *may* help (example of legal␣
↪syntax is 'assume(k>0)', see `assume?` for more details)
Is k positive, negative or zero?
sage: a2=assuming(k>0)
sage: with a1,a2:integrate(dgamma(y,k,theta),y,0,oo)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints;␣
↪using the 'assume' command before evaluation *may* help (example of legal␣
↪syntax is 'assume(k>0)', see `assume?` for more details)
Is k an integer?
sage: a3=assuming(k,"noninteger")
sage: with a1,a2,a3:integrate(dgamma(y,k,theta),y,0,oo)
1
sage: a4=assuming(k,"integer")
sage: with a1,a2,a4:integrate(dgamma(y,k,theta),y,0,oo)
```

(continues on next page)

(continued from previous page)

```
1
```

```
>>> from sage.all import *
>>> var("y,k,theta")
(y, k, theta)
>>> __tmp__=var("y,k,theta"); dgamma = symbolic_expression(y**(k-Integer(1))*e**(-
↪y/theta)/(theta**k*gamma(k))).function(y,k,theta)
>>> integrate(dgamma(y,k,theta),y,Integer(0),oo)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints;␣
↪using the 'assume' command before evaluation *may* help (example of legal␣
↪syntax is 'assume(theta>0)', see `assume?` for more details)
Is theta positive or negative?
>>> a1=assuming(theta>Integer(0))
>>> with a1:integrate(dgamma(y,k,theta),y,Integer(0),oo)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints;␣
↪using the 'assume' command before evaluation *may* help (example of legal␣
↪syntax is 'assume(k>0)', see `assume?` for more details)
Is k positive, negative or zero?
>>> a2=assuming(k>Integer(0))
>>> with a1,a2:integrate(dgamma(y,k,theta),y,Integer(0),oo)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints;␣
↪using the 'assume' command before evaluation *may* help (example of legal␣
↪syntax is 'assume(k>0)', see `assume?` for more details)
Is k an integer?
>>> a3=assuming(k,"noninteger")
>>> with a1,a2,a3:integrate(dgamma(y,k,theta),y,Integer(0),oo)
1
>>> a4=assuming(k,"integer")
>>> with a1,a2,a4:integrate(dgamma(y,k,theta),y,Integer(0),oo)
1
```

As mentioned above, it is an error to try to introduce redundant or inconsistent assumptions.

```
sage: assume(x > 0)
sage: with assuming(x > -1): "I won't see this"
Traceback (most recent call last):
...
ValueError: Assumption is redundant

sage: with assuming(x < -1): "I won't see this"
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
```

```
>>> from sage.all import *
>>> assume(x > Integer(0))
```

(continues on next page)

(continued from previous page)

```
>>> with assuming(x > -Integer(1)): "I won't see this"
Traceback (most recent call last):
...
ValueError: Assumption is redundant

>>> with assuming(x < -Integer(1)): "I won't see this"
Traceback (most recent call last):
...
ValueError: Assumption is inconsistent
```

sage.symbolic.assumptions.**assumptions**(**args*)

List all current symbolic assumptions.

INPUT:

- args – list of variables which can be empty

OUTPUT: list of assumptions on variables; if args is empty it returns all
assumptions

EXAMPLES:

```
sage: var('x, y, z, w')
(x, y, z, w)
sage: forget()
sage: assume(x^2+y^2 > 0)
sage: assumptions()
[x^2 + y^2 > 0]
sage: forget(x^2+y^2 > 0)
sage: assumptions()
[]
sage: assume(x > y)
sage: assume(z > w)
sage: sorted(assumptions(), key=lambda x: str(x))
[x > y, z > w]
sage: forget()
sage: assumptions()
[]
```

```
>>> from sage.all import *
>>> var('x, y, z, w')
(x, y, z, w)
>>> forget()
>>> assume(x**Integer(2)+y**Integer(2) > Integer(0))
>>> assumptions()
[x^2 + y^2 > 0]
>>> forget(x**Integer(2)+y**Integer(2) > Integer(0))
>>> assumptions()
[]
>>> assume(x > y)
>>> assume(z > w)
>>> sorted(assumptions(), key=lambda x: str(x))
[x > y, z > w]
```

(continues on next page)

(continued from previous page)

```
>>> forget()
>>> assumptions()
[]
```

It is also possible to query for assumptions on a variable independently:

```
sage: x, y, z = var('x y z')
sage: assume(x, 'integer')
sage: assume(y > 0)
sage: assume(y**2 + z**2 == 1)
sage: assume(x < 0)
sage: assumptions()
[x is integer, y > 0, y^2 + z^2 == 1, x < 0]
sage: assumptions(x)
[x is integer, x < 0]
sage: assumptions(x, y)
[x is integer, x < 0, y > 0, y^2 + z^2 == 1]
sage: assumptions(z)
[y^2 + z^2 == 1]
```

```
>>> from sage.all import *
>>> x, y, z = var('x y z')
>>> assume(x, 'integer')
>>> assume(y > Integer(0))
>>> assume(y**Integer(2) + z**Integer(2) == Integer(1))
>>> assume(x < Integer(0))
>>> assumptions()
[x is integer, y > 0, y^2 + z^2 == 1, x < 0]
>>> assumptions(x)
[x is integer, x < 0]
>>> assumptions(x, y)
[x is integer, x < 0, y > 0, y^2 + z^2 == 1]
>>> assumptions(z)
[y^2 + z^2 == 1]
```

sage.symbolic.assumptions.**forget**(**args*)

Forget the given assumption, or call with no arguments to forget all assumptions.

Here an assumption is some sort of symbolic constraint.

INPUT:

- `*args` – assumptions (default: forget all assumptions)

EXAMPLES:

We define and forget multiple assumptions:

```
sage: forget()
sage: var('x,y,z')
(x, y, z)
sage: assume(x>0, y>0, z == 1, y>0)
sage: sorted(assumptions(), key=lambda x:str(x))
[x > 0, y > 0, z == 1]
```

(continues on next page)

(continued from previous page)

```
sage: forget(x>0, z==1)
sage: assumptions()
[y > 0]
sage: assume(y, 'even', z, 'complex')
sage: assumptions()
[y > 0, y is even, z is complex]
sage: cos(y*pi).simplify()
1
sage: forget(y,'even')
sage: cos(y*pi).simplify()
cos(pi*y)
sage: assumptions()
[y > 0, z is complex]
sage: forget()
sage: assumptions()
[]
```

```
>>> from sage.all import *
>>> forget()
>>> var('x,y,z')
(x, y, z)
>>> assume(x>Integer(0), y>Integer(0), z == Integer(1), y>Integer(0))
>>> sorted(assumptions(), key=lambda x:str(x))
[x > 0, y > 0, z == 1]
>>> forget(x>Integer(0), z==Integer(1))
>>> assumptions()
[y > 0]
>>> assume(y, 'even', z, 'complex')
>>> assumptions()
[y > 0, y is even, z is complex]
>>> cos(y*pi).simplify()
1
>>> forget(y,'even')
>>> cos(y*pi).simplify()
cos(pi*y)
>>> assumptions()
[y > 0, z is complex]
>>> forget()
>>> assumptions()
[]
```

sage.symbolic.assumptions.**preprocess_assumptions**(*args*)

Turn a list of the form `(var1, var2, ..., 'property')` into a sequence of declarations `(var1 is property), (var2 is property), ...`

EXAMPLES:

```
sage: from sage.symbolic.assumptions import preprocess_assumptions
sage: preprocess_assumptions([x, 'integer', x > 4])
[x is integer, x > 4]
sage: var('x, y')
(x, y)
```

(continues on next page)

(continued from previous page)

```
sage: preprocess_assumptions([x, y, 'integer', x > 4, y, 'even'])
[x is integer, y is integer, x > 4, y is even]
```

```
>>> from sage.all import *
>>> from sage.symbolic.assumptions import preprocess_assumptions
>>> preprocess_assumptions([x, 'integer', x > Integer(4)])
[x is integer, x > 4]
>>> var('x, y')
(x, y)
>>> preprocess_assumptions([x, y, 'integer', x > Integer(4), y, 'even'])
[x is integer, y is integer, x > 4, y is even]
```

2.5 Symbolic Equations and Inequalities

Sage can solve symbolic equations and inequalities. For example, we derive the quadratic formula as follows:

```
sage: a,b,c = var('a,b,c')
sage: qe = (a*x^2 + b*x + c == 0)
sage: qe
a*x^2 + b*x + c == 0
sage: print(solve(qe, x))
[
x == -1/2*(b + sqrt(b^2 - 4*a*c))/a,
x == -1/2*(b - sqrt(b^2 - 4*a*c))/a
]
```

```
>>> from sage.all import *
>>> a,b,c = var('a,b,c')
>>> qe = (a*x**Integer(2) + b*x + c == Integer(0))
>>> qe
a*x^2 + b*x + c == 0
>>> print(solve(qe, x))
[
x == -1/2*(b + sqrt(b^2 - 4*a*c))/a,
x == -1/2*(b - sqrt(b^2 - 4*a*c))/a
]
```

2.5.1 The operator, left hand side, and right hand side

Operators:

```
sage: eqn = x^3 + 2/3 >= x - pi
sage: eqn.operator()
<built-in function ge>
sage: (x^3 + 2/3 < x - pi).operator()
<built-in function lt>
sage: (x^3 + 2/3 == x - pi).operator()
<built-in function eq>
```

```
>>> from sage.all import *
>>> eqn = x**Integer(3) + Integer(2)/Integer(3) >= x - pi
>>> eqn.operator()
<built-in function ge>
>>> (x**Integer(3) + Integer(2)/Integer(3) < x - pi).operator()
<built-in function lt>
>>> (x**Integer(3) + Integer(2)/Integer(3) == x - pi).operator()
<built-in function eq>
```

Left hand side:

```
sage: eqn = x^3 + 2/3 >= x - pi
sage: eqn.lhs()
x^3 + 2/3
sage: eqn.left()
x^3 + 2/3
sage: eqn.left_hand_side()
x^3 + 2/3
```

```
>>> from sage.all import *
>>> eqn = x**Integer(3) + Integer(2)/Integer(3) >= x - pi
>>> eqn.lhs()
x^3 + 2/3
>>> eqn.left()
x^3 + 2/3
>>> eqn.left_hand_side()
x^3 + 2/3
```

Right hand side:

```
sage: (x + sqrt(2) >= sqrt(3) + 5/2).right()
sqrt(3) + 5/2
sage: (x + sqrt(2) >= sqrt(3) + 5/2).rhs()
sqrt(3) + 5/2
sage: (x + sqrt(2) >= sqrt(3) + 5/2).right_hand_side()
sqrt(3) + 5/2
```

```
>>> from sage.all import *
>>> (x + sqrt(Integer(2)) >= sqrt(Integer(3)) + Integer(5)/Integer(2)).right()
sqrt(3) + 5/2
>>> (x + sqrt(Integer(2)) >= sqrt(Integer(3)) + Integer(5)/Integer(2)).rhs()
sqrt(3) + 5/2
>>> (x + sqrt(Integer(2)) >= sqrt(Integer(3)) + Integer(5)/Integer(2)).right_hand_
↪side()
sqrt(3) + 5/2
```

2.5.2 Arithmetic

Add two symbolic equations:

```
sage: var('a,b')
(a, b)
```

(continues on next page)

(continued from previous page)

```
sage: m = 144 == -10 * a + b
sage: n = 136 == 10 * a + b
sage: m + n
280 == 2*b
sage: int(-144) + m
0 == -10*a + b - 144
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> m = Integer(144) == -Integer(10) * a + b
>>> n = Integer(136) == Integer(10) * a + b
>>> m + n
280 == 2*b
>>> int(-Integer(144)) + m
0 == -10*a + b - 144
```

Subtract two symbolic equations:

```
sage: var('a,b')
(a, b)
sage: m = 144 == 20 * a + b
sage: n = 136 == 10 * a + b
sage: m - n
8 == 10*a
sage: int(144) - m
0 == -20*a - b + 144
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> m = Integer(144) == Integer(20) * a + b
>>> n = Integer(136) == Integer(10) * a + b
>>> m - n
8 == 10*a
>>> int(Integer(144)) - m
0 == -20*a - b + 144
```

Multiply two symbolic equations:

```
sage: x = var('x')
sage: m = x == 5*x + 1
sage: n = sin(x) == sin(x+2*pi, hold=True)
sage: m * n
x*sin(x) == (5*x + 1)*sin(2*pi + x)
sage: m = 2*x == 3*x^2 - 5
sage: int(-1) * m
-2*x == -3*x^2 + 5
```

```
>>> from sage.all import *
>>> x = var('x')
>>> m = x == Integer(5)*x + Integer(1)
```

(continues on next page)

(continued from previous page)

```
>>> n = sin(x) == sin(x+Integer(2)*pi, hold=True)
>>> m * n
x*sin(x) == (5*x + 1)*sin(2*pi + x)
>>> m = Integer(2)*x == Integer(3)*x**Integer(2) - Integer(5)
>>> int(-Integer(1)) * m
-2*x == -3*x^2 + 5
```

Divide two symbolic equations:

```
sage: x = var('x')
sage: m = x == 5*x + 1
sage: n = sin(x) == sin(x+2*pi, hold=True)
sage: m/n
x/sin(x) == (5*x + 1)/sin(2*pi + x)
sage: m = x != 5*x + 1
sage: n = sin(x) != sin(x+2*pi, hold=True)
sage: m/n
x/sin(x) != (5*x + 1)/sin(2*pi + x)
```

```
>>> from sage.all import *
>>> x = var('x')
>>> m = x == Integer(5)*x + Integer(1)
>>> n = sin(x) == sin(x+Integer(2)*pi, hold=True)
>>> m/n
x/sin(x) == (5*x + 1)/sin(2*pi + x)
>>> m = x != Integer(5)*x + Integer(1)
>>> n = sin(x) != sin(x+Integer(2)*pi, hold=True)
>>> m/n
x/sin(x) != (5*x + 1)/sin(2*pi + x)
```

2.5.3 Substitution

Substitution into relations:

```
sage: x, a = var('x, a')
sage: eq = (x^3 + a == sin(x/a)); eq
x^3 + a == sin(x/a)
sage: eq.substitute(x=5*x)
125*x^3 + a == sin(5*x/a)
sage: eq.substitute(a=1)
x^3 + 1 == sin(x)
sage: eq.substitute(a=x)
x^3 + x == sin(1)
sage: eq.substitute(a=x, x=1)
x + 1 == sin(1/x)
sage: eq.substitute({a:x, x:1})
x + 1 == sin(1/x)
```

```
>>> from sage.all import *
>>> x, a = var('x, a')
>>> eq = (x**Integer(3) + a == sin(x/a)); eq
```

(continues on next page)

(continued from previous page)

```
x^3 + a == sin(x/a)
>>> eq.substitute(x=Integer(5)*x)
125*x^3 + a == sin(5*x/a)
>>> eq.substitute(a=Integer(1))
x^3 + 1 == sin(x)
>>> eq.substitute(a=x)
x^3 + x == sin(1)
>>> eq.substitute(a=x, x=Integer(1))
x + 1 == sin(1/x)
>>> eq.substitute({a:x, x:Integer(1)})
x + 1 == sin(1/x)
```

You can even substitute multivariable and matrix expressions:

```
sage: x,y = var('x, y')
sage: M = Matrix([[x+1,y],[x^2,y^3]]); M
[x + 1     y]
[  x^2   y^3]
sage: M.substitute({x:0,y:1})
[1 1]
[0 1]
```

```
>>> from sage.all import *
>>> x,y = var('x, y')
>>> M = Matrix([[x+Integer(1),y],[x**Integer(2),y**Integer(3)]]); M
[x + 1     y]
[  x^2   y^3]
>>> M.substitute({x:Integer(0),y:Integer(1)})
[1 1]
[0 1]
```

2.5.4 Solving

We can solve equations:

```
sage: x = var('x')
sage: S = solve(x^3 - 1 == 0, x)
sage: S
[x == 1/2*I*sqrt(3) - 1/2, x == -1/2*I*sqrt(3) - 1/2, x == 1]
sage: S[0]
x == 1/2*I*sqrt(3) - 1/2
sage: S[0].right()
1/2*I*sqrt(3) - 1/2
sage: S = solve(x^3 - 1 == 0, x, solution_dict=True)
sage: S
[{x: 1/2*I*sqrt(3) - 1/2}, {x: -1/2*I*sqrt(3) - 1/2}, {x: 1}]
sage: z = 5
sage: solve(z^2 == sqrt(3),z)
Traceback (most recent call last):
...
TypeError: 5 is not a valid variable.
```

```
>>> from sage.all import *
>>> x = var('x')
>>> S = solve(x**Integer(3) - Integer(1) == Integer(0), x)
>>> S
[x == 1/2*I*sqrt(3) - 1/2, x == -1/2*I*sqrt(3) - 1/2, x == 1]
>>> S[Integer(0)]
x == 1/2*I*sqrt(3) - 1/2
>>> S[Integer(0)].right()
1/2*I*sqrt(3) - 1/2
>>> S = solve(x**Integer(3) - Integer(1) == Integer(0), x, solution_dict=True)
>>> S
[{x: 1/2*I*sqrt(3) - 1/2}, {x: -1/2*I*sqrt(3) - 1/2}, {x: 1}]
>>> z = Integer(5)
>>> solve(z**Integer(2) == sqrt(Integer(3)),z)
Traceback (most recent call last):
...
TypeError: 5 is not a valid variable.
```

We can also solve equations involving matrices. The following example defines a multivariable function `f(x,y)`, then solves for where the partial derivatives with respect to `x` and `y` are zero. Then it substitutes one of the solutions into the Hessian matrix `H` for `f`:

```
sage: f(x,y) = x^2*y+y^2+y
sage: solutions = solve(list(f.diff()),[x,y],solution_dict=True)
sage: solutions == [{x: -I, y: 0}, {x: I, y: 0}, {x: 0, y: -1/2}]
True
sage: H = f.diff(2) # Hessian matrix
sage: H.subs(solutions[2])
[(x, y) |--> -1  (x, y) |--> 0]
[ (x, y) |--> 0  (x, y) |--> 2]
sage: H(x,y).subs(solutions[2])
[-1  0]
[ 0  2]
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(x**Integer(2)*y+y**Integer(2)+y).
↪function(x,y)
>>> solutions = solve(list(f.diff()),[x,y],solution_dict=True)
>>> solutions == [{x: -I, y: Integer(0)}, {x: I, y: Integer(0)}, {x: Integer(0), y: -
↪Integer(1)/Integer(2)}]
True
>>> H = f.diff(Integer(2)) # Hessian matrix
>>> H.subs(solutions[Integer(2)])
[(x, y) |--> -1  (x, y) |--> 0]
[ (x, y) |--> 0  (x, y) |--> 2]
>>> H(x,y).subs(solutions[Integer(2)])
[-1  0]
[ 0  2]
```

We illustrate finding multiplicities of solutions:

```
sage: f = (x-1)^5*(x^2+1)
sage: solve(f == 0, x)
```

(continues on next page)

(continued from previous page)

```
[x == -I, x == I, x == 1]
sage: solve(f == 0, x, multiplicities=True)
([x == -I, x == I, x == 1], [1, 1, 5])
```

```
>>> from sage.all import *
>>> f = (x-Integer(1))**Integer(5)*(x**Integer(2)+Integer(1))
>>> solve(f == Integer(0), x)
[x == -I, x == I, x == 1]
>>> solve(f == Integer(0), x, multiplicities=True)
([x == -I, x == I, x == 1], [1, 1, 5])
```

We can also solve many inequalities:

```
sage: solve(1/(x-1)<=8,x)
[[x < 1], [x >= (9/8)]]
```

```
>>> from sage.all import *
>>> solve(Integer(1)/(x-Integer(1))<=Integer(8),x)
[[x < 1], [x >= (9/8)]]
```

We can numerically find roots of equations:

```
sage: (x == sin(x)).find_root(-2,2)
0.0
sage: (x^5 + 3*x + 2 == 0).find_root(-2,2,x)
-0.6328345202421523
sage: (cos(x) == sin(x)).find_root(10,20)
19.634954084936208
```

```
>>> from sage.all import *
>>> (x == sin(x)).find_root(-Integer(2),Integer(2))
0.0
>>> (x**Integer(5) + Integer(3)*x + Integer(2) == Integer(0)).find_root(-Integer(2),
↪Integer(2),x)
-0.6328345202421523
>>> (cos(x) == sin(x)).find_root(Integer(10),Integer(20))
19.634954084936208
```

We illustrate some valid error conditions:

```
sage: (cos(x) != sin(x)).find_root(10,20)
Traceback (most recent call last):
...
ValueError: Symbolic equation must be an equality.
sage: (SR(3)==SR(2)).find_root(-1,1)
Traceback (most recent call last):
...
RuntimeError: no zero in the interval, since constant expression is not 0.
```

```
>>> from sage.all import *
>>> (cos(x) != sin(x)).find_root(Integer(10),Integer(20))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: Symbolic equation must be an equality.
>>> (SR(Integer(3))==SR(Integer(2))).find_root(-Integer(1),Integer(1))
Traceback (most recent call last):
...
RuntimeError: no zero in the interval, since constant expression is not 0.
```

There must be at most one variable:

```
sage: x, y = var('x,y')
sage: (x == y).find_root(-2,2)
Traceback (most recent call last):
...
NotImplementedError: root finding currently only implemented in 1 dimension.
```

```
>>> from sage.all import *
>>> x, y = var('x,y')
>>> (x == y).find_root(-Integer(2),Integer(2))
Traceback (most recent call last):
...
NotImplementedError: root finding currently only implemented in 1 dimension.
```

2.5.5 Assumptions

Forgetting assumptions:

```
sage: var('x,y')
(x, y)
sage: forget() #Clear assumptions
sage: assume(x>0, y < 2)
sage: assumptions()
[x > 0, y < 2]
sage: (y < 2).forget()
sage: assumptions()
[x > 0]
sage: forget()
sage: assumptions()
[]
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> forget() #Clear assumptions
>>> assume(x>Integer(0), y < Integer(2))
>>> assumptions()
[x > 0, y < 2]
>>> (y < Integer(2)).forget()
>>> assumptions()
[x > 0]
>>> forget()
```

(continues on next page)

(continued from previous page)

```
>>> assumptions()
[]
```

2.5.6 Miscellaneous

Conversion to Maxima:

```
sage: x = var('x')
sage: eq = (x^(3/5) >= pi^2 + e^i)
sage: eq._maxima_init_()
'(_SAGE_VAR_x)^(3/5) >= ((%pi)^(2))+(exp(0+%i*1))'
sage: e1 = x^3 + x == sin(2*x)
sage: z = e1._maxima_()
sage: z.parent() is sage.calculus.calculus.maxima
True
sage: z = e1._maxima_(maxima)
sage: z.parent() is maxima
True
sage: z = maxima(e1)
sage: z.parent() is maxima
True
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eq = (x**(Integer(3)/Integer(5)) >= pi**Integer(2) + e**i)
>>> eq._maxima_init_()
'(_SAGE_VAR_x)^(3/5) >= ((%pi)^(2))+(exp(0+%i*1))'
>>> e1 = x**Integer(3) + x == sin(Integer(2)*x)
>>> z = e1._maxima_()
>>> z.parent() is sage.calculus.calculus.maxima
True
>>> z = e1._maxima_(maxima)
>>> z.parent() is maxima
True
>>> z = maxima(e1)
>>> z.parent() is maxima
True
```

Conversion to Maple:

```
sage: x = var('x')
sage: eq = (x == 2)
sage: eq._maple_init_()
'x = 2'
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eq = (x == Integer(2))
>>> eq._maple_init_()
'x = 2'
```

Comparison:

```
sage: x = var('x')
sage: (x>0) == (x>0)
True
sage: (x>0) == (x>1)
False
sage: (x>0) != (x>1)
True
```

```
>>> from sage.all import *
>>> x = var('x')
>>> (x>Integer(0)) == (x>Integer(0))
True
>>> (x>Integer(0)) == (x>Integer(1))
False
>>> (x>Integer(0)) != (x>Integer(1))
True
```

Variables appearing in the relation:

```
sage: var('x,y,z,w')
(x, y, z, w)
sage: f =  (x+y+w) == (x^2 - y^2 - z^3);   f
w + x + y == -z^3 + x^2 - y^2
sage: f.variables()
(w, x, y, z)
```

```
>>> from sage.all import *
>>> var('x,y,z,w')
(x, y, z, w)
>>> f =  (x+y+w) == (x**Integer(2) - y**Integer(2) - z**Integer(3));   f
w + x + y == -z^3 + x^2 - y^2
>>> f.variables()
(w, x, y, z)
```

LaTeX output:

```
sage: latex(x^(3/5) >= pi)
x^{\frac{3}{5}} \geq \pi
```

```
>>> from sage.all import *
>>> latex(x**(Integer(3)/Integer(5)) >= pi)
x^{\frac{3}{5}} \geq \pi
```

When working with the symbolic complex number I, notice that comparisons do not automatically simplify even in trivial situations:

```
sage: SR(I)^2 == -1
-1 == -1
sage: SR(I)^2 < 0
-1 < 0
sage: (SR(I)+1)^4 > 0
-4 > 0
```

```
>>> from sage.all import *
>>> SR(I)**Integer(2) == -Integer(1)
-1 == -1
>>> SR(I)**Integer(2) < Integer(0)
-1 < 0
>>> (SR(I)+Integer(1))**Integer(4) > Integer(0)
-4 > 0
```

Nevertheless, if you force the comparison, you get the right answer (Issue #7160):

```
sage: bool(SR(I)^2 == -1)
True
sage: bool(SR(I)^2 < 0)
True
sage: bool((SR(I)+1)^4 > 0)
False
```

```
>>> from sage.all import *
>>> bool(SR(I)**Integer(2) == -Integer(1))
True
>>> bool(SR(I)**Integer(2) < Integer(0))
True
>>> bool((SR(I)+Integer(1))**Integer(4) > Integer(0))
False
```

2.5.7 More Examples

```
sage: x,y,a = var('x,y,a')
sage: f = x^2 + y^2 == 1
sage: f.solve(x)
[x == -sqrt(-y^2 + 1), x == sqrt(-y^2 + 1)]
```

```
>>> from sage.all import *
>>> x,y,a = var('x,y,a')
>>> f = x**Integer(2) + y**Integer(2) == Integer(1)
>>> f.solve(x)
[x == -sqrt(-y^2 + 1), x == sqrt(-y^2 + 1)]
```

```
sage: f = x^5 + a
sage: solve(f==0,x)
[x == 1/4*(-a)^(1/5)*(sqrt(5) + I*sqrt(2*sqrt(5) + 10) - 1), x == -1/4*(-a)^(1/
↪5)*(sqrt(5) - I*sqrt(-2*sqrt(5) + 10) + 1), x == -1/4*(-a)^(1/5)*(sqrt(5) + I*sqrt(-
↪2*sqrt(5) + 10) + 1), x == 1/4*(-a)^(1/5)*(sqrt(5) - I*sqrt(2*sqrt(5) + 10) - 1), x␣
↪== (-a)^(1/5)]
```

```
>>> from sage.all import *
>>> f = x**Integer(5) + a
>>> solve(f==Integer(0),x)
[x == 1/4*(-a)^(1/5)*(sqrt(5) + I*sqrt(2*sqrt(5) + 10) - 1), x == -1/4*(-a)^(1/
↪5)*(sqrt(5) - I*sqrt(-2*sqrt(5) + 10) + 1), x == -1/4*(-a)^(1/5)*(sqrt(5) + I*sqrt(-
↪2*sqrt(5) + 10) + 1), x == 1/4*(-a)^(1/5)*(sqrt(5) - I*sqrt(2*sqrt(5) + 10) - 1), x␣
```

(continues on next page)

(continued from previous page)

```
↪== (-a)^(1/5)]
```

You can also do arithmetic with inequalities, as illustrated below:

```
sage: var('x y')
(x, y)
sage: f = x + 3 == y - 2
sage: f
x + 3 == y - 2
sage: g = f - 3; g
x == y - 5
sage: h =  x^3 + sqrt(2) == x*y*sin(x)
sage: h
x^3 + sqrt(2) == x*y*sin(x)
sage: h - sqrt(2)
x^3 == x*y*sin(x) - sqrt(2)
sage: h + f
x^3 + x + sqrt(2) + 3 == x*y*sin(x) + y - 2
sage: f = x + 3 < y - 2
sage: g = 2 < x+10
sage: f - g
x + 1 < -x + y - 12
sage: f + g
x + 5 < x + y + 8
sage: f*(-1)
-x - 3 < -y + 2
```

```
>>> from sage.all import *
>>> var('x y')
(x, y)
>>> f = x + Integer(3) == y - Integer(2)
>>> f
x + 3 == y - 2
>>> g = f - Integer(3); g
x == y - 5
>>> h =  x**Integer(3) + sqrt(Integer(2)) == x*y*sin(x)
>>> h
x^3 + sqrt(2) == x*y*sin(x)
>>> h - sqrt(Integer(2))
x^3 == x*y*sin(x) - sqrt(2)
>>> h + f
x^3 + x + sqrt(2) + 3 == x*y*sin(x) + y - 2
>>> f = x + Integer(3) < y - Integer(2)
>>> g = Integer(2) < x+Integer(10)
>>> f - g
x + 1 < -x + y - 12
>>> f + g
x + 5 < x + y + 8
>>> f*(-Integer(1))
-x - 3 < -y + 2
```

AUTHORS:

- Bobby Moretti: initial version (based on a trick that Robert Bradshaw suggested).
- William Stein: second version
- William Stein (2007-07-16): added arithmetic with symbolic equations

sage.symbolic.relation.**solve**(*f*, **args*, ***kwds*)

Algebraically solve an equation or system of equations (over the complex numbers) for given variables. Inequalities and systems of inequalities are also supported.

INPUT:

- `f` – equation or system of equations (given by a list or tuple)
- `*args` – variables to solve for
- `solution_dict` – boolean (default: `False`); if `True` or nonzero, return a list of dictionaries containing the solutions. If there are no solutions, return an empty list (rather than a list containing an empty dictionary). Likewise, if there's only a single solution, return a list containing one dictionary with that solution.

There are a few optional keywords if you are trying to solve a single equation. They may only be used in that context.

- `multiplicities` – boolean (default: `False`); if True, return corresponding multiplicities. This keyword is incompatible with `to_poly_solve=True` and does not make any sense when solving inequalities.
- `explicit_solutions` – boolean (default: `False`); require that all roots be explicit rather than implicit. Not used when solving inequalities.
- `to_poly_solve` – boolean (default: `False`) or string; use Maxima's `to_poly_solver` package to search for more possible solutions, but possibly encounter approximate solutions. This keyword is incompatible with `multiplicities=True` and is not used when solving inequalities. Setting `to_poly_solve` to 'force' (string) omits Maxima's solve command (useful when some solutions of trigonometric equations are lost).
- `algorithm` – string (default: `'maxima'`); to use SymPy's solvers set this to 'sympy'. Note that SymPy is always used for diophantine equations. Another choice, if it is installed, is 'giac'.
- `domain` – string (default: `'complex'`); setting this to 'real' changes the way SymPy solves single equations; inequalities are always solved in the real domain.

EXAMPLES:

```
sage: x, y = var('x, y')
sage: solve([x+y==6, x-y==4], x, y)
[[x == 5, y == 1]]
sage: solve([x^2+y^2 == 1, y^2 == x^3 + x + 1], x, y)
[[x == -1/2*I*sqrt(3) - 1/2, y == -sqrt(-1/2*I*sqrt(3) + 3/2)],
 [x == -1/2*I*sqrt(3) - 1/2, y == sqrt(-1/2*I*sqrt(3) + 3/2)],
 [x == 1/2*I*sqrt(3) - 1/2, y == -sqrt(1/2*I*sqrt(3) + 3/2)],
 [x == 1/2*I*sqrt(3) - 1/2, y == sqrt(1/2*I*sqrt(3) + 3/2)],
 [x == 0, y == -1],
 [x == 0, y == 1]]
sage: solve([sqrt(x) + sqrt(y) == 5, x + y == 10], x, y)
[[x == -5/2*I*sqrt(5) + 5, y == 5/2*I*sqrt(5) + 5], [x == 5/2*I*sqrt(5) + 5, y ==␣
↪-5/2*I*sqrt(5) + 5]]
sage: solutions = solve([x^2+y^2 == 1, y^2 == x^3 + x + 1], x, y, solution_
↪dict=True)
sage: for solution in solutions: print("{} , {}".format(solution[x].n(digits=3),␣
↪solution[y].n(digits=3)))
-0.500 - 0.866*I , -1.27 + 0.341*I
```

(continues on next page)

(continued from previous page)

```
-0.500 - 0.866*I , 1.27 - 0.341*I
-0.500 + 0.866*I , -1.27 - 0.341*I
-0.500 + 0.866*I , 1.27 + 0.341*I
0.000 , -1.00
0.000 , 1.00
```

```
>>> from sage.all import *
>>> x, y = var('x, y')
>>> solve([x+y==Integer(6), x-y==Integer(4)], x, y)
[[x == 5, y == 1]]
>>> solve([x**Integer(2)+y**Integer(2) == Integer(1), y**Integer(2) ==␣
↪x**Integer(3) + x + Integer(1)], x, y)
[[x == -1/2*I*sqrt(3) - 1/2, y == -sqrt(-1/2*I*sqrt(3) + 3/2)],
 [x == -1/2*I*sqrt(3) - 1/2, y == sqrt(-1/2*I*sqrt(3) + 3/2)],
 [x == 1/2*I*sqrt(3) - 1/2, y == -sqrt(1/2*I*sqrt(3) + 3/2)],
 [x == 1/2*I*sqrt(3) - 1/2, y == sqrt(1/2*I*sqrt(3) + 3/2)],
 [x == 0, y == -1],
 [x == 0, y == 1]]
>>> solve([sqrt(x) + sqrt(y) == Integer(5), x + y == Integer(10)], x, y)
[[x == -5/2*I*sqrt(5) + 5, y == 5/2*I*sqrt(5) + 5], [x == 5/2*I*sqrt(5) + 5, y ==␣
↪-5/2*I*sqrt(5) + 5]]
>>> solutions = solve([x**Integer(2)+y**Integer(2) == Integer(1), y**Integer(2)␣
↪== x**Integer(3) + x + Integer(1)], x, y, solution_dict=True)
>>> for solution in solutions: print("{} , {}".format(solution[x].
↪n(digits=Integer(3)), solution[y].n(digits=Integer(3))))
-0.500 - 0.866*I , -1.27 + 0.341*I
-0.500 - 0.866*I , 1.27 - 0.341*I
-0.500 + 0.866*I , -1.27 - 0.341*I
-0.500 + 0.866*I , 1.27 + 0.341*I
0.000 , -1.00
0.000 , 1.00
```

Whenever possible, answers will be symbolic, but with systems of equations, at times approximations will be given by Maxima, due to the underlying algorithm:

```
sage: sols = solve([x^3==y,y^2==x], [x,y]); sols[-1], sols[0] # abs tol 1e-15
([x == 0, y == 0],
 [x == (0.3090169943749475 + 0.9510565162951535*I),
  y == (-0.8090169943749475 - 0.5877852522924731*I)])
sage: sols[0][0].rhs().pyobject().parent()
Complex Double Field

sage: solve([y^6==y],y)
[y == 1/4*sqrt(5) + 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,
 y == -1/4*sqrt(5) + 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,
 y == -1/4*sqrt(5) - 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,
 y == 1/4*sqrt(5) - 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,
 y == 1,
 y == 0]
sage: solve( [y^6 == y], y)==solve( y^6 == y, y)
True
```

```
>>> from sage.all import *
>>> sols = solve([x**Integer(3)==y,y**Integer(2)==x], [x,y]); sols[-Integer(1)],␣
↪sols[Integer(0)] # abs tol 1e-15
([x == 0, y == 0],
 [x == (0.3090169943749475 + 0.9510565162951535*I),
  y == (-0.8090169943749475 - 0.5877852522924731*I)])
>>> sols[Integer(0)][Integer(0)].rhs().pyobject().parent()
Complex Double Field

>>> solve([y**Integer(6)==y],y)
[y == 1/4*sqrt(5) + 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,
 y == -1/4*sqrt(5) + 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,
 y == -1/4*sqrt(5) - 1/4*I*sqrt(-2*sqrt(5) + 10) - 1/4,
 y == 1/4*sqrt(5) - 1/4*I*sqrt(2*sqrt(5) + 10) - 1/4,
 y == 1,
 y == 0]
>>> solve( [y**Integer(6) == y], y)==solve( y**Integer(6) == y, y)
True
```

Here we demonstrate very basic use of the optional keywords:

```
sage: ((x^2-1)^2).solve(x)
[x == -1, x == 1]
sage: ((x^2-1)^2).solve(x,multiplicities=True)
([x == -1, x == 1], [2, 2])
sage: solve(sin(x)==x,x)
[x == sin(x)]
sage: solve(sin(x)==x,x,explicit_solutions=True)
[]
sage: solve(abs(1-abs(1-x)) == 10, x)
[abs(abs(x - 1) - 1) == 10]
sage: solve(abs(1-abs(1-x)) == 10, x, to_poly_solve=True)
[x == -10, x == 12]

sage: from sage.symbolic.expression import Expression
sage: Expression.solve(x^2==1,x)
[x == -1, x == 1]
```

```
>>> from sage.all import *
>>> ((x**Integer(2)-Integer(1))**Integer(2)).solve(x)
[x == -1, x == 1]
>>> ((x**Integer(2)-Integer(1))**Integer(2)).solve(x,multiplicities=True)
([x == -1, x == 1], [2, 2])
>>> solve(sin(x)==x,x)
[x == sin(x)]
>>> solve(sin(x)==x,x,explicit_solutions=True)
[]
>>> solve(abs(Integer(1)-abs(Integer(1)-x)) == Integer(10), x)
[abs(abs(x - 1) - 1) == 10]
>>> solve(abs(Integer(1)-abs(Integer(1)-x)) == Integer(10), x, to_poly_solve=True)
[x == -10, x == 12]

>>> from sage.symbolic.expression import Expression
```

(continues on next page)

(continued from previous page)

```
>>> Expression.solve(x**Integer(2)==Integer(1),x)
[x == -1, x == 1]
```

We must solve with respect to actual variables:

```
sage: z = 5
sage: solve([8*z + y == 3, -z +7*y == 0],y,z)
Traceback (most recent call last):
...
TypeError: 5 is not a valid variable.
```

```
>>> from sage.all import *
>>> z = Integer(5)
>>> solve([Integer(8)*z + y == Integer(3), -z +Integer(7)*y == Integer(0)],y,z)
Traceback (most recent call last):
...
TypeError: 5 is not a valid variable.
```

If we ask for dictionaries containing the solutions, we get them:

```
sage: solve([x^2-1],x,solution_dict=True)
[{x: -1}, {x: 1}]
sage: solve([x^2-4*x+4],x,solution_dict=True)
[{x: 2}]
sage: res = solve([x^2 == y, y == 4],x,y,solution_dict=True)
sage: for soln in res: print("x: %s, y: %s" % (soln[x], soln[y]))
x: 2, y: 4
x: -2, y: 4
```

```
>>> from sage.all import *
>>> solve([x**Integer(2)-Integer(1)],x,solution_dict=True)
[{x: -1}, {x: 1}]
>>> solve([x**Integer(2)-Integer(4)*x+Integer(4)],x,solution_dict=True)
[{x: 2}]
>>> res = solve([x**Integer(2) == y, y == Integer(4)],x,y,solution_dict=True)
>>> for soln in res: print("x: %s, y: %s" % (soln[x], soln[y]))
x: 2, y: 4
x: -2, y: 4
```

If there is a parameter in the answer, that will show up as a new variable. In the following example, `r1` is an arbitrary constant (because of the `r`):

```
sage: forget()
sage: x, y = var('x,y')
sage: solve([x+y == 3, 2*x+2*y == 6],x,y)
[[x == -r1 + 3, y == r1]]

sage: var('b, c')
(b, c)
sage: solve((b-1)*(c-1), [b,c])
[[b == 1, c == r...], [b == r..., c == 1]]
```

```
>>> from sage.all import *
>>> forget()
>>> x, y = var('x,y')
>>> solve([x+y == Integer(3), Integer(2)*x+Integer(2)*y == Integer(6)],x,y)
[[x == -r1 + 3, y == r1]]

>>> var('b, c')
(b, c)
>>> solve((b-Integer(1))*(c-Integer(1)), [b,c])
[[b == 1, c == r...], [b == r..., c == 1]]
```

Especially with trigonometric functions, the dummy variable may be implicitly an integer (hence the `z`):

```
sage: solve( sin(x)==cos(x), x, to_poly_solve=True)
[x == 1/4*pi + pi*z...]
sage: solve([cos(x)*sin(x) == 1/2, x+y == 0],x,y)
[[x == 1/4*pi + pi*z..., y == -1/4*pi - pi*z...]]
```

```
>>> from sage.all import *
>>> solve( sin(x)==cos(x), x, to_poly_solve=True)
[x == 1/4*pi + pi*z...]
>>> solve([cos(x)*sin(x) == Integer(1)/Integer(2), x+y == Integer(0)],x,y)
[[x == 1/4*pi + pi*z..., y == -1/4*pi - pi*z...]]
```

Expressions which are not equations are assumed to be set equal to zero, as with x in the following example:

```
sage: solve([x, y == 2],x,y)
[[x == 0, y == 2]]
```

```
>>> from sage.all import *
>>> solve([x, y == Integer(2)],x,y)
[[x == 0, y == 2]]
```

If `True` appears in the list of equations it is ignored, and if `False` appears in the list then no solutions are returned. E.g., note that the first `3==3` evaluates to `True`, not to a symbolic equation.

```
sage: solve([3==3, 1.00000000000000*x^3 == 0], x)
[x == 0]
sage: solve([1.00000000000000*x^3 == 0], x)
[x == 0]
```

```
>>> from sage.all import *
>>> solve([Integer(3)==Integer(3), RealNumber('1.00000000000000')*x**Integer(3)␣
↪== Integer(0)], x)
[x == 0]
>>> solve([RealNumber('1.00000000000000')*x**Integer(3) == Integer(0)], x)
[x == 0]
```

Here, the first equation evaluates to `False`, so there are no solutions:

```
sage: solve([1==3, 1.00000000000000*x^3 == 0], x)
[]
```

```
>>> from sage.all import *
>>> solve([Integer(1)==Integer(3), RealNumber('1.00000000000000')*x**Integer(3)␣
↪== Integer(0)], x)
[]
```

Completely symbolic solutions are supported:

```
sage: var('s,j,b,m,g')
(s, j, b, m, g)
sage: sys = [ m*(1-s) - b*s*j, b*s*j-g*j ]
sage: solve(sys,s,j)
[[s == 1, j == 0], [s == g/b, j == (b - g)*m/(b*g)]]
sage: solve(sys,(s,j))
[[s == 1, j == 0], [s == g/b, j == (b - g)*m/(b*g)]]
sage: solve(sys,[s,j])
[[s == 1, j == 0], [s == g/b, j == (b - g)*m/(b*g)]]

sage: z = var('z')
sage: solve((x-z)^2==2, x)
[x == z - sqrt(2), x == z + sqrt(2)]
```

```
>>> from sage.all import *
>>> var('s,j,b,m,g')
(s, j, b, m, g)
>>> sys = [ m*(Integer(1)-s) - b*s*j, b*s*j-g*j ]
>>> solve(sys,s,j)
[[s == 1, j == 0], [s == g/b, j == (b - g)*m/(b*g)]]
>>> solve(sys,(s,j))
[[s == 1, j == 0], [s == g/b, j == (b - g)*m/(b*g)]]
>>> solve(sys,[s,j])
[[s == 1, j == 0], [s == g/b, j == (b - g)*m/(b*g)]]

>>> z = var('z')
>>> solve((x-z)**Integer(2)==Integer(2), x)
[x == z - sqrt(2), x == z + sqrt(2)]
```

Inequalities can be also solved:

```
sage: solve(x^2>8,x)
[[x < -2*sqrt(2)], [x > 2*sqrt(2)]]
sage: x,y = var('x,y'); (ln(x)-ln(y)>0).solve(x)
[[log(x) - log(y) > 0]]
sage: x,y = var('x,y'); (ln(x)>ln(y)).solve(x)  # random
[[0 < y, y < x, 0 < x]]
[[y < x, 0 < y]]
```

```
>>> from sage.all import *
>>> solve(x**Integer(2)>Integer(8),x)
[[x < -2*sqrt(2)], [x > 2*sqrt(2)]]
>>> x,y = var('x,y'); (ln(x)-ln(y)>Integer(0)).solve(x)
[[log(x) - log(y) > 0]]
>>> x,y = var('x,y'); (ln(x)>ln(y)).solve(x)  # random
```

(continues on next page)

(continued from previous page)

```
[[0 < y, y < x, 0 < x]]
[[y < x, 0 < y]]
```

A simple example to show the use of the keyword `multiplicities`:

```
sage: ((x^2-1)^2).solve(x)
[x == -1, x == 1]
sage: ((x^2-1)^2).solve(x,multiplicities=True)
([x == -1, x == 1], [2, 2])
sage: ((x^2-1)^2).solve(x,multiplicities=True,to_poly_solve=True)
Traceback (most recent call last):
...
NotImplementedError: to_poly_solve does not return multiplicities
```

```
>>> from sage.all import *
>>> ((x**Integer(2)-Integer(1))**Integer(2)).solve(x)
[x == -1, x == 1]
>>> ((x**Integer(2)-Integer(1))**Integer(2)).solve(x,multiplicities=True)
([x == -1, x == 1], [2, 2])
>>> ((x**Integer(2)-Integer(1))**Integer(2)).solve(x,multiplicities=True,to_poly_
↪solve=True)
Traceback (most recent call last):
...
NotImplementedError: to_poly_solve does not return multiplicities
```

Here is how the `explicit_solutions` keyword functions:

```
sage: solve(sin(x)==x,x)
[x == sin(x)]
sage: solve(sin(x)==x,x,explicit_solutions=True)
[]
sage: solve(x*sin(x)==x^2,x)
[x == 0, x == sin(x)]
sage: solve(x*sin(x)==x^2,x,explicit_solutions=True)
[x == 0]
```

```
>>> from sage.all import *
>>> solve(sin(x)==x,x)
[x == sin(x)]
>>> solve(sin(x)==x,x,explicit_solutions=True)
[]
>>> solve(x*sin(x)==x**Integer(2),x)
[x == 0, x == sin(x)]
>>> solve(x*sin(x)==x**Integer(2),x,explicit_solutions=True)
[x == 0]
```

The following examples show the use of the keyword `to_poly_solve`:

```
sage: solve(abs(1-abs(1-x)) == 10, x)
[abs(abs(x - 1) - 1) == 10]
sage: solve(abs(1-abs(1-x)) == 10, x, to_poly_solve=True)
[x == -10, x == 12]
```

(continues on next page)

(continued from previous page)

```
sage: var('Q')
Q
sage: solve(Q*sqrt(Q^2 + 2) - 1, Q)
[Q == 1/sqrt(Q^2 + 2)]
```

```
>>> from sage.all import *
>>> solve(abs(Integer(1)-abs(Integer(1)-x)) == Integer(10), x)
[abs(abs(x - 1) - 1) == 10]
>>> solve(abs(Integer(1)-abs(Integer(1)-x)) == Integer(10), x, to_poly_solve=True)
[x == -10, x == 12]

>>> var('Q')
Q
>>> solve(Q*sqrt(Q**Integer(2) + Integer(2)) - Integer(1), Q)
[Q == 1/sqrt(Q^2 + 2)]
```

The following example is a regression in Maxima 5.39.0. It used to be possible to get one more solution here, namely `1/sqrt(sqrt(2) + 1)`, see https://sourceforge.net/p/maxima/bugs/3276/:

```
sage: solve(Q*sqrt(Q^2 + 2) - 1, Q, to_poly_solve=True)
[Q == -sqrt(-sqrt(2) - 1), Q == sqrt(sqrt(2) + 1)*(sqrt(2) - 1)]
```

```
>>> from sage.all import *
>>> solve(Q*sqrt(Q**Integer(2) + Integer(2)) - Integer(1), Q, to_poly_solve=True)
[Q == -sqrt(-sqrt(2) - 1), Q == sqrt(sqrt(2) + 1)*(sqrt(2) - 1)]
```

An effort is made to only return solutions that satisfy the current assumptions:

```
sage: solve(x^2==4, x)
[x == -2, x == 2]
sage: assume(x<0)
sage: solve(x^2==4, x)
[x == -2]
sage: solve((x^2-4)^2 == 0, x, multiplicities=True)
([x == -2], [2])
sage: solve(x^2==2, x)
[x == -sqrt(2)]
sage: z = var('z')
sage: solve(x^2==2-z, x)
[x == -sqrt(-z + 2)]
sage: assume(x, 'rational')
sage: solve(x^2 == 2, x)
[]
```

```
>>> from sage.all import *
>>> solve(x**Integer(2)==Integer(4), x)
[x == -2, x == 2]
>>> assume(x<Integer(0))
>>> solve(x**Integer(2)==Integer(4), x)
[x == -2]
>>> solve((x**Integer(2)-Integer(4))**Integer(2) == Integer(0), x,
```

(continues on next page)

(continued from previous page)

```
↪multiplicities=True)
([x == -2], [2])
>>> solve(x**Integer(2)==Integer(2), x)
[x == -sqrt(2)]
>>> z = var('z')
>>> solve(x**Integer(2)==Integer(2)-z, x)
[x == -sqrt(-z + 2)]
>>> assume(x, 'rational')
>>> solve(x**Integer(2) == Integer(2), x)
[]
```

In some cases it may be worthwhile to directly use `to_poly_solve` if one suspects some answers are being missed:

```
sage: forget()
sage: solve(cos(x)==0, x)
[x == 1/2*pi]
sage: solve(cos(x)==0, x, to_poly_solve=True)
[x == 1/2*pi]
sage: solve(cos(x)==0, x, to_poly_solve='force')
[x == 1/2*pi + pi*z...]
```

```
>>> from sage.all import *
>>> forget()
>>> solve(cos(x)==Integer(0), x)
[x == 1/2*pi]
>>> solve(cos(x)==Integer(0), x, to_poly_solve=True)
[x == 1/2*pi]
>>> solve(cos(x)==Integer(0), x, to_poly_solve='force')
[x == 1/2*pi + pi*z...]
```

The same may also apply if a returned unsolved expression has a denominator, but the original one did not:

```
sage: solve(cos(x) * sin(x) == 1/2, x, to_poly_solve=True)
[sin(x) == 1/2/cos(x)]
sage: solve(cos(x) * sin(x) == 1/2, x, to_poly_solve=True, explicit_
↪solutions=True)
[x == 1/4*pi + pi*z...]
sage: solve(cos(x) * sin(x) == 1/2, x, to_poly_solve='force')
[x == 1/4*pi + pi*z...]
```

```
>>> from sage.all import *
>>> solve(cos(x) * sin(x) == Integer(1)/Integer(2), x, to_poly_solve=True)
[sin(x) == 1/2/cos(x)]
>>> solve(cos(x) * sin(x) == Integer(1)/Integer(2), x, to_poly_solve=True,␣
↪explicit_solutions=True)
[x == 1/4*pi + pi*z...]
>>> solve(cos(x) * sin(x) == Integer(1)/Integer(2), x, to_poly_solve='force')
[x == 1/4*pi + pi*z...]
```

We use `use_grobner` in Maxima if no solution is obtained from Maxima's `to_poly_solve`:

```
sage: x,y = var('x y')
sage: c1(x,y) = (x-5)^2+y^2-16
sage: c2(x,y) = (y-3)^2+x^2-9
sage: solve([c1(x,y),c2(x,y)],[x,y])
[[x == -9/68*sqrt(55) + 135/68, y == -15/68*sqrt(55) + 123/68],
 [x == 9/68*sqrt(55) + 135/68, y == 15/68*sqrt(55) + 123/68]]
```

```
>>> from sage.all import *
>>> x,y = var('x y')
>>> __tmp__=var("x,y"); c1 = symbolic_expression((x-
↪Integer(5))**Integer(2)+y**Integer(2)-Integer(16)).function(x,y)
>>> __tmp__=var("x,y"); c2 = symbolic_expression((y-
↪Integer(3))**Integer(2)+x**Integer(2)-Integer(9)).function(x,y)
>>> solve([c1(x,y),c2(x,y)],[x,y])
[[x == -9/68*sqrt(55) + 135/68, y == -15/68*sqrt(55) + 123/68],
 [x == 9/68*sqrt(55) + 135/68, y == 15/68*sqrt(55) + 123/68]]
```

We use SymPy for Diophantine equations, see `Expression.solve_diophantine`:

```
sage: assume(x, 'integer')
sage: assume(z, 'integer')
sage: solve((x-z)^2==2, x)
[]

sage: forget()
```

```
>>> from sage.all import *
>>> assume(x, 'integer')
>>> assume(z, 'integer')
>>> solve((x-z)**Integer(2)==Integer(2), x)
[]

>>> forget()
```

The following shows some more of SymPy's capabilities that cannot be handled by Maxima:

```
sage: _ = var('t')
sage: r = solve([x^2 - y^2/exp(x), y-1], x, y, algorithm='sympy')
sage: (r[0][x], r[0][y])
(2*lambert_w(-1/2), 1)
sage: solve(-2*x**3 + 4*x**2 - 2*x + 6 > 0, x, algorithm='sympy')
[x < 1/3*(1/2)^(1/3)*(9*sqrt(77) + 79)^(1/3) + 2/3*(1/2)^(2/3)/(9*sqrt(77) + 79)^
↪(1/3) + 2/3]
sage: solve(sqrt(2*x^2 - 7) - (3 - x),x,algorithm='sympy')
[x == -8, x == 2]
sage: solve(sqrt(2*x + 9) - sqrt(x + 1) - sqrt(x + 4),x,algorithm='sympy')
[x == 0]
sage: r = solve([x + y + z + t, -z - t], x, y, z, t, algorithm='sympy')
sage: (r[0][x], r[0][z])
(-y, -t)
sage: r = solve([x^2+y+z, y+x^2+z, x+y+z^2], x, y,z, algorithm='sympy')
sage: (r[0][x], r[0][y])
(z, -(z + 1)*z)
```

(continues on next page)

(continued from previous page)

```
sage: (r[1][x], r[1][y])
(-z + 1, -z^2 + z - 1)
sage: solve(abs(x + 3) - 2*abs(x - 3),x,algorithm='sympy',domain='real')
[x == 1, x == 9]
```

```
>>> from sage.all import *
>>> _ = var('t')
>>> r = solve([x**Integer(2) - y**Integer(2)/exp(x), y-Integer(1)], x, y,␣
↪algorithm='sympy')
>>> (r[Integer(0)][x], r[Integer(0)][y])
(2*lambert_w(-1/2), 1)
>>> solve(-Integer(2)*x**Integer(3) + Integer(4)*x**Integer(2) - Integer(2)*x +␣
↪Integer(6) > Integer(0), x, algorithm='sympy')
[x < 1/3*(1/2)^(1/3)*(9*sqrt(77) + 79)^(1/3) + 2/3*(1/2)^(2/3)/(9*sqrt(77) + 79)^
↪(1/3) + 2/3]
>>> solve(sqrt(Integer(2)*x**Integer(2) - Integer(7)) - (Integer(3) - x),x,
↪algorithm='sympy')
[x == -8, x == 2]
>>> solve(sqrt(Integer(2)*x + Integer(9)) - sqrt(x + Integer(1)) - sqrt(x +␣
↪Integer(4)),x,algorithm='sympy')
[x == 0]
>>> r = solve([x + y + z + t, -z - t], x, y, z, t, algorithm='sympy')
>>> (r[Integer(0)][x], r[Integer(0)][z])
(-y, -t)
>>> r = solve([x**Integer(2)+y+z, y+x**Integer(2)+z, x+y+z**Integer(2)], x, y,z,␣
↪algorithm='sympy')
>>> (r[Integer(0)][x], r[Integer(0)][y])
(z, -(z + 1)*z)
>>> (r[Integer(1)][x], r[Integer(1)][y])
(-z + 1, -z^2 + z - 1)
>>> solve(abs(x + Integer(3)) - Integer(2)*abs(x - Integer(3)),x,algorithm='sympy
↪',domain='real')
[x == 1, x == 9]
```

We cannot translate all results from SymPy but we can at least print them:

```
sage: solve(sinh(x) - 2*cosh(x),x,algorithm='sympy')
[ImageSet(Lambda(_n, I*(2*_n*pi + pi/2) + log(sqrt(3))), Integers),
 ImageSet(Lambda(_n, I*(2*_n*pi - pi/2) + log(sqrt(3))), Integers)]
sage: solve(2*sin(x) - 2*sin(2*x), x,algorithm='sympy')
[ImageSet(Lambda(_n, 2*_n*pi), Integers),
 ImageSet(Lambda(_n, 2*_n*pi + pi), Integers),
 ImageSet(Lambda(_n, 2*_n*pi + 5*pi/3), Integers),
 ImageSet(Lambda(_n, 2*_n*pi + pi/3), Integers)]

sage: solve(x^5 + 3*x^3 + 7, x, algorithm='sympy')[0] # known bug
complex_root_of(x^5 + 3*x^3 + 7, 0)
```

```
>>> from sage.all import *
>>> solve(sinh(x) - Integer(2)*cosh(x),x,algorithm='sympy')
[ImageSet(Lambda(_n, I*(2*_n*pi + pi/2) + log(sqrt(3))), Integers),
 ImageSet(Lambda(_n, I*(2*_n*pi - pi/2) + log(sqrt(3))), Integers)]
```

(continues on next page)

(continued from previous page)

```
>>> solve(Integer(2)*sin(x) - Integer(2)*sin(Integer(2)*x), x,algorithm='sympy')
[ImageSet(Lambda(_n, 2*_n*pi), Integers),
 ImageSet(Lambda(_n, 2*_n*pi + pi), Integers),
 ImageSet(Lambda(_n, 2*_n*pi + 5*pi/3), Integers),
 ImageSet(Lambda(_n, 2*_n*pi + pi/3), Integers)]

>>> solve(x**Integer(5) + Integer(3)*x**Integer(3) + Integer(7), x, algorithm=
↪'sympy')[Integer(0)] # known bug
complex_root_of(x^5 + 3*x^3 + 7, 0)
```

A basic interface to Giac is provided:

```
sage: # needs sage.libs.giac
sage: solve([(2/3)^x-2], [x], algorithm='giac')
...[[-log(2)/(log(3) - log(2))]]

sage: # needs sage.libs.giac
sage: f = (sin(x) - 8*cos(x)*sin(x))*(sin(x)^2 + cos(x)) - (2*cos(x)*sin(x) -␣
↪sin(x))*(-2*sin(x)^2 + 2*cos(x)^2 - cos(x))
sage: solve(f, x, algorithm='giac')
...[-2*arctan(sqrt(2)), 0, 2*arctan(sqrt(2)), pi]

sage: # needs sage.libs.giac
sage: x, y = SR.var('x,y')
sage: solve([x+y-4,x*y-3],[x,y],algorithm='giac')
[[1, 3], [3, 1]]
```

```
>>> from sage.all import *
>>> # needs sage.libs.giac
>>> solve([(Integer(2)/Integer(3))**x-Integer(2)], [x], algorithm='giac')
...[[-log(2)/(log(3) - log(2))]]

>>> # needs sage.libs.giac
>>> f = (sin(x) - Integer(8)*cos(x)*sin(x))*(sin(x)**Integer(2) + cos(x)) -␣
↪(Integer(2)*cos(x)*sin(x) - sin(x))*(-Integer(2)*sin(x)**Integer(2) +␣
↪Integer(2)*cos(x)**Integer(2) - cos(x))
>>> solve(f, x, algorithm='giac')
...[-2*arctan(sqrt(2)), 0, 2*arctan(sqrt(2)), pi]

>>> # needs sage.libs.giac
>>> x, y = SR.var('x,y')
>>> solve([x+y-Integer(4),x*y-Integer(3)],[x,y],algorithm='giac')
[[1, 3], [3, 1]]
```

`sage.symbolic.relation.`**`solve_ineq`**`(`*ineq*, *vars=None*`)`

Solve inequalities and systems of inequalities using Maxima. Switches between rational inequalities (sage.symbolic.relation.solve_ineq_rational) and Fourier elimination (sage.symbolic.relation.solve_ineq_fouried). See the documentation of these functions for more details.

INPUT:

- `ineq` – one inequality or a list of inequalities

  Case1: If `ineq` is one equality, then it should be rational expression in one variable. This input is passed to sage.symbolic.relation.solve_ineq_univar function.

Case2: If `ineq` is a list involving one or more inequalities, than the input is passed to sage.symbolic.relation.solve_ineq_fourier function. This function can be used for system of linear inequalities and for some types of nonlinear inequalities. See http://maxima.cvs.sourceforge.net/viewvc/maxima/maxima/share/contrib/fourier_elim/rtest_fourier_elim.mac for a big gallery of problems covered by this algorithm.

- `vars` – (optional) parameter with list of variables. This list is used only if Fourier elimination is used. If omitted or if rational inequality is solved, then variables are determined automatically.

OUTPUT:

- `list` – output is list of solutions as a list of simple inequalities output [A,B,C] means (A or B or C) each A, B, C is again a list and if A=[a,b], then A means (a and b).

EXAMPLES:

```
sage: from sage.symbolic.relation import solve_ineq
```

```
>>> from sage.all import *
>>> from sage.symbolic.relation import solve_ineq
```

Inequalities in one variable. The variable is detected automatically:

```
sage: solve_ineq(x^2-1>3)
[[x < -2], [x > 2]]

sage: solve_ineq(1/(x-1)<=8)
[[x < 1], [x >= (9/8)]]
```

```
>>> from sage.all import *
>>> solve_ineq(x**Integer(2)-Integer(1)>Integer(3))
[[x < -2], [x > 2]]

>>> solve_ineq(Integer(1)/(x-Integer(1))<=Integer(8))
[[x < 1], [x >= (9/8)]]
```

System of inequalities with automatically detected inequalities:

```
sage: y = var('y')
sage: solve_ineq([x-y<0,x+y-3<0],[y,x])
[[x < y, y < -x + 3, x < (3/2)]]
sage: solve_ineq([x-y<0,x+y-3<0],[x,y])
[[x < min(-y + 3, y)]]
```

```
>>> from sage.all import *
>>> y = var('y')
>>> solve_ineq([x-y<Integer(0),x+y-Integer(3)<Integer(0)],[y,x])
[[x < y, y < -x + 3, x < (3/2)]]
>>> solve_ineq([x-y<Integer(0),x+y-Integer(3)<Integer(0)],[x,y])
[[x < min(-y + 3, y)]]
```

Note that although Sage will detect the variables automatically, the order it puts them in may depend on the system, so the following command is only guaranteed to give you one of the above answers:

```
sage: solve_ineq([x-y<0,x+y-3<0])   # random
[[x < y, y < -x + 3, x < (3/2)]]
```

```
>>> from sage.all import *
>>> solve_ineq([x-y<Integer(0),x+y-Integer(3)<Integer(0)])   # random
[[x < y, y < -x + 3, x < (3/2)]]
```

ALGORITHM:

Calls `solve_ineq_fourier` if inequalities are list and `solve_ineq_univar` of the inequality is symbolic expression. See the description of these commands for more details related to the set of inequalities which can be solved. The list is empty if there is no solution.

AUTHORS:

- Robert Marik (01-2010)

`sage.symbolic.relation.`**`solve_ineq_fourier`**(*ineq*, *vars=None*)

Solve system of inequalities using Maxima and Fourier elimination.

Can be used for system of linear inequalities and for some types of nonlinear inequalities. For examples, see the example section below and http://maxima.cvs.sourceforge.net/viewvc/maxima/maxima/share/contrib/fourier_elim/rtest_fourier_elim.mac

INPUT:

- `ineq` – list with system of inequalities
- `vars` – optionally list with variables for Fourier elimination

OUTPUT:

- `list` – output is list of solutions as a list of simple inequalities output [A,B,C] means (A or B or C) each A, B, C is again a list and if A=[a,b], then A means (a and b). The list is empty if there is no solution.

EXAMPLES:

```
sage: from sage.symbolic.relation import solve_ineq_fourier
sage: y = var('y')
sage: solve_ineq_fourier([x+y<9,x-y>4],[x,y])
[[y + 4 < x, x < -y + 9, y < (5/2)]]
sage: solve_ineq_fourier([x+y<9,x-y>4],[y,x])[0][0](x=42)
y < -33

sage: solve_ineq_fourier([x^2>=0])
[[x < +Infinity]]

sage: solve_ineq_fourier([log(x)>log(y)],[x,y])
[[y < x, 0 < y]]
sage: solve_ineq_fourier([log(x)>log(y)],[y,x])
[[0 < y, y < x, 0 < x]]
```

```
>>> from sage.all import *
>>> from sage.symbolic.relation import solve_ineq_fourier
>>> y = var('y')
>>> solve_ineq_fourier([x+y<Integer(9),x-y>Integer(4)],[x,y])
[[y + 4 < x, x < -y + 9, y < (5/2)]]
```

(continues on next page)

(continued from previous page)

```
>>> solve_ineq_fourier([x+y<Integer(9),x-y>Integer(4)],[y,
↪x])[Integer(0)][Integer(0)](x=Integer(42))
y < -33

>>> solve_ineq_fourier([x**Integer(2)>=Integer(0)])
[[x < +Infinity]]

>>> solve_ineq_fourier([log(x)>log(y)],[x,y])
[[y < x, 0 < y]]
>>> solve_ineq_fourier([log(x)>log(y)],[y,x])
[[0 < y, y < x, 0 < x]]
```

Note that different systems will find default variables in different orders, so the following is not tested:

```
sage: solve_ineq_fourier([log(x)>log(y)])  # random (one of the following appears)
[[0 < y, y < x, 0 < x]]
[[y < x, 0 < y]]
```

```
>>> from sage.all import *
>>> solve_ineq_fourier([log(x)>log(y)])  # random (one of the following appears)
[[0 < y, y < x, 0 < x]]
[[y < x, 0 < y]]
```

ALGORITHM:

Calls Maxima command `fourier_elim`

AUTHORS:

- Robert Marik (01-2010)

`sage.symbolic.relation.`**`solve_ineq_univar`**`(`*ineq*`)`

Function solves rational inequality in one variable.

INPUT:

- `ineq` – inequality in one variable

OUTPUT:

- `list` – output is list of solutions as a list of simple inequalities output [A,B,C] means (A or B or C) each A, B, C is again a list and if A=[a,b], then A means (a and b). The list is empty if there is no solution.

EXAMPLES:

```
sage: from sage.symbolic.relation import solve_ineq_univar
sage: solve_ineq_univar(x-1/x>0)
[[x > -1, x < 0], [x > 1]]

sage: solve_ineq_univar(x^2-1/x>0)
[[x < 0], [x > 1]]

sage: solve_ineq_univar((x^3-1)*x<=0)
[[x >= 0, x <= 1]]
```

```
>>> from sage.all import *
>>> from sage.symbolic.relation import solve_ineq_univar
>>> solve_ineq_univar(x-Integer(1)/x>Integer(0))
[[x > -1, x < 0], [x > 1]]

>>> solve_ineq_univar(x**Integer(2)-Integer(1)/x>Integer(0))
[[x < 0], [x > 1]]

>>> solve_ineq_univar((x**Integer(3)-Integer(1))*x<=Integer(0))
[[x >= 0, x <= 1]]
```

ALGORITHM:

Calls Maxima command `solve_rat_ineq`

AUTHORS:

- Robert Marik (01-2010)

`sage.symbolic.relation.`**`solve_mod`**(*eqns*, *modulus*, *solution_dict=False*)

Return all solutions to an equation or list of equations modulo the given integer modulus. Each equation must involve only polynomials in 1 or many variables.

By default the solutions are returned as n-tuples, where n is the number of variables appearing anywhere in the given equations. The variables are in alphabetical order.

INPUT:

- `eqns` – equation or list of equations
- `modulus` – integer
- `solution_dict` – boolean (default: `False`); if `True` or nonzero, return a list of dictionaries containing the solutions. If there are no solutions, return an empty list (rather than a list containing an empty dictionary). Likewise, if there's only a single solution, return a list containing one dictionary with that solution.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: solve_mod([x^2 + 2 == x, x^2 + y == y^2], 14)
[(4, 2), (4, 6), (4, 9), (4, 13)]
sage: solve_mod([x^2 == 1, 4*x  == 11], 15)
[(14,)]
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> solve_mod([x**Integer(2) + Integer(2) == x, x**Integer(2) + y ==␣
↪y**Integer(2)], Integer(14))
[(4, 2), (4, 6), (4, 9), (4, 13)]
>>> solve_mod([x**Integer(2) == Integer(1), Integer(4)*x  == Integer(11)],␣
↪Integer(15))
[(14,)]
```

Fermat's equation modulo 3 with exponent 5:

```
sage: var('x,y,z')
(x, y, z)
sage: solve_mod([x^5 + y^5 == z^5], 3)
[(0, 0, 0), (0, 1, 1), (0, 2, 2), (1, 0, 1), (1, 1, 2), (1, 2, 0), (2, 0, 2), (2,␣
↪1, 0), (2, 2, 1)]
```

```
>>> from sage.all import *
>>> var('x,y,z')
(x, y, z)
>>> solve_mod([x**Integer(5) + y**Integer(5) == z**Integer(5)], Integer(3))
[(0, 0, 0), (0, 1, 1), (0, 2, 2), (1, 0, 1), (1, 1, 2), (1, 2, 0), (2, 0, 2), (2,␣
↪1, 0), (2, 2, 1)]
```

We can solve with respect to a bigger modulus if it consists only of small prime factors:

```
sage: [d] = solve_mod([5*x + y == 3, 2*x - 3*y == 9], 3*5*7*11*19*23*29, solution_
↪dict = True)
sage: d[x]
12915279
sage: d[y]
8610183
```

```
>>> from sage.all import *
>>> [d] = solve_mod([Integer(5)*x + y == Integer(3), Integer(2)*x - Integer(3)*y␣
↪== Integer(9)],␣
↪Integer(3)*Integer(5)*Integer(7)*Integer(11)*Integer(19)*Integer(23)*Integer(29),
↪ solution_dict = True)
>>> d[x]
12915279
>>> d[y]
8610183
```

For cases where there are relatively few solutions and the prime factors are small, this can be efficient even if the modulus itself is large:

```
sage: sorted(solve_mod([x^2 == 41], 10^20))
[(4538602480526452429,), (11445932736758703821,), (38554067263241296179,),
(45461397519473547571,), (54538602480526452429,), (61445932736758703821,),
(88554067263241296179,), (95461397519473547571,)]
```

```
>>> from sage.all import *
>>> sorted(solve_mod([x**Integer(2) == Integer(41)], Integer(10)**Integer(20)))
[(4538602480526452429,), (11445932736758703821,), (38554067263241296179,),
(45461397519473547571,), (54538602480526452429,), (61445932736758703821,),
(88554067263241296179,), (95461397519473547571,)]
```

We solve a simple equation modulo 2:

```
sage: x,y = var('x,y')
sage: solve_mod([x == y], 2)
[(0, 0), (1, 1)]
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> solve_mod([x == y], Integer(2))
[(0, 0), (1, 1)]
```

Warning

The current implementation splits the modulus into prime powers, then naively enumerates all possible solutions (starting modulo primes and then working up through prime powers), and finally combines the solution using the Chinese Remainder Theorem. The interface is good, but the algorithm is very inefficient if the modulus has some larger prime factors! Sage *does* have the ability to do something much faster in certain cases at least by using Groebner basis, linear algebra techniques, etc. But for a lot of toy problems this function as is might be useful. At least it establishes an interface.

sage.symbolic.relation.**string_to_list_of_solutions**(*s*)

Used internally by the symbolic solve command to convert the output of Maxima's solve command to a list of solutions in Sage's symbolic package.

EXAMPLES:

We derive the (monic) quadratic formula:

```
sage: var('x,a,b')
(x, a, b)
sage: solve(x^2 + a*x + b == 0, x)
[x == -1/2*a - 1/2*sqrt(a^2 - 4*b), x == -1/2*a + 1/2*sqrt(a^2 - 4*b)]
```

```
>>> from sage.all import *
>>> var('x,a,b')
(x, a, b)
>>> solve(x**Integer(2) + a*x + b == Integer(0), x)
[x == -1/2*a - 1/2*sqrt(a^2 - 4*b), x == -1/2*a + 1/2*sqrt(a^2 - 4*b)]
```

Behind the scenes when the above is evaluated the function *string_to_list_of_solutions()* is called with input the string s below:

```
sage: s = '[x=-(sqrt(a^2-4*b)+a)/2,x=(sqrt(a^2-4*b)-a)/2]'
sage: sage.symbolic.relation.string_to_list_of_solutions(s)
 [x == -1/2*a - 1/2*sqrt(a^2 - 4*b), x == -1/2*a + 1/2*sqrt(a^2 - 4*b)]
```

```
>>> from sage.all import *
>>> s = '[x=-(sqrt(a^2-4*b)+a)/2,x=(sqrt(a^2-4*b)-a)/2]'
>>> sage.symbolic.relation.string_to_list_of_solutions(s)
 [x == -1/2*a - 1/2*sqrt(a^2 - 4*b), x == -1/2*a + 1/2*sqrt(a^2 - 4*b)]
```

sage.symbolic.relation.**test_relation_maxima**(*relation*)

Return `True` if this (in)equality is definitely true. Return `False` if it is false or the algorithm for testing (in)equality is inconclusive.

EXAMPLES:

```
sage: from sage.symbolic.relation import test_relation_maxima
sage: k = var('k')
sage: pol = 1/(k-1) - 1/k -1/k/(k-1)
sage: test_relation_maxima(pol == 0)
True
sage: f = sin(x)^2 + cos(x)^2 - 1
sage: test_relation_maxima(f == 0)
True
sage: test_relation_maxima( x == x )
True
sage: test_relation_maxima( x != x )
False
sage: test_relation_maxima( x > x )
False
sage: test_relation_maxima( x^2 > x )
False
sage: test_relation_maxima( x + 2 > x )
True
sage: test_relation_maxima( x - 2 > x )
False
```

```
>>> from sage.all import *
>>> from sage.symbolic.relation import test_relation_maxima
>>> k = var('k')
>>> pol = Integer(1)/(k-Integer(1)) - Integer(1)/k -Integer(1)/k/(k-Integer(1))
>>> test_relation_maxima(pol == Integer(0))
True
>>> f = sin(x)**Integer(2) + cos(x)**Integer(2) - Integer(1)
>>> test_relation_maxima(f == Integer(0))
True
>>> test_relation_maxima( x == x )
True
>>> test_relation_maxima( x != x )
False
>>> test_relation_maxima( x > x )
False
>>> test_relation_maxima( x**Integer(2) > x )
False
>>> test_relation_maxima( x + Integer(2) > x )
True
>>> test_relation_maxima( x - Integer(2) > x )
False
```

Here are some examples involving assumptions:

```
sage: x, y, z = var('x, y, z')
sage: assume(x>=y,y>=z,z>=x)
sage: test_relation_maxima(x==z)
True
sage: test_relation_maxima(z<x)
False
sage: test_relation_maxima(z>y)
False
```

(continues on next page)

(continued from previous page)

```
sage: test_relation_maxima(y==z)
True
sage: forget()
sage: assume(x>=1,x<=1)
sage: test_relation_maxima(x==1)
True
sage: test_relation_maxima(x>1)
False
sage: test_relation_maxima(x>=1)
True
sage: test_relation_maxima(x!=1)
False
sage: forget()
sage: assume(x>0)
sage: test_relation_maxima(x==0)
False
sage: test_relation_maxima(x>-1)
True
sage: test_relation_maxima(x!=0)
True
sage: test_relation_maxima(x!=1)
False
sage: forget()
```

```
>>> from sage.all import *
>>> x, y, z = var('x, y, z')
>>> assume(x>=y,y>=z,z>=x)
>>> test_relation_maxima(x==z)
True
>>> test_relation_maxima(z<x)
False
>>> test_relation_maxima(z>y)
False
>>> test_relation_maxima(y==z)
True
>>> forget()
>>> assume(x>=Integer(1),x<=Integer(1))
>>> test_relation_maxima(x==Integer(1))
True
>>> test_relation_maxima(x>Integer(1))
False
>>> test_relation_maxima(x>=Integer(1))
True
>>> test_relation_maxima(x!=Integer(1))
False
>>> forget()
>>> assume(x>Integer(0))
>>> test_relation_maxima(x==Integer(0))
False
>>> test_relation_maxima(x>-Integer(1))
True
>>> test_relation_maxima(x!=Integer(0))
```

(continues on next page)

(continued from previous page)

```
True
>>> test_relation_maxima(x!=Integer(1))
False
>>> forget()
```

2.6 Symbolic Computation

AUTHORS:

- Bobby Moretti and William Stein (2006-2007)
- Robert Bradshaw (2007-10): minpoly(), numerical algorithm
- Robert Bradshaw (2008-10): minpoly(), algebraic algorithm
- Golam Mortuza Hossain (2009-06-15): _limit_latex()
- Golam Mortuza Hossain (2009-06-22): _laplace_latex(), _inverse_laplace_latex()
- Tom Coates (2010-06-11): fixed Issue #9217

EXAMPLES:

The basic units of the calculus package are symbolic expressions which are elements of the symbolic expression ring (SR). To create a symbolic variable object in Sage, use the *var()* function, whose argument is the text of that variable. Note that Sage is intelligent about LaTeXing variable names.

```
sage: x1 = var('x1'); x1
x1
sage: latex(x1)
x_{1}
sage: theta = var('theta'); theta
theta
sage: latex(theta)
\theta
```

```
>>> from sage.all import *
>>> x1 = var('x1'); x1
x1
>>> latex(x1)
x_{1}
>>> theta = var('theta'); theta
theta
>>> latex(theta)
\theta
```

Sage predefines x to be a global indeterminate. Thus the following works:

```
sage: x^2
x^2
sage: type(x)
<class 'sage.symbolic.expression.Expression'>
```

```
>>> from sage.all import *
>>> x**Integer(2)
x^2
>>> type(x)
<class 'sage.symbolic.expression.Expression'>
```

More complicated expressions in Sage can be built up using ordinary arithmetic. The following are valid, and follow the rules of Python arithmetic: (The '=' operator represents assignment, and not equality)

```
sage: var('x,y,z')
(x, y, z)
sage: f = x + y + z/(2*sin(y*z/55))
sage: g = f^f; g
(x + y + 1/2*z/sin(1/55*y*z))^(x + y + 1/2*z/sin(1/55*y*z))
```

```
>>> from sage.all import *
>>> var('x,y,z')
(x, y, z)
>>> f = x + y + z/(Integer(2)*sin(y*z/Integer(55)))
>>> g = f**f; g
(x + y + 1/2*z/sin(1/55*y*z))^(x + y + 1/2*z/sin(1/55*y*z))
```

Differentiation and integration are available, but behind the scenes through Maxima:

```
sage: f = sin(x)/cos(2*y)
sage: f.derivative(y)
2*sin(x)*sin(2*y)/cos(2*y)^2
sage: g = f.integral(x); g
-cos(x)/cos(2*y)
```

```
>>> from sage.all import *
>>> f = sin(x)/cos(Integer(2)*y)
>>> f.derivative(y)
2*sin(x)*sin(2*y)/cos(2*y)^2
>>> g = f.integral(x); g
-cos(x)/cos(2*y)
```

Note that these methods usually require an explicit variable name. If none is given, Sage will try to find one for you.

```
sage: f = sin(x); f.derivative()
cos(x)
```

```
>>> from sage.all import *
>>> f = sin(x); f.derivative()
cos(x)
```

If the expression is a callable symbolic expression (i.e., the variable order is specified), then Sage can calculate the matrix derivative (i.e., the gradient, Jacobian matrix, etc.) if no variables are specified. In the example below, we use the second derivative test to determine that there is a saddle point at (0,-1/2).

```
sage: f(x,y) = x^2*y + y^2 + y
sage: f.diff()  # gradient
(x, y) |--> (2*x*y, x^2 + 2*y + 1)
```

(continues on next page)

(continued from previous page)

```
sage: solve(list(f.diff()), [x,y])
[[x == -I, y == 0], [x == I, y == 0], [x == 0, y == (-1/2)]]
sage: H=f.diff(2); H  # Hessian matrix
[(x, y) |--> 2*y (x, y) |--> 2*x]
[(x, y) |--> 2*x   (x, y) |--> 2]
sage: H(x=0, y=-1/2)
[-1  0]
[ 0  2]
sage: H(x=0, y=-1/2).eigenvalues()
[-1, 2]
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(x**Integer(2)*y + y**Integer(2) + y).
↪function(x,y)
>>> f.diff()  # gradient
(x, y) |--> (2*x*y, x^2 + 2*y + 1)
>>> solve(list(f.diff()), [x,y])
[[x == -I, y == 0], [x == I, y == 0], [x == 0, y == (-1/2)]]
>>> H=f.diff(Integer(2)); H  # Hessian matrix
[(x, y) |--> 2*y (x, y) |--> 2*x]
[(x, y) |--> 2*x   (x, y) |--> 2]
>>> H(x=Integer(0), y=-Integer(1)/Integer(2))
[-1  0]
[ 0  2]
>>> H(x=Integer(0), y=-Integer(1)/Integer(2)).eigenvalues()
[-1, 2]
```

Here we calculate the Jacobian for the polar coordinate transformation:

```
sage: T(r,theta) = [r*cos(theta),r*sin(theta)]
sage: T
(r, theta) |--> (r*cos(theta), r*sin(theta))
sage: T.diff() # Jacobian matrix
[   (r, theta) |--> cos(theta) (r, theta) |--> -r*sin(theta)]
[   (r, theta) |--> sin(theta)  (r, theta) |--> r*cos(theta)]
sage: diff(T) # Jacobian matrix
[   (r, theta) |--> cos(theta) (r, theta) |--> -r*sin(theta)]
[   (r, theta) |--> sin(theta)  (r, theta) |--> r*cos(theta)]
sage: T.diff().det() # Jacobian
(r, theta) |--> r*cos(theta)^2 + r*sin(theta)^2
```

```
>>> from sage.all import *
>>> __tmp__=var("r,theta"); T = symbolic_expression([r*cos(theta),r*sin(theta)]).
↪function(r,theta)
>>> T
(r, theta) |--> (r*cos(theta), r*sin(theta))
>>> T.diff() # Jacobian matrix
[   (r, theta) |--> cos(theta) (r, theta) |--> -r*sin(theta)]
[   (r, theta) |--> sin(theta)  (r, theta) |--> r*cos(theta)]
>>> diff(T) # Jacobian matrix
[   (r, theta) |--> cos(theta) (r, theta) |--> -r*sin(theta)]
[   (r, theta) |--> sin(theta)  (r, theta) |--> r*cos(theta)]
```

(continues on next page)

(continued from previous page)

```
>>> T.diff().det() # Jacobian
(r, theta) |--> r*cos(theta)^2 + r*sin(theta)^2
```

When the order of variables is ambiguous, Sage will raise an exception when differentiating:

```
sage: f = sin(x+y); f.derivative()
Traceback (most recent call last):
...
ValueError: No differentiation variable specified.
```

```
>>> from sage.all import *
>>> f = sin(x+y); f.derivative()
Traceback (most recent call last):
...
ValueError: No differentiation variable specified.
```

Simplifying symbolic sums is also possible, using the `sum()` command, which also uses Maxima in the background:

```
sage: k, m = var('k, m')
sage: sum(1/k^4, k, 1, oo)
1/90*pi^4
sage: sum(binomial(m,k), k, 0, m)
2^m
```

```
>>> from sage.all import *
>>> k, m = var('k, m')
>>> sum(Integer(1)/k**Integer(4), k, Integer(1), oo)
1/90*pi^4
>>> sum(binomial(m,k), k, Integer(0), m)
2^m
```

Symbolic matrices can be used as well in various ways, including exponentiation:

```
sage: M = matrix([[x,x^2],[1/x,x]])
sage: M^2
[x^2 + x   2*x^3]
[      2 x^2 + x]
sage: e^M
[           1/2*(e^(2*sqrt(x)) + 1)*e^(x - sqrt(x)) 1/2*(x*e^(2*sqrt(x)) -␣
↪x)*sqrt(x)*e^(x - sqrt(x))]
[  1/2*(e^(2*sqrt(x)) - 1)*e^(x - sqrt(x))/x^(3/2)            1/2*(e^(2*sqrt(x)) +␣
↪1)*e^(x - sqrt(x))]
```

```
>>> from sage.all import *
>>> M = matrix([[x,x**Integer(2)],[Integer(1)/x,x]])
>>> M**Integer(2)
[x^2 + x   2*x^3]
[      2 x^2 + x]
>>> e**M
[           1/2*(e^(2*sqrt(x)) + 1)*e^(x - sqrt(x)) 1/2*(x*e^(2*sqrt(x)) -␣
↪x)*sqrt(x)*e^(x - sqrt(x))]
```

(continues on next page)

(continued from previous page)

```
[  1/2*(e^(2*sqrt(x)) - 1)*e^(x - sqrt(x))/x^(3/2)           1/2*(e^(2*sqrt(x)) +␣
↪1)*e^(x - sqrt(x))]
```

Complex exponentiation works, but may require a patched version of maxima (Issue #32898) for now:

```
sage: M = i*matrix([[pi]])
sage: e^M  # not tested, requires patched maxima
[-1]
sage: M = i*matrix([[pi,0],[0,2*pi]])
sage: e^M
[-1  0]
[ 0  1]
sage: M = matrix([[0,pi],[-pi,0]])
sage: e^M
[-1  0]
[ 0 -1]
```

```
>>> from sage.all import *
>>> M = i*matrix([[pi]])
>>> e**M  # not tested, requires patched maxima
[-1]
>>> M = i*matrix([[pi,Integer(0)],[Integer(0),Integer(2)*pi]])
>>> e**M
[-1  0]
[ 0  1]
>>> M = matrix([[Integer(0),pi],[-pi,Integer(0)]])
>>> e**M
[-1  0]
[ 0 -1]
```

Substitution works similarly. We can substitute with a python dict:

```
sage: f = sin(x*y - z)
sage: f({x: var('t'), y: z})
sin(t*z - z)
```

```
>>> from sage.all import *
>>> f = sin(x*y - z)
>>> f({x: var('t'), y: z})
sin(t*z - z)
```

Also we can substitute with keywords:

```
sage: f = sin(x*y - z)
sage: f(x=t, y=z)
sin(t*z - z)
```

```
>>> from sage.all import *
>>> f = sin(x*y - z)
>>> f(x=t, y=z)
sin(t*z - z)
```

Another example:

```
sage: f = sin(2*pi*x/y)
sage: f(x=4)
sin(8*pi/y)
```

```
>>> from sage.all import *
>>> f = sin(Integer(2)*pi*x/y)
>>> f(x=Integer(4))
sin(8*pi/y)
```

It is no longer allowed to call expressions with positional arguments:

```
sage: f = sin(x)
sage: f(y)
Traceback (most recent call last):
...
TypeError: Substitution using function-call syntax and unnamed
arguments has been removed. You can use named arguments instead, like
EXPR(x=..., y=...)
sage: f(x=pi)
0
```

```
>>> from sage.all import *
>>> f = sin(x)
>>> f(y)
Traceback (most recent call last):
...
TypeError: Substitution using function-call syntax and unnamed
arguments has been removed. You can use named arguments instead, like
EXPR(x=..., y=...)
>>> f(x=pi)
0
```

We can also make a `CallableSymbolicExpression`, which is a `SymbolicExpression` that is a function of specified variables in a fixed order. Each `SymbolicExpression` has a `function(...)` method that is used to create a `CallableSymbolicExpression`, as illustrated below:

```
sage: u = log((2-x)/(y+5))
sage: f = u.function(x, y); f
(x, y) |--> log(-(x - 2)/(y + 5))
```

```
>>> from sage.all import *
>>> u = log((Integer(2)-x)/(y+Integer(5)))
>>> f = u.function(x, y); f
(x, y) |--> log(-(x - 2)/(y + 5))
```

There is an easier way of creating a `CallableSymbolicExpression`, which relies on the Sage preparser.

```
sage: f(x,y) = log(x)*cos(y); f
(x, y) |--> cos(y)*log(x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(log(x)*cos(y)).function(x,y); f
(x, y) |--> cos(y)*log(x)
```

Then we have fixed an order of variables and there is no ambiguity substituting or evaluating:

```
sage: f(x,y) = log((2-x)/(y+5))
sage: f(7,t)
log(-5/(t + 5))
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(log((Integer(2)-x)/(y+Integer(5)))).
↪function(x,y)
>>> f(Integer(7),t)
log(-5/(t + 5))
```

Some further examples:

```
sage: f = 5*sin(x)
sage: f
5*sin(x)
sage: f(x=2)
5*sin(2)
sage: f(x=pi)
0
sage: float(f(x=pi))
0.0
```

```
>>> from sage.all import *
>>> f = Integer(5)*sin(x)
>>> f
5*sin(x)
>>> f(x=Integer(2))
5*sin(2)
>>> f(x=pi)
0
>>> float(f(x=pi))
0.0
```

Another example:

```
sage: f = integrate(1/sqrt(9+x^2), x); f
arcsinh(1/3*x)
sage: f(x=3)
arcsinh(1)
sage: f.derivative(x)
1/sqrt(x^2 + 9)
```

```
>>> from sage.all import *
>>> f = integrate(Integer(1)/sqrt(Integer(9)+x**Integer(2)), x); f
arcsinh(1/3*x)
>>> f(x=Integer(3))
arcsinh(1)
>>> f.derivative(x)
1/sqrt(x^2 + 9)
```

We compute the length of the parabola from 0 to 2:

```
sage: x = var('x')
sage: y = x^2
sage: dy = derivative(y,x)
sage: z = integral(sqrt(1 + dy^2), x, 0, 2)
sage: z
sqrt(17) + 1/4*arcsinh(4)
sage: n(z,200)
4.6467837624329358733826155674904591885104869874232887508703
sage: float(z)
4.646783762432936
```

```
>>> from sage.all import *
>>> x = var('x')
>>> y = x**Integer(2)
>>> dy = derivative(y,x)
>>> z = integral(sqrt(Integer(1) + dy**Integer(2)), x, Integer(0), Integer(2))
>>> z
sqrt(17) + 1/4*arcsinh(4)
>>> n(z,Integer(200))
4.6467837624329358733826155674904591885104869874232887508703
>>> float(z)
4.646783762432936
```

We test pickling:

```
sage: x, y = var('x,y')
sage: f = -sqrt(pi)*(x^3 + sin(x/cos(y)))
sage: bool(loads(dumps(f)) == f)
True
```

```
>>> from sage.all import *
>>> x, y = var('x,y')
>>> f = -sqrt(pi)*(x**Integer(3) + sin(x/cos(y)))
>>> bool(loads(dumps(f)) == f)
True
```

Coercion examples:

We coerce various symbolic expressions into the complex numbers:

```
sage: CC(I)
1.00000000000000*I
sage: CC(2*I)
2.00000000000000*I
sage: ComplexField(200)(2*I)
2.0000000000000000000000000000000000000000000000000000000000*I
sage: ComplexField(200)(sin(I))
1.1752011936438014568823818505956008151557179813340958702296*I
sage: f = sin(I) + cos(I/2); f
cosh(1/2) + I*sinh(1)
sage: CC(f)
1.12762596520638 + 1.17520119364380*I
sage: ComplexField(200)(f)
```

(continues on next page)

(continued from previous page)

```
1.1276259652063807852262251614026720125478471180986674836290
 + 1.1752011936438014568823818505956008151557179813340958702296*I
sage: ComplexField(100)(f)
1.1276259652063807852262251614 + 1.1752011936438014568823818506*I
```

```
>>> from sage.all import *
>>> CC(I)
1.00000000000000*I
>>> CC(Integer(2)*I)
2.00000000000000*I
>>> ComplexField(Integer(200))(Integer(2)*I)
2.0000000000000000000000000000000000000000000000000000000000*I
>>> ComplexField(Integer(200))(sin(I))
1.1752011936438014568823818505956008151557179813340958702296*I
>>> f = sin(I) + cos(I/Integer(2)); f
cosh(1/2) + I*sinh(1)
>>> CC(f)
1.12762596520638 + 1.17520119364380*I
>>> ComplexField(Integer(200))(f)
1.1276259652063807852262251614026720125478471180986674836290
 + 1.1752011936438014568823818505956008151557179813340958702296*I
>>> ComplexField(Integer(100))(f)
1.1276259652063807852262251614 + 1.1752011936438014568823818506*I
```

We illustrate construction of an inverse sum where each denominator has a new variable name:

```
sage: f = sum(1/var('n%s'%i)^i for i in range(10))
sage: f
1/n1 + 1/n2^2 + 1/n3^3 + 1/n4^4 + 1/n5^5 + 1/n6^6 + 1/n7^7 + 1/n8^8 + 1/n9^9 + 1
```

```
>>> from sage.all import *
>>> f = sum(Integer(1)/var('n%s'%i)**i for i in range(Integer(10)))
>>> f
1/n1 + 1/n2^2 + 1/n3^3 + 1/n4^4 + 1/n5^5 + 1/n6^6 + 1/n7^7 + 1/n8^8 + 1/n9^9 + 1
```

Note that after calling var, the variables are immediately available for use:

```
sage: (n1 + n2)^5
(n1 + n2)^5
```

```
>>> from sage.all import *
>>> (n1 + n2)**Integer(5)
(n1 + n2)^5
```

We can, of course, substitute:

```
sage: f(n9=9, n7=n6)
1/n1 + 1/n2^2 + 1/n3^3 + 1/n4^4 + 1/n5^5 + 1/n6^6 + 1/n6^7 + 1/n8^8
 + 387420490/387420489
```

```
>>> from sage.all import *
>>> f(n9=Integer(9), n7=n6)
```

(continues on next page)

(continued from previous page)

```
1/n1 + 1/n2^2 + 1/n3^3 + 1/n4^4 + 1/n5^5 + 1/n6^6 + 1/n6^7 + 1/n8^8
 + 387420490/387420489
```

sage.calculus.calculus.**at**(*ex*, **args*, ***kwds*)

Parses at formulations from other systems, such as Maxima. Replaces evaluation 'at' a point with substitution method of a symbolic expression.

EXAMPLES:

We do not import at at the top level, but we can use it as a synonym for substitution if we import it:

```
sage: g = x^3 - 3
sage: from sage.calculus.calculus import at
sage: at(g, x=1)
-2
sage: g.subs(x=1)
-2
```

```
>>> from sage.all import *
>>> g = x**Integer(3) - Integer(3)
>>> from sage.calculus.calculus import at
>>> at(g, x=Integer(1))
-2
>>> g.subs(x=Integer(1))
-2
```

We find a formal Taylor expansion:

```
sage: h,x = var('h,x')
sage: u = function('u')
sage: u(x + h)
u(h + x)
sage: diff(u(x+h), x)
D[0](u)(h + x)
sage: taylor(u(x+h), h, 0, 4)
1/24*h^4*diff(u(x), x, x, x, x) + 1/6*h^3*diff(u(x), x, x, x)
 + 1/2*h^2*diff(u(x), x, x) + h*diff(u(x), x) + u(x)
```

```
>>> from sage.all import *
>>> h,x = var('h,x')
>>> u = function('u')
>>> u(x + h)
u(h + x)
>>> diff(u(x+h), x)
D[0](u)(h + x)
>>> taylor(u(x+h), h, Integer(0), Integer(4))
1/24*h^4*diff(u(x), x, x, x, x) + 1/6*h^3*diff(u(x), x, x, x)
 + 1/2*h^2*diff(u(x), x, x) + h*diff(u(x), x) + u(x)
```

We compute a Laplace transform:

```
sage: var('s,t')
(s, t)
```

(continues on next page)

(continued from previous page)

```
sage: f = function('f')(t)
sage: f.diff(t, 2)
diff(f(t), t, t)
sage: f.diff(t,2).laplace(t,s)
s^2*laplace(f(t), t, s) - s*f(0) - D[0](f)(0)
```

```
>>> from sage.all import *
>>> var('s,t')
(s, t)
>>> f = function('f')(t)
>>> f.diff(t, Integer(2))
diff(f(t), t, t)
>>> f.diff(t,Integer(2)).laplace(t,s)
s^2*laplace(f(t), t, s) - s*f(0) - D[0](f)(0)
```

We can also accept a non-keyword list of expression substitutions, like Maxima does (Issue #12796):

```
sage: from sage.calculus.calculus import at
sage: f = function('f')
sage: at(f(x), [x == 1])
f(1)
```

```
>>> from sage.all import *
>>> from sage.calculus.calculus import at
>>> f = function('f')
>>> at(f(x), [x == Integer(1)])
f(1)
```

sage.calculus.calculus.**dummy_diff**(**args*)

This function is called when 'diff' appears in a Maxima string.

EXAMPLES:

```
sage: from sage.calculus.calculus import dummy_diff
sage: x,y = var('x,y')
sage: dummy_diff(sin(x*y), x, SR(2), y, SR(1))
-x*y^2*cos(x*y) - 2*y*sin(x*y)
```

```
>>> from sage.all import *
>>> from sage.calculus.calculus import dummy_diff
>>> x,y = var('x,y')
>>> dummy_diff(sin(x*y), x, SR(Integer(2)), y, SR(Integer(1)))
-x*y^2*cos(x*y) - 2*y*sin(x*y)
```

Here the function is used implicitly:

```
sage: a = var('a')
sage: f = function('cr')(a)
sage: g = f.diff(a); g
diff(cr(a), a)
```

```
>>> from sage.all import *
>>> a = var('a')
>>> f = function('cr')(a)
>>> g = f.diff(a); g
diff(cr(a), a)
```

sage.calculus.calculus.**dummy_integrate**(**args*)

This function is called to create formal wrappers of integrals that Maxima can't compute:

EXAMPLES:

```
sage: from sage.calculus.calculus import dummy_integrate
sage: f = function('f')
sage: dummy_integrate(f(x), x)
integrate(f(x), x)
sage: a,b = var('a,b')
sage: dummy_integrate(f(x), x, a, b)
integrate(f(x), x, a, b)
```

```
>>> from sage.all import *
>>> from sage.calculus.calculus import dummy_integrate
>>> f = function('f')
>>> dummy_integrate(f(x), x)
integrate(f(x), x)
>>> a,b = var('a,b')
>>> dummy_integrate(f(x), x, a, b)
integrate(f(x), x, a, b)
```

sage.calculus.calculus.**dummy_inverse_laplace**(**args*)

This function is called to create formal wrappers of inverse Laplace transforms that Maxima can't compute:

EXAMPLES:

```
sage: from sage.calculus.calculus import dummy_inverse_laplace
sage: s,t = var('s,t')
sage: F = function('F')
sage: dummy_inverse_laplace(F(s), s, t)
ilt(F(s), s, t)
```

```
>>> from sage.all import *
>>> from sage.calculus.calculus import dummy_inverse_laplace
>>> s,t = var('s,t')
>>> F = function('F')
>>> dummy_inverse_laplace(F(s), s, t)
ilt(F(s), s, t)
```

sage.calculus.calculus.**dummy_laplace**(**args*)

This function is called to create formal wrappers of laplace transforms that Maxima can't compute:

EXAMPLES:

```
sage: from sage.calculus.calculus import dummy_laplace
sage: s,t = var('s,t')
```

(continues on next page)

(continued from previous page)

```
sage: f = function('f')
sage: dummy_laplace(f(t), t, s)
laplace(f(t), t, s)
```

```
>>> from sage.all import *
>>> from sage.calculus.calculus import dummy_laplace
>>> s,t = var('s,t')
>>> f = function('f')
>>> dummy_laplace(f(t), t, s)
laplace(f(t), t, s)
```

sage.calculus.calculus.**dummy_pochhammer**(**args*)

This function is called to create formal wrappers of Pochhammer symbols.

EXAMPLES:

```
sage: from sage.calculus.calculus import dummy_pochhammer
sage: s,t = var('s,t')
sage: dummy_pochhammer(s, t)
gamma(s + t)/gamma(s)
```

```
>>> from sage.all import *
>>> from sage.calculus.calculus import dummy_pochhammer
>>> s,t = var('s,t')
>>> dummy_pochhammer(s, t)
gamma(s + t)/gamma(s)
```

sage.calculus.calculus.**inverse_laplace**(*ex*, *s*, *t*, *algorithm='maxima'*)

Return the inverse Laplace transform with respect to the variable t and transform parameter s, if possible.

If this function cannot find a solution, a formal function is returned. The function that is returned may be viewed as a function of t.

DEFINITION:

The inverse Laplace transform of a function $F(s)$ is the function $f(t)$, defined by

$$f(t) = \frac{1}{2\pi i} \int_{\gamma-i\infty}^{\gamma+i\infty} e^{st} F(s) ds,$$

where γ is chosen so that the contour path of integration is in the region of convergence of $F(s)$.

INPUT:

- `ex` – a symbolic expression
- `s` – transform parameter
- `t` – independent variable
- `algorithm` – (default: `'maxima'`) one of
  - `'maxima'` – use Maxima (the default)
  - `'sympy'` – use SymPy
  - `'giac'` – use Giac (optional)

See also

laplace()

EXAMPLES:

```
sage: var('w, m')
(w, m)
sage: f = (1/(w^2+10)).inverse_laplace(w, m); f
1/10*sqrt(10)*sin(sqrt(10)*m)
sage: laplace(f, m, w)
1/(w^2 + 10)

sage: f(t) = t*cos(t)
sage: s = var('s')
sage: L = laplace(f, t, s); L
t |--> 2*s^2/(s^2 + 1)^2 - 1/(s^2 + 1)
sage: inverse_laplace(L, s, t)
t |--> t*cos(t)
sage: inverse_laplace(1/(s^3+1), s, t)
1/3*(sqrt(3)*sin(1/2*sqrt(3)*t) - cos(1/2*sqrt(3)*t))*e^(1/2*t) + 1/3*e^(-t)
```

```
>>> from sage.all import *
>>> var('w, m')
(w, m)
>>> f = (Integer(1)/(w**Integer(2)+Integer(10))).inverse_laplace(w, m); f
1/10*sqrt(10)*sin(sqrt(10)*m)
>>> laplace(f, m, w)
1/(w^2 + 10)

>>> __tmp__=var("t"); f = symbolic_expression(t*cos(t)).function(t)
>>> s = var('s')
>>> L = laplace(f, t, s); L
t |--> 2*s^2/(s^2 + 1)^2 - 1/(s^2 + 1)
>>> inverse_laplace(L, s, t)
t |--> t*cos(t)
>>> inverse_laplace(Integer(1)/(s**Integer(3)+Integer(1)), s, t)
1/3*(sqrt(3)*sin(1/2*sqrt(3)*t) - cos(1/2*sqrt(3)*t))*e^(1/2*t) + 1/3*e^(-t)
```

No explicit inverse Laplace transform, so one is returned formally a function `ilt`:

```
sage: inverse_laplace(cos(s), s, t)
ilt(cos(s), s, t)
```

```
>>> from sage.all import *
>>> inverse_laplace(cos(s), s, t)
ilt(cos(s), s, t)
```

Transform an expression involving a time-shift, via SymPy:

```
sage: inverse_laplace(1/s^2*exp(-s), s, t, algorithm='sympy').simplify()
(t - 1)*heaviside(t - 1)
```

```
>>> from sage.all import *
>>> inverse_laplace(Integer(1)/s**Integer(2)*exp(-s), s, t, algorithm='sympy').
↪simplify()
(t - 1)*heaviside(t - 1)
```

The same instance with Giac:

```
sage: # needs giac
sage: inverse_laplace(1/s^2*exp(-s), s, t, algorithm='giac')
(t - 1)*heaviside(t - 1)
```

```
>>> from sage.all import *
>>> # needs giac
>>> inverse_laplace(Integer(1)/s**Integer(2)*exp(-s), s, t, algorithm='giac')
(t - 1)*heaviside(t - 1)
```

Transform a rational expression:

```
sage: # needs giac
sage: inverse_laplace((2*s^2*exp(-2*s) - exp(-s))/(s^3+1), s, t,
....:                 algorithm='giac')
-1/3*(sqrt(3)*e^(1/2*t - 1/2)*sin(1/2*sqrt(3)*(t - 1))
 - cos(1/2*sqrt(3)*(t - 1))*e^(1/2*t - 1/2) + e^(-t + 1))*heaviside(t - 1)
 + 2/3*(2*cos(1/2*sqrt(3)*(t - 2))*e^(1/2*t - 1) + e^(-t + 2))*heaviside(t - 2)

sage: inverse_laplace(1/(s - 1), s, x)
e^x
```

```
>>> from sage.all import *
>>> # needs giac
>>> inverse_laplace((Integer(2)*s**Integer(2)*exp(-Integer(2)*s) - exp(-s))/
↪(s**Integer(3)+Integer(1)), s, t,
...                 algorithm='giac')
-1/3*(sqrt(3)*e^(1/2*t - 1/2)*sin(1/2*sqrt(3)*(t - 1))
 - cos(1/2*sqrt(3)*(t - 1))*e^(1/2*t - 1/2) + e^(-t + 1))*heaviside(t - 1)
 + 2/3*(2*cos(1/2*sqrt(3)*(t - 2))*e^(1/2*t - 1) + e^(-t + 2))*heaviside(t - 2)

>>> inverse_laplace(Integer(1)/(s - Integer(1)), s, x)
e^x
```

The inverse Laplace transform of a constant is a delta distribution:

```
sage: inverse_laplace(1, s, t)
dirac_delta(t)
sage: inverse_laplace(1, s, t, algorithm='sympy')
dirac_delta(t)
sage: inverse_laplace(1, s, t, algorithm='giac')  # needs giac
dirac_delta(t)
```

```
>>> from sage.all import *
>>> inverse_laplace(Integer(1), s, t)
dirac_delta(t)
```

(continues on next page)

(continued from previous page)

```
>>> inverse_laplace(Integer(1), s, t, algorithm='sympy')
dirac_delta(t)
>>> inverse_laplace(Integer(1), s, t, algorithm='giac')  # needs giac
dirac_delta(t)
```

sage.calculus.calculus.**laplace**(*ex*, *t*, *s*, *algorithm='maxima'*)

Return the Laplace transform with respect to the variable t and transform parameter s, if possible.

If this function cannot find a solution, a formal function is returned. The function that is returned may be viewed as a function of s.

DEFINITION:

The Laplace transform of a function $f(t)$, defined for all real numbers $t \geq 0$, is the function $F(s)$ defined by

$$F(s) = \int_0^\infty e^{-st} f(t) dt.$$

INPUT:

- `ex` – a symbolic expression
- `t` – independent variable
- `s` – transform parameter
- `algorithm` – (default: `'maxima'`) one of
  - `'maxima'` – use Maxima (the default)
  - `'sympy'` – use SymPy
  - `'giac'` – use Giac (optional)

Note

The `'sympy'` algorithm returns the tuple (F, a, `cond`) where F is the Laplace transform of $f(t)$, $Re(s) > a$ is the half-plane of convergence, and `cond` are auxiliary convergence conditions.

See also

inverse_laplace()

EXAMPLES:

We compute a few Laplace transforms:

```
sage: var('x, s, z, t, t0')
(x, s, z, t, t0)
sage: sin(x).laplace(x, s)
1/(s^2 + 1)
sage: (z + exp(x)).laplace(x, s)
z/s + 1/(s - 1)
sage: log(t/t0).laplace(t, s)
-(euler_gamma + log(s) + log(t0))/s
```

```
>>> from sage.all import *
>>> var('x, s, z, t, t0')
(x, s, z, t, t0)
>>> sin(x).laplace(x, s)
1/(s^2 + 1)
>>> (z + exp(x)).laplace(x, s)
z/s + 1/(s - 1)
>>> log(t/t0).laplace(t, s)
-(euler_gamma + log(s) + log(t0))/s
```

We do a formal calculation:

```
sage: f = function('f')(x)
sage: g = f.diff(x); g
diff(f(x), x)
sage: g.laplace(x, s)
s*laplace(f(x), x, s) - f(0)
```

```
>>> from sage.all import *
>>> f = function('f')(x)
>>> g = f.diff(x); g
diff(f(x), x)
>>> g.laplace(x, s)
s*laplace(f(x), x, s) - f(0)
```

A BATTLE BETWEEN the X-women and the Y-men (by David Joyner): Solve

$$x' = -16y, x(0) = 270, y' = -x + 1, y(0) = 90.$$

This models a fight between two sides, the "X-women" and the "Y-men", where the X-women have 270 initially and the Y-men have 90, but the Y-men are better at fighting, because of the higher factor of "-16" vs "-1", and also get an occasional reinforcement, because of the "+1" term.

```
sage: var('t')
t
sage: t = var('t')
sage: x = function('x')(t)
sage: y = function('y')(t)
sage: de1 = x.diff(t) + 16*y
sage: de2 = y.diff(t) + x - 1
sage: de1.laplace(t, s)
s*laplace(x(t), t, s) + 16*laplace(y(t), t, s) - x(0)
sage: de2.laplace(t, s)
s*laplace(y(t), t, s) - 1/s + laplace(x(t), t, s) - y(0)
```

```
>>> from sage.all import *
>>> var('t')
t
>>> t = var('t')
>>> x = function('x')(t)
>>> y = function('y')(t)
>>> de1 = x.diff(t) + Integer(16)*y
>>> de2 = y.diff(t) + x - Integer(1)
```

(continues on next page)

(continued from previous page)

```
>>> de1.laplace(t, s)
s*laplace(x(t), t, s) + 16*laplace(y(t), t, s) - x(0)
>>> de2.laplace(t, s)
s*laplace(y(t), t, s) - 1/s + laplace(x(t), t, s) - y(0)
```

Next we form the augmented matrix of the above system:

```
sage: A = matrix([[s, 16, 270], [1, s, 90+1/s]])
sage: E = A.echelon_form()
sage: xt = E[0,2].inverse_laplace(s,t)
sage: yt = E[1,2].inverse_laplace(s,t)
sage: xt
-91/2*e^(4*t) + 629/2*e^(-4*t) + 1
sage: yt
91/8*e^(4*t) + 629/8*e^(-4*t)
sage: p1 = plot(xt, 0, 1/2, rgbcolor=(1,0,0))                              #␣
↪needs sage.plot
sage: p2 = plot(yt, 0, 1/2, rgbcolor=(0,1,0))                              #␣
↪needs sage.plot
sage: import tempfile
sage: with tempfile.NamedTemporaryFile(suffix='.png') as f:                #␣
↪needs sage.plot
....:     (p1 + p2).save(f.name)
```

```
>>> from sage.all import *
>>> A = matrix([[s, Integer(16), Integer(270)], [Integer(1), s,␣
↪Integer(90)+Integer(1)/s]])
>>> E = A.echelon_form()
>>> xt = E[Integer(0),Integer(2)].inverse_laplace(s,t)
>>> yt = E[Integer(1),Integer(2)].inverse_laplace(s,t)
>>> xt
-91/2*e^(4*t) + 629/2*e^(-4*t) + 1
>>> yt
91/8*e^(4*t) + 629/8*e^(-4*t)
>>> p1 = plot(xt, Integer(0), Integer(1)/Integer(2), rgbcolor=(Integer(1),
↪Integer(0),Integer(0)))                                   # needs sage.plot
>>> p2 = plot(yt, Integer(0), Integer(1)/Integer(2), rgbcolor=(Integer(0),
↪Integer(1),Integer(0)))                                   # needs sage.plot
>>> import tempfile
>>> with tempfile.NamedTemporaryFile(suffix='.png') as f:                #␣
↪needs sage.plot
...     (p1 + p2).save(f.name)
```

Another example:

```
sage: var('a,s,t')
(a, s, t)
sage: f = exp (2*t + a) * sin(t) * t; f
t*e^(a + 2*t)*sin(t)
sage: L = laplace(f, t, s); L
2*(s - 2)*e^a/(s^2 - 4*s + 5)^2
sage: inverse_laplace(L, s, t)
```

(continues on next page)

(continued from previous page)

```
t*e^(a + 2*t)*sin(t)
```

```
>>> from sage.all import *
>>> var('a,s,t')
(a, s, t)
>>> f = exp (Integer(2)*t + a) * sin(t) * t; f
t*e^(a + 2*t)*sin(t)
>>> L = laplace(f, t, s); L
2*(s - 2)*e^a/(s^2 - 4*s + 5)^2
>>> inverse_laplace(L, s, t)
t*e^(a + 2*t)*sin(t)
```

The Laplace transform of the exponential function:

```
sage: laplace(exp(x), x, s)
1/(s - 1)
```

```
>>> from sage.all import *
>>> laplace(exp(x), x, s)
1/(s - 1)
```

Dirac's delta distribution is handled (the output of SymPy is related to a choice that has to be made when defining Laplace transforms of distributions):

```
sage: laplace(dirac_delta(t), t, s)
1
sage: F, a, cond = laplace(dirac_delta(t), t, s, algorithm='sympy')
sage: a, cond  # random - sympy <1.10 gives (-oo, True)
(0, True)
sage: F        # random - sympy <1.9 includes undefined heaviside(0) in answer
1
sage: laplace(dirac_delta(t), t, s, algorithm='giac')  # needs giac
1
```

```
>>> from sage.all import *
>>> laplace(dirac_delta(t), t, s)
1
>>> F, a, cond = laplace(dirac_delta(t), t, s, algorithm='sympy')
>>> a, cond  # random - sympy <1.10 gives (-oo, True)
(0, True)
>>> F        # random - sympy <1.9 includes undefined heaviside(0) in answer
1
>>> laplace(dirac_delta(t), t, s, algorithm='giac')  # needs giac
1
```

Heaviside step function can be handled with different interfaces. Try with Maxima:

```
sage: laplace(heaviside(t-1), t, s)
e^(-s)/s
```

```
>>> from sage.all import *
>>> laplace(heaviside(t-Integer(1)), t, s)
e^(-s)/s
```

Try with giac, if it is installed:

```
sage: # needs giac
sage: laplace(heaviside(t-1), t, s, algorithm='giac')
e^(-s)/s
```

```
>>> from sage.all import *
>>> # needs giac
>>> laplace(heaviside(t-Integer(1)), t, s, algorithm='giac')
e^(-s)/s
```

Try with SymPy:

```
sage: laplace(heaviside(t-1), t, s, algorithm='sympy')
(e^(-s)/s, 0, True)
```

```
>>> from sage.all import *
>>> laplace(heaviside(t-Integer(1)), t, s, algorithm='sympy')
(e^(-s)/s, 0, True)
```

`sage.calculus.calculus.`**`lim`**(*ex*, *dir=None*, *taylor=False*, *algorithm='maxima'*, ***argv*)

Return the limit as the variable v approaches a from the given direction.

```
expr.limit(x = a)
expr.limit(x = a, dir='+')
```

INPUT:

- `dir` – (default: `None`) may have the value `'plus'` (or `'+'` or `'right'` or `'above'`) for a limit from above, `'minus'` (or `'-'` or `'left'` or `'below'`) for a limit from below, or may be omitted (implying a two-sided limit is to be computed).
- `taylor` – (default: `False`) if `True`, use Taylor series, which allows more limits to be computed (but may also crash in some obscure cases due to bugs in Maxima).
- `**argv` – 1 named parameter

Note

The output may also use `und` (undefined), `ind` (indefinite but bounded), and `infinity` (complex infinity).

EXAMPLES:

```
sage: x = var('x')
sage: f = (1 + 1/x)^x
sage: f.limit(x=oo)
e
sage: f.limit(x=5)
7776/3125
```

```
>>> from sage.all import *
>>> x = var('x')
>>> f = (Integer(1) + Integer(1)/x)**x
>>> f.limit(x=oo)
e
>>> f.limit(x=Integer(5))
7776/3125
```

Domain to real, a regression in 5.46.0, see https://sf.net/p/maxima/bugs/4138

```
sage: maxima_calculus.eval("domain:real")
...
sage: f.limit(x=1.2).n()
2.06961575467...
sage: maxima_calculus.eval("domain:complex");
...
```

```
>>> from sage.all import *
>>> maxima_calculus.eval("domain:real")
...
>>> f.limit(x=RealNumber('1.2')).n()
2.06961575467...
>>> maxima_calculus.eval("domain:complex");
...
```

Otherwise, it works

```
sage: f.limit(x=I, taylor=True)
(-I + 1)^I
sage: f(x=1.2)
2.0696157546720...
sage: f(x=I)
(-I + 1)^I
sage: CDF(f(x=I))
2.0628722350809046 + 0.7450070621797239*I
sage: CDF(f.limit(x=I))
2.0628722350809046 + 0.7450070621797239*I
```

```
>>> from sage.all import *
>>> f.limit(x=I, taylor=True)
(-I + 1)^I
>>> f(x=RealNumber('1.2'))
2.0696157546720...
>>> f(x=I)
(-I + 1)^I
>>> CDF(f(x=I))
2.0628722350809046 + 0.7450070621797239*I
>>> CDF(f.limit(x=I))
2.0628722350809046 + 0.7450070621797239*I
```

Notice that Maxima may ask for more information:

```
sage: var('a')
a
sage: limit(x^a,x=0)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)', see
`assume?` for more details)
Is a positive, negative or zero?
```

```
>>> from sage.all import *
>>> var('a')
a
>>> limit(x**a,x=Integer(0))
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)', see
`assume?` for more details)
Is a positive, negative or zero?
```

With this example, Maxima is looking for a LOT of information:

```
sage: assume(a>0)
sage: limit(x^a,x=0)  # random - maxima 5.46.0 does not need extra assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(a>0)', see `assume?` for
 more details)
Is a an integer?
sage: assume(a,'integer')
sage: limit(x^a, x=0)  # random - maxima 5.46.0 does not need extra assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(a>0)', see `assume?` for
 more details)
Is a an even number?
sage: assume(a, 'even')
sage: limit(x^a, x=0)
0
sage: forget()
```

```
>>> from sage.all import *
>>> assume(a>Integer(0))
>>> limit(x**a,x=Integer(0))  # random - maxima 5.46.0 does not need extra␣
↪assumption
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(a>0)', see `assume?` for
 more details)
Is a an integer?
>>> assume(a,'integer')
>>> limit(x**a, x=Integer(0))  # random - maxima 5.46.0 does not need extra␣
↪assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(a>0)', see `assume?` for
 more details)
Is a an even number?
>>> assume(a, 'even')
>>> limit(x**a, x=Integer(0))
0
>>> forget()
```

More examples:

```
sage: limit(x*log(x), x=0, dir='+')
0
sage: lim((x+1)^(1/x), x=0)
e
sage: lim(e^x/x, x=oo)
+Infinity
sage: lim(e^x/x, x=-oo)
0
sage: lim(-e^x/x, x=oo)
-Infinity
sage: lim((cos(x))/(x^2), x=0)
+Infinity
sage: lim(sqrt(x^2+1) - x, x=oo)
0
sage: lim(x^2/(sec(x)-1), x=0)
2
sage: lim(cos(x)/(cos(x)-1), x=0)
-Infinity
sage: lim(x*sin(1/x), x=0)
0
sage: limit(e^(-1/x), x=0, dir='right')
0
sage: limit(e^(-1/x), x=0, dir='left')
+Infinity
```

```
>>> from sage.all import *
>>> limit(x*log(x), x=Integer(0), dir='+')
0
```

(continues on next page)

(continued from previous page)

```
>>> lim((x+Integer(1))**(Integer(1)/x), x=Integer(0))
e
>>> lim(e**x/x, x=oo)
+Infinity
>>> lim(e**x/x, x=-oo)
0
>>> lim(-e**x/x, x=oo)
-Infinity
>>> lim((cos(x))/(x**Integer(2)), x=Integer(0))
+Infinity
>>> lim(sqrt(x**Integer(2)+Integer(1)) - x, x=oo)
0
>>> lim(x**Integer(2)/(sec(x)-Integer(1)), x=Integer(0))
2
>>> lim(cos(x)/(cos(x)-Integer(1)), x=Integer(0))
-Infinity
>>> lim(x*sin(Integer(1)/x), x=Integer(0))
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='right')
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='left')
+Infinity
```

```
sage: f = log(log(x)) / log(x)
sage: forget(); assume(x < -2); lim(f, x=0, taylor=True)
0
sage: forget()
```

```
>>> from sage.all import *
>>> f = log(log(x)) / log(x)
>>> forget(); assume(x < -Integer(2)); lim(f, x=Integer(0), taylor=True)
0
>>> forget()
```

Here ind means "indefinite but bounded":

```
sage: lim(sin(1/x), x = 0)
ind
```

```
>>> from sage.all import *
>>> lim(sin(Integer(1)/x), x = Integer(0))
ind
```

We can use other packages than maxima, namely "sympy", "giac", "fricas".

With the standard package Giac:

```
sage: # needs sage.libs.giac
sage: (exp(-x)/(2+sin(x))).limit(x=oo, algorithm='giac')
0
sage: limit(e^(-1/x), x=0, dir='right', algorithm='giac')
0
```

(continues on next page)

(continued from previous page)

```
sage: limit(e^(-1/x), x=0, dir='left', algorithm='giac')
+Infinity
sage: (x / (x+2^x+cos(x))).limit(x=-infinity, algorithm='giac')
1
```

```
>>> from sage.all import *
>>> # needs sage.libs.giac
>>> (exp(-x)/(Integer(2)+sin(x))).limit(x=oo, algorithm='giac')
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='right', algorithm='giac')
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='left', algorithm='giac')
+Infinity
>>> (x / (x+Integer(2)**x+cos(x))).limit(x=-infinity, algorithm='giac')
1
```

With the optional package FriCAS:

```
sage: (x / (x+2^x+cos(x))).limit(x=-infinity, algorithm='fricas')       #␣
↪optional - fricas
1
sage: limit(e^(-1/x), x=0, dir='right', algorithm='fricas')             #␣
↪optional - fricas
0
sage: limit(e^(-1/x), x=0, dir='left', algorithm='fricas')              #␣
↪optional - fricas
+Infinity
```

```
>>> from sage.all import *
>>> (x / (x+Integer(2)**x+cos(x))).limit(x=-infinity, algorithm='fricas')       #␣
↪optional - fricas
1
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='right', algorithm='fricas')     ␣
↪       # optional - fricas
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='left', algorithm='fricas')      ␣
↪       # optional - fricas
+Infinity
```

One can also call Mathematica's online interface:

```
sage: limit(pi+log(x)/x,x=oo, algorithm='mathematica_free') # optional - internet
pi
```

```
>>> from sage.all import *
>>> limit(pi+log(x)/x,x=oo, algorithm='mathematica_free') # optional - internet
pi
```

`sage.calculus.calculus.`**`limit`**(*ex*, *dir=None*, *taylor=False*, *algorithm='maxima'*, ***argv*)

Return the limit as the variable v approaches a from the given direction.

```
expr.limit(x = a)
expr.limit(x = a, dir='+')
```

INPUT:

- `dir` – (default: `None`) may have the value `'plus'` (or `'+'` or `'right'` or `'above'`) for a limit from above, `'minus'` (or `'-'` or `'left'` or `'below'`) for a limit from below, or may be omitted (implying a two-sided limit is to be computed).
- `taylor` – (default: `False`) if `True`, use Taylor series, which allows more limits to be computed (but may also crash in some obscure cases due to bugs in Maxima).
- `**argv` – 1 named parameter

Note

The output may also use `und` (undefined), `ind` (indefinite but bounded), and `infinity` (complex infinity).

EXAMPLES:

```
sage: x = var('x')
sage: f = (1 + 1/x)^x
sage: f.limit(x=oo)
e
sage: f.limit(x=5)
7776/3125
```

```
>>> from sage.all import *
>>> x = var('x')
>>> f = (Integer(1) + Integer(1)/x)**x
>>> f.limit(x=oo)
e
>>> f.limit(x=Integer(5))
7776/3125
```

Domain to real, a regression in 5.46.0, see https://sf.net/p/maxima/bugs/4138

```
sage: maxima_calculus.eval("domain:real")
...
sage: f.limit(x=1.2).n()
2.06961575467...
sage: maxima_calculus.eval("domain:complex");
...
```

```
>>> from sage.all import *
>>> maxima_calculus.eval("domain:real")
...
>>> f.limit(x=RealNumber('1.2')).n()
2.06961575467...
>>> maxima_calculus.eval("domain:complex");
...
```

Otherwise, it works

```
sage: f.limit(x=I, taylor=True)
(-I + 1)^I
sage: f(x=1.2)
2.0696157546720...
sage: f(x=I)
(-I + 1)^I
sage: CDF(f(x=I))
2.0628722350809046 + 0.7450070621797239*I
sage: CDF(f.limit(x=I))
2.0628722350809046 + 0.7450070621797239*I
```

```
>>> from sage.all import *
>>> f.limit(x=I, taylor=True)
(-I + 1)^I
>>> f(x=RealNumber('1.2'))
2.0696157546720...
>>> f(x=I)
(-I + 1)^I
>>> CDF(f(x=I))
2.0628722350809046 + 0.7450070621797239*I
>>> CDF(f.limit(x=I))
2.0628722350809046 + 0.7450070621797239*I
```

Notice that Maxima may ask for more information:

```
sage: var('a')
a
sage: limit(x^a,x=0)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)', see
`assume?` for more details)
Is a positive, negative or zero?
```

```
>>> from sage.all import *
>>> var('a')
a
>>> limit(x**a,x=Integer(0))
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)', see
`assume?` for more details)
Is a positive, negative or zero?
```

With this example, Maxima is looking for a LOT of information:

```
sage: assume(a>0)
sage: limit(x^a,x=0)  # random - maxima 5.46.0 does not need extra assumption
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(a>0)', see `assume?` for
 more details)
Is a an integer?
sage: assume(a,'integer')
sage: limit(x^a, x=0)   # random - maxima 5.46.0 does not need extra assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(a>0)', see `assume?` for
 more details)
Is a an even number?
sage: assume(a, 'even')
sage: limit(x^a, x=0)
0
sage: forget()
```

```
>>> from sage.all import *
>>> assume(a>Integer(0))
>>> limit(x**a,x=Integer(0))   # random - maxima 5.46.0 does not need extra␣
↪assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(a>0)', see `assume?` for
 more details)
Is a an integer?
>>> assume(a,'integer')
>>> limit(x**a, x=Integer(0))   # random - maxima 5.46.0 does not need extra␣
↪assumption
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation *may* help
(example of legal syntax is 'assume(a>0)', see `assume?` for
 more details)
Is a an even number?
>>> assume(a, 'even')
>>> limit(x**a, x=Integer(0))
0
>>> forget()
```

More examples:

```
sage: limit(x*log(x), x=0, dir='+')
0
sage: lim((x+1)^(1/x), x=0)
e
```

(continues on next page)

(continued from previous page)

```
sage: lim(e^x/x, x=oo)
+Infinity
sage: lim(e^x/x, x=-oo)
0
sage: lim(-e^x/x, x=oo)
-Infinity
sage: lim((cos(x))/(x^2), x=0)
+Infinity
sage: lim(sqrt(x^2+1) - x, x=oo)
0
sage: lim(x^2/(sec(x)-1), x=0)
2
sage: lim(cos(x)/(cos(x)-1), x=0)
-Infinity
sage: lim(x*sin(1/x), x=0)
0
sage: limit(e^(-1/x), x=0, dir='right')
0
sage: limit(e^(-1/x), x=0, dir='left')
+Infinity
```

```
>>> from sage.all import *
>>> limit(x*log(x), x=Integer(0), dir='+')
0
>>> lim((x+Integer(1))**(Integer(1)/x), x=Integer(0))
e
>>> lim(e**x/x, x=oo)
+Infinity
>>> lim(e**x/x, x=-oo)
0
>>> lim(-e**x/x, x=oo)
-Infinity
>>> lim((cos(x))/(x**Integer(2)), x=Integer(0))
+Infinity
>>> lim(sqrt(x**Integer(2)+Integer(1)) - x, x=oo)
0
>>> lim(x**Integer(2)/(sec(x)-Integer(1)), x=Integer(0))
2
>>> lim(cos(x)/(cos(x)-Integer(1)), x=Integer(0))
-Infinity
>>> lim(x*sin(Integer(1)/x), x=Integer(0))
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='right')
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='left')
+Infinity
```

```
sage: f = log(log(x)) / log(x)
sage: forget(); assume(x < -2); lim(f, x=0, taylor=True)
0
sage: forget()
```

```
>>> from sage.all import *
>>> f = log(log(x)) / log(x)
>>> forget(); assume(x < -Integer(2)); lim(f, x=Integer(0), taylor=True)
0
>>> forget()
```

Here ind means "indefinite but bounded":

```
sage: lim(sin(1/x), x = 0)
ind
```

```
>>> from sage.all import *
>>> lim(sin(Integer(1)/x), x = Integer(0))
ind
```

We can use other packages than maxima, namely "sympy", "giac", "fricas".

With the standard package Giac:

```
sage: # needs sage.libs.giac
sage: (exp(-x)/(2+sin(x))).limit(x=oo, algorithm='giac')
0
sage: limit(e^(-1/x), x=0, dir='right', algorithm='giac')
0
sage: limit(e^(-1/x), x=0, dir='left', algorithm='giac')
+Infinity
sage: (x / (x+2^x+cos(x))).limit(x=-infinity, algorithm='giac')
1
```

```
>>> from sage.all import *
>>> # needs sage.libs.giac
>>> (exp(-x)/(Integer(2)+sin(x))).limit(x=oo, algorithm='giac')
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='right', algorithm='giac')
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='left', algorithm='giac')
+Infinity
>>> (x / (x+Integer(2)**x+cos(x))).limit(x=-infinity, algorithm='giac')
1
```

With the optional package FriCAS:

```
sage: (x / (x+2^x+cos(x))).limit(x=-infinity, algorithm='fricas')        #␣
↪optional - fricas
1
sage: limit(e^(-1/x), x=0, dir='right', algorithm='fricas')              #␣
↪optional - fricas
0
sage: limit(e^(-1/x), x=0, dir='left', algorithm='fricas')               #␣
↪optional - fricas
+Infinity
```

```
>>> from sage.all import *
>>> (x / (x+Integer(2)**x+cos(x))).limit(x=-infinity, algorithm='fricas')        #␣
↪optional - fricas
1
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='right', algorithm='fricas')     ␣
↪       # optional - fricas
0
>>> limit(e**(-Integer(1)/x), x=Integer(0), dir='left', algorithm='fricas')      ␣
↪       # optional - fricas
+Infinity
```

One can also call Mathematica's online interface:

```
sage: limit(pi+log(x)/x,x=oo, algorithm='mathematica_free') # optional - internet
pi
```

```
>>> from sage.all import *
>>> limit(pi+log(x)/x,x=oo, algorithm='mathematica_free') # optional - internet
pi
```

sage.calculus.calculus.**mapped_opts**(*v*)

Used internally when creating a string of options to pass to Maxima.

INPUT:

- v – an object

OUTPUT: string

The main use of this is to turn Python bools into lower case strings.

EXAMPLES:

```
sage: sage.calculus.calculus.mapped_opts(True)
'true'
sage: sage.calculus.calculus.mapped_opts(False)
'false'
sage: sage.calculus.calculus.mapped_opts('bar')
'bar'
```

```
>>> from sage.all import *
>>> sage.calculus.calculus.mapped_opts(True)
'true'
>>> sage.calculus.calculus.mapped_opts(False)
'false'
>>> sage.calculus.calculus.mapped_opts('bar')
'bar'
```

sage.calculus.calculus.**maxima_options**(***kwds*)

Used internally to create a string of options to pass to Maxima.

EXAMPLES:

```
sage: sage.calculus.calculus.maxima_options(an_option=True, another=False, foo=
↪'bar')
'an_option=true,another=false,foo=bar'
```

```
>>> from sage.all import *
>>> sage.calculus.calculus.maxima_options(an_option=True, another=False, foo='bar
↪')
'an_option=true,another=false,foo=bar'
```

sage.calculus.calculus.**minpoly**(*ex*, *var='x'*, *algorithm=None*, *bits=None*, *degree=None*, *epsilon=0*)

Return the minimal polynomial of `self`, if possible.

INPUT:

- `var` – polynomial variable name (default: `'x'`)
- `algorithm` – `'algebraic'` or `'numerical'` (default both, but with numerical first)
- `bits` – the number of bits to use in numerical approx
- `degree` – the expected algebraic degree
- `epsilon` – return without error as long as f(self) epsilon, in the case that the result cannot be proven

  All of the above parameters are optional, with epsilon=0, `bits` and `degree` tested up to 1000 and 24 by default respectively. The numerical algorithm will be faster if bits and/or degree are given explicitly. The algebraic algorithm ignores the last three parameters.

OUTPUT: the minimal polynomial of `self`. If the numerical algorithm is used, then it is proved symbolically when `epsilon=0` (default).

If the minimal polynomial could not be found, two distinct kinds of errors are raised. If no reasonable candidate was found with the given `bits`/`degree` parameters, a `ValueError` will be raised. If a reasonable candidate was found but (perhaps due to limits in the underlying symbolic package) was unable to be proved correct, a `NotImplementedError` will be raised.

ALGORITHM: Two distinct algorithms are used, depending on the algorithm parameter. By default, the numerical algorithm is attempted first, then the algebraic one.

Algebraic: Attempt to evaluate this expression in `QQbar`, using cyclotomic fields to resolve exponential and trig functions at rational multiples of π, field extensions to handle roots and rational exponents, and computing compositums to represent the full expression as an element of a number field where the minimal polynomial can be computed exactly. The `bits`, `degree`, and `epsilon` parameters are ignored.

Numerical: Computes a numerical approximation of `self` and use PARI's pari:algdep to get a candidate minpoly f. If $f(\texttt{self})$, evaluated to a higher precision, is close enough to 0 then evaluate $f(\texttt{self})$ symbolically, attempting to prove vanishing. If this fails, and `epsilon` is nonzero, return f if and only if $f(\texttt{self}) < \texttt{epsilon}$. Otherwise raise a `ValueError` (if no suitable candidate was found) or a `NotImplementedError` (if a likely candidate was found but could not be proved correct).

EXAMPLES: First some simple examples:

```
sage: sqrt(2).minpoly()
x^2 - 2
sage: minpoly(2^(1/3))
x^3 - 2
sage: minpoly(sqrt(2) + sqrt(-1))
x^4 - 2*x^2 + 9
sage: minpoly(sqrt(2)-3^(1/3))
x^6 - 6*x^4 + 6*x^3 + 12*x^2 + 36*x + 1
```

```
>>> from sage.all import *
>>> sqrt(Integer(2)).minpoly()
```

(continues on next page)

(continued from previous page)

```
x^2 - 2
>>> minpoly(Integer(2)**(Integer(1)/Integer(3)))
x^3 - 2
>>> minpoly(sqrt(Integer(2)) + sqrt(-Integer(1)))
x^4 - 2*x^2 + 9
>>> minpoly(sqrt(Integer(2))-Integer(3)**(Integer(1)/Integer(3)))
x^6 - 6*x^4 + 6*x^3 + 12*x^2 + 36*x + 1
```

Works with trig and exponential functions too.

```
sage: sin(pi/3).minpoly()
x^2 - 3/4
sage: sin(pi/7).minpoly()
x^6 - 7/4*x^4 + 7/8*x^2 - 7/64
sage: minpoly(exp(I*pi/17))
x^16 - x^15 + x^14 - x^13 + x^12 - x^11 + x^10 - x^9 + x^8
 - x^7 + x^6 - x^5 + x^4 - x^3 + x^2 - x + 1
```

```
>>> from sage.all import *
>>> sin(pi/Integer(3)).minpoly()
x^2 - 3/4
>>> sin(pi/Integer(7)).minpoly()
x^6 - 7/4*x^4 + 7/8*x^2 - 7/64
>>> minpoly(exp(I*pi/Integer(17)))
x^16 - x^15 + x^14 - x^13 + x^12 - x^11 + x^10 - x^9 + x^8
 - x^7 + x^6 - x^5 + x^4 - x^3 + x^2 - x + 1
```

Here we verify it gives the same result as the abstract number field.

```
sage: (sqrt(2) + sqrt(3) + sqrt(6)).minpoly()
x^4 - 22*x^2 - 48*x - 23
sage: K.<a,b> = NumberField([x^2-2, x^2-3])
sage: (a+b+a*b).absolute_minpoly()
x^4 - 22*x^2 - 48*x - 23
```

```
>>> from sage.all import *
>>> (sqrt(Integer(2)) + sqrt(Integer(3)) + sqrt(Integer(6))).minpoly()
x^4 - 22*x^2 - 48*x - 23
>>> K = NumberField([x**Integer(2)-Integer(2), x**Integer(2)-Integer(3)], names=(
↪'a', 'b',)); (a, b,) = K._first_ngens(2)
>>> (a+b+a*b).absolute_minpoly()
x^4 - 22*x^2 - 48*x - 23
```

The `minpoly()` function is used implicitly when creating number fields:

```
sage: x = var('x')
sage: eqn =  x^3 + sqrt(2)*x + 5 == 0
sage: a = solve(eqn, x)[0].rhs()
sage: QQ[a]
Number Field in a with defining polynomial x^6 + 10*x^3 - 2*x^2 + 25
 with a = 0.7185272465828846? - 1.721353471724806?*I
```

```
>>> from sage.all import *
>>> x = var('x')
>>> eqn =  x**Integer(3) + sqrt(Integer(2))*x + Integer(5) == Integer(0)
>>> a = solve(eqn, x)[Integer(0)].rhs()
>>> QQ[a]
Number Field in a with defining polynomial x^6 + 10*x^3 - 2*x^2 + 25
 with a = 0.7185272465828846? - 1.721353471724806?*I
```

Here we solve a cubic and then recover it from its complicated radical expansion.

```
sage: f = x^3 - x + 1
sage: a = f.solve(x)[0].rhs(); a
-1/2*(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)*(I*sqrt(3) + 1)
 - 1/6*(-I*sqrt(3) + 1)/(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)
sage: a.minpoly()
x^3 - x + 1
```

```
>>> from sage.all import *
>>> f = x**Integer(3) - x + Integer(1)
>>> a = f.solve(x)[Integer(0)].rhs(); a
-1/2*(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)*(I*sqrt(3) + 1)
 - 1/6*(-I*sqrt(3) + 1)/(1/18*sqrt(23)*sqrt(3) - 1/2)^(1/3)
>>> a.minpoly()
x^3 - x + 1
```

Note that simplification may be necessary to see that the minimal polynomial is correct.

```
sage: a = sqrt(2)+sqrt(3)+sqrt(5)
sage: f = a.minpoly(); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
sage: f(a)
(sqrt(5) + sqrt(3) + sqrt(2))^8 - 40*(sqrt(5) + sqrt(3) + sqrt(2))^6
 + 352*(sqrt(5) + sqrt(3) + sqrt(2))^4 - 960*(sqrt(5) + sqrt(3) + sqrt(2))^2
 + 576
sage: f(a).expand()
0
```

```
>>> from sage.all import *
>>> a = sqrt(Integer(2))+sqrt(Integer(3))+sqrt(Integer(5))
>>> f = a.minpoly(); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
>>> f(a)
(sqrt(5) + sqrt(3) + sqrt(2))^8 - 40*(sqrt(5) + sqrt(3) + sqrt(2))^6
 + 352*(sqrt(5) + sqrt(3) + sqrt(2))^4 - 960*(sqrt(5) + sqrt(3) + sqrt(2))^2
 + 576
>>> f(a).expand()
0
```

```
sage: a = sin(pi/7)
sage: f = a.minpoly(algorithm='numerical'); f
x^6 - 7/4*x^4 + 7/8*x^2 - 7/64
sage: f(a).horner(a).numerical_approx(100)
0.00000000000000000000000000000
```

```
>>> from sage.all import *
>>> a = sin(pi/Integer(7))
>>> f = a.minpoly(algorithm='numerical'); f
x^6 - 7/4*x^4 + 7/8*x^2 - 7/64
>>> f(a).horner(a).numerical_approx(Integer(100))
0.00000000000000000000000000000
```

The degree must be high enough (default tops out at 24).

```
sage: a = sqrt(3) + sqrt(2)
sage: a.minpoly(algorithm='numerical', bits=100, degree=3)
Traceback (most recent call last):
...
ValueError: Could not find minimal polynomial (100 bits, degree 3).
sage: a.minpoly(algorithm='numerical', bits=100, degree=10)
x^4 - 10*x^2 + 1
```

```
>>> from sage.all import *
>>> a = sqrt(Integer(3)) + sqrt(Integer(2))
>>> a.minpoly(algorithm='numerical', bits=Integer(100), degree=Integer(3))
Traceback (most recent call last):
...
ValueError: Could not find minimal polynomial (100 bits, degree 3).
>>> a.minpoly(algorithm='numerical', bits=Integer(100), degree=Integer(10))
x^4 - 10*x^2 + 1
```

```
sage: cos(pi/33).minpoly(algorithm='algebraic')
x^10 + 1/2*x^9 - 5/2*x^8 - 5/4*x^7 + 17/8*x^6 + 17/16*x^5
 - 43/64*x^4 - 43/128*x^3 + 3/64*x^2 + 3/128*x + 1/1024
sage: cos(pi/33).minpoly(algorithm='numerical')
x^10 + 1/2*x^9 - 5/2*x^8 - 5/4*x^7 + 17/8*x^6 + 17/16*x^5
 - 43/64*x^4 - 43/128*x^3 + 3/64*x^2 + 3/128*x + 1/1024
```

```
>>> from sage.all import *
>>> cos(pi/Integer(33)).minpoly(algorithm='algebraic')
x^10 + 1/2*x^9 - 5/2*x^8 - 5/4*x^7 + 17/8*x^6 + 17/16*x^5
 - 43/64*x^4 - 43/128*x^3 + 3/64*x^2 + 3/128*x + 1/1024
>>> cos(pi/Integer(33)).minpoly(algorithm='numerical')
x^10 + 1/2*x^9 - 5/2*x^8 - 5/4*x^7 + 17/8*x^6 + 17/16*x^5
 - 43/64*x^4 - 43/128*x^3 + 3/64*x^2 + 3/128*x + 1/1024
```

Sometimes it fails, as it must given that some numbers aren't algebraic:

```
sage: sin(1).minpoly(algorithm='numerical')
Traceback (most recent call last):
...
ValueError: Could not find minimal polynomial (1000 bits, degree 24).
```

```
>>> from sage.all import *
>>> sin(Integer(1)).minpoly(algorithm='numerical')
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: Could not find minimal polynomial (1000 bits, degree 24).
```

Note: Of course, failure to produce a minimal polynomial does not necessarily indicate that this number is transcendental.

sage.calculus.calculus.**mma_free_limit**(*expression*, *v*, *a*, *dir=None*)

Limit using Mathematica's online interface.

INPUT:

- `expression` – symbolic expression
- `v` – variable
- `a` – value where the variable goes to
- `dir` – `'+'`, `'-'` or `None` (default: `None`)

EXAMPLES:

```
sage: from sage.calculus.calculus import mma_free_limit
sage: mma_free_limit(sin(x)/x, x, a=0) # optional - internet
1
```

```
>>> from sage.all import *
>>> from sage.calculus.calculus import mma_free_limit
>>> mma_free_limit(sin(x)/x, x, a=Integer(0)) # optional - internet
1
```

Another simple limit:

```
sage: mma_free_limit(e^(-x), x, a=oo) # optional - internet
0
```

```
>>> from sage.all import *
>>> mma_free_limit(e**(-x), x, a=oo) # optional - internet
0
```

sage.calculus.calculus.**nintegral**(*ex*, *x*, *a*, *b*, *desired_relative_error='1e-8'*, *maximum_num_subintervals=200*)

Return a floating point machine precision numerical approximation to the integral of `self` from a to b, computed using floating point arithmetic via maxima.

INPUT:

- `x` – variable to integrate with respect to
- `a` – lower endpoint of integration
- `b` – upper endpoint of integration
- `desired_relative_error` – (default: `1e-8`) the desired relative error
- `maximum_num_subintervals` – (default: 200) maximal number of subintervals

OUTPUT: float; approximation to the integral

- float: estimated absolute error of the approximation
- the number of integrand evaluations
- an error code:
  - `0` – no problems were encountered
  - `1` – too many subintervals were done
  - `2` – excessive roundoff error
  - `3` – extremely bad integrand behavior
  - `4` – failed to converge
  - `5` – integral is probably divergent or slowly convergent
  - `6` – the input is invalid; this includes the case of `desired_relative_error` being too small to be achieved

ALIAS: `nintegrate()` is the same as `nintegral()`

REMARK: There is also a function `numerical_integral()` that implements numerical integration using the GSL C library. It is potentially much faster and applies to arbitrary user defined functions.

Also, there are limits to the precision to which Maxima can compute the integral due to limitations in quadpack. In the following example, remark that the last value of the returned tuple is `6`, indicating that the input was invalid, in this case because of a too high desired precision.

```
sage: f = x
sage: f.nintegral(x, 0, 1, 1e-14)
(0.0, 0.0, 0, 6)
```

```
>>> from sage.all import *
>>> f = x
>>> f.nintegral(x, Integer(0), Integer(1), RealNumber('1e-14'))
(0.0, 0.0, 0, 6)
```

EXAMPLES:

```
sage: f(x) = exp(-sqrt(x))
sage: f.nintegral(x, 0, 1)
(0.5284822353142306, 4.163...e-11, 231, 0)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(exp(-sqrt(x))).function(x)
>>> f.nintegral(x, Integer(0), Integer(1))
(0.5284822353142306, 4.163...e-11, 231, 0)
```

We can also use the `numerical_integral()` function, which calls the GSL C library.

```
sage: numerical_integral(f, 0, 1)
(0.528482232253147, 6.83928460...e-07)
```

```
>>> from sage.all import *
>>> numerical_integral(f, Integer(0), Integer(1))
(0.528482232253147, 6.83928460...e-07)
```

Note that in exotic cases where floating point evaluation of the expression leads to the wrong value, then the output can be completely wrong:

```
sage: f = exp(pi*sqrt(163)) - 262537412640768744
```

```
>>> from sage.all import *
>>> f = exp(pi*sqrt(Integer(163))) - Integer(262537412640768744)
```

Despite appearance, f is really very close to 0, but one gets a nonzero value since the definition of `float(f)` is that it makes all constants inside the expression floats, then evaluates each function and each arithmetic operation using float arithmetic:

```
sage: float(f)
-480.0
```

```
>>> from sage.all import *
>>> float(f)
-480.0
```

Computing to higher precision we see the truth:

```
sage: f.n(200)
-7.4992740280181431112064614366622348652078895136533593355718e-13
sage: f.n(300)
-7.4992740280181431112064614366622348652078895136533593355718e-13
sage: f.n(300)
-7.49927402801814311120646143662663009137292462589621789352095066181709095575681963967103004e-13
```

```
>>> from sage.all import *
>>> f.n(Integer(200))
-7.4992740280181431112064614366622348652078895136533593355718e-13
>>> f.n(Integer(300))
-7.49927402801814311120646143662663009137292462589621789352095066181709095575681963967103004e-13
```

Now numerically integrating, we see why the answer is wrong:

```
sage: f.nintegrate(x,0,1)
(-480.000000000000..., 5.32907051820075...e-12, 21, 0)
```

```
>>> from sage.all import *
>>> f.nintegrate(x,Integer(0),Integer(1))
(-480.000000000000..., 5.32907051820075...e-12, 21, 0)
```

It is just because every floating point evaluation of f returns -480.0 in floating point.

Important note: using PARI/GP one can compute numerical integrals to high precision:

```
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829176211363206621657 E-127'
sage: old_prec = gp.set_real_precision(50)
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304957417746108003917 E-127'
sage: gp.set_real_precision(old_prec)
57
```

```
>>> from sage.all import *
>>> gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.565728500561051482917621136320662165 7 E-127'
>>> old_prec = gp.set_real_precision(Integer(50))
>>> gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304957417746108003917 E-127'
>>> gp.set_real_precision(old_prec)
57
```

Note that the input function above is a string in PARI syntax.

`sage.calculus.calculus.nintegrate`(*ex*, *x*, *a*, *b*, *desired_relative_error='1e-8'*, *maximum_num_subintervals=200*)

Return a floating point machine precision numerical approximation to the integral of `self` from a to b, computed using floating point arithmetic via maxima.

INPUT:

- `x` – variable to integrate with respect to
- `a` – lower endpoint of integration
- `b` – upper endpoint of integration
- `desired_relative_error` – (default: `1e-8`) the desired relative error
- `maximum_num_subintervals` – (default: 200) maximal number of subintervals

OUTPUT: float; approximation to the integral

- float: estimated absolute error of the approximation
- the number of integrand evaluations
- an error code:
  - `0` – no problems were encountered
  - `1` – too many subintervals were done
  - `2` – excessive roundoff error
  - `3` – extremely bad integrand behavior
  - `4` – failed to converge
  - `5` – integral is probably divergent or slowly convergent
  - `6` – the input is invalid; this includes the case of `desired_relative_error` being too small to be achieved

ALIAS: `nintegrate()` is the same as `nintegral()`

REMARK: There is also a function `numerical_integral()` that implements numerical integration using the GSL C library. It is potentially much faster and applies to arbitrary user defined functions.

Also, there are limits to the precision to which Maxima can compute the integral due to limitations in quadpack. In the following example, remark that the last value of the returned tuple is `6`, indicating that the input was invalid, in this case because of a too high desired precision.

```
sage: f = x
sage: f.nintegral(x, 0, 1, 1e-14)
(0.0, 0.0, 0, 6)
```

```
>>> from sage.all import *
>>> f = x
>>> f.nintegral(x, Integer(0), Integer(1), RealNumber('1e-14'))
(0.0, 0.0, 0, 6)
```

EXAMPLES:

```
sage: f(x) = exp(-sqrt(x))
sage: f.nintegral(x, 0, 1)
(0.5284822353142306, 4.163...e-11, 231, 0)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(exp(-sqrt(x))).function(x)
>>> f.nintegral(x, Integer(0), Integer(1))
(0.5284822353142306, 4.163...e-11, 231, 0)
```

We can also use the `numerical_integral()` function, which calls the GSL C library.

```
sage: numerical_integral(f, 0, 1)
(0.528482232253147, 6.83928460...e-07)
```

```
>>> from sage.all import *
>>> numerical_integral(f, Integer(0), Integer(1))
(0.528482232253147, 6.83928460...e-07)
```

Note that in exotic cases where floating point evaluation of the expression leads to the wrong value, then the output can be completely wrong:

```
sage: f = exp(pi*sqrt(163)) - 262537412640768744
```

```
>>> from sage.all import *
>>> f = exp(pi*sqrt(Integer(163))) - Integer(262537412640768744)
```

Despite appearance, f is really very close to 0, but one gets a nonzero value since the definition of `float(f)` is that it makes all constants inside the expression floats, then evaluates each function and each arithmetic operation using float arithmetic:

```
sage: float(f)
-480.0
```

```
>>> from sage.all import *
>>> float(f)
-480.0
```

Computing to higher precision we see the truth:

```
sage: f.n(200)
-7.4992740280181431112064614366622348652078895136533593355718e-13
sage: f.n(300)
-7.49927402801814311120646143662663009137292462589621789352095066181709095575681963967103004e-13
```

```
>>> from sage.all import *
>>> f.n(Integer(200))
-7.499274028018143111206461436662234865207889513653359335571 8e-13
>>> f.n(Integer(300))
-7.499274028018143111206461436626630091372924625896217893520950661817090955756819 6
↪3967103004e-13
```

Now numerically integrating, we see why the answer is wrong:

```
sage: f.nintegrate(x,0,1)
(-480.000000000000..., 5.32907051820075...e-12, 21, 0)
```

```
>>> from sage.all import *
>>> f.nintegrate(x,Integer(0),Integer(1))
(-480.000000000000..., 5.32907051820075...e-12, 21, 0)
```

It is just because every floating point evaluation of f returns -480.0 in floating point.

Important note: using PARI/GP one can compute numerical integrals to high precision:

```
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.565728500561051482917621136320662 1657 E-127'
sage: old_prec = gp.set_real_precision(50)
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304957417746108003917 E-127'
sage: gp.set_real_precision(old_prec)
57
```

```
>>> from sage.all import *
>>> gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.565728500561051482917621136320662 1657 E-127'
>>> old_prec = gp.set_real_precision(Integer(50))
>>> gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304957417746108003917 E-127'
>>> gp.set_real_precision(old_prec)
57
```

Note that the input function above is a string in PARI syntax.

`sage.calculus.calculus.`**`symbolic_expression_from_maxima_string`**(*x*, *equals_sub=False*, *maxima=Maxima_lib*)

Given a string representation of a Maxima expression, parse it and return the corresponding Sage symbolic expression.

INPUT:

- `x` – string
- `equals_sub` – boolean (default: `False`); if `True`, replace '=' by '==' in self
- `maxima` – (default: the calculus package's copy of Maxima) the Maxima interpreter to use

EXAMPLES:

```
sage: from sage.calculus.calculus import symbolic_expression_from_maxima_string␣
↪as sefms
```

(continues on next page)

(continued from previous page)

```
sage: sefms('x^%e + %e^%pi + %i + sin(0)')
x^e + e^pi + I
sage: f = function('f')(x)
sage: sefms('?%at(f(x),x=2)#1')
f(2) != 1
sage: a = sage.calculus.calculus.maxima("x#0"); a
x # 0
sage: a.sage()
x != 0
```

```
>>> from sage.all import *
>>> from sage.calculus.calculus import symbolic_expression_from_maxima_string as␣
↪sefms
>>> sefms('x^%e + %e^%pi + %i + sin(0)')
x^e + e^pi + I
>>> f = function('f')(x)
>>> sefms('?%at(f(x),x=2)#1')
f(2) != 1
>>> a = sage.calculus.calculus.maxima("x#0"); a
x # 0
>>> a.sage()
x != 0
```

sage.calculus.calculus.**symbolic_expression_from_string**(*s*, *syms*, *accept_sequence=None*, *parser=False*)

Given a string, (attempt to) parse it and return the corresponding Sage symbolic expression. Normally used to return Maxima output to the user.

INPUT:

- `s` – string
- `syms` – (default: `{}`) dictionary of strings to be regarded as symbols or functions; keys are pairs (string, number of arguments)
- `accept_sequence` – boolean (default: `False`); controls whether to allow a (possibly nested) set of lists and tuples as input
- `parser` – (default: `SR_parser`) parser for internal use

EXAMPLES:

```
sage: from sage.calculus.calculus import symbolic_expression_from_string
sage: y = var('y')
sage: symbolic_expression_from_string('[sin(0)*x^2,3*spam+e^pi]',
....:                                 syms={('spam',0): y}, accept_sequence=True)
[0, 3*y + e^pi]
```

```
>>> from sage.all import *
>>> from sage.calculus.calculus import symbolic_expression_from_string
>>> y = var('y')
>>> symbolic_expression_from_string('[sin(0)*x^2,3*spam+e^pi]',
...                                 syms={('spam',Integer(0)): y}, accept_
↪sequence=True)
[0, 3*y + e^pi]
```

sage.calculus.calculus.**symbolic_product**(*expression, v, a, b, algorithm='maxima', hold=False*)

Return the symbolic product $\prod_{v=a}^{b} expression$ with respect to the variable v with endpoints a and b.

INPUT:

- `expression` – a symbolic expression
- `v` – a variable or variable name
- `a` – lower endpoint of the product
- `b` – upper endpoint of the prduct
- `algorithm` – (default: `'maxima'`) one of
  - `'maxima'` – use Maxima (the default)
  - `'giac'` – use Giac (optional)
  - `'sympy'` – use SymPy
  - `'mathematica'` – (optional) use Mathematica
- `hold` – boolean (default: `False`); if `True`, don't evaluate

EXAMPLES:

```
sage: i, k, n = var('i,k,n')
sage: from sage.calculus.calculus import symbolic_product
sage: symbolic_product(k, k, 1, n)
factorial(n)
sage: symbolic_product(x + i*(i+1)/2, i, 1, 4)
x^4 + 20*x^3 + 127*x^2 + 288*x + 180
sage: symbolic_product(i^2, i, 1, 7)
25401600
sage: f = function('f')
sage: symbolic_product(f(i), i, 1, 7)
f(7)*f(6)*f(5)*f(4)*f(3)*f(2)*f(1)
sage: symbolic_product(f(i), i, 1, n)
product(f(i), i, 1, n)
sage: assume(k>0)
sage: symbolic_product(integrate (x^k, x, 0, 1), k, 1, n)
1/factorial(n + 1)
sage: symbolic_product(f(i), i, 1, n).log().log_expand()
sum(log(f(i)), i, 1, n)
```

```
>>> from sage.all import *
>>> i, k, n = var('i,k,n')
>>> from sage.calculus.calculus import symbolic_product
>>> symbolic_product(k, k, Integer(1), n)
factorial(n)
>>> symbolic_product(x + i*(i+Integer(1))/Integer(2), i, Integer(1), Integer(4))
x^4 + 20*x^3 + 127*x^2 + 288*x + 180
>>> symbolic_product(i**Integer(2), i, Integer(1), Integer(7))
25401600
>>> f = function('f')
>>> symbolic_product(f(i), i, Integer(1), Integer(7))
f(7)*f(6)*f(5)*f(4)*f(3)*f(2)*f(1)
>>> symbolic_product(f(i), i, Integer(1), n)
```

(continues on next page)

(continued from previous page)

```
product(f(i), i, 1, n)
>>> assume(k>Integer(0))
>>> symbolic_product(integrate (x**k, x, Integer(0), Integer(1)), k, Integer(1),␣
↪n)
1/factorial(n + 1)
>>> symbolic_product(f(i), i, Integer(1), n).log().log_expand()
sum(log(f(i)), i, 1, n)
```

sage.calculus.calculus.**symbolic_sum**(*expression*, *v*, *a*, *b*, *algorithm='maxima'*, *hold=False*)

Return the symbolic sum $\sum_{v=a}^{b} expression$ with respect to the variable v with endpoints a and b.

INPUT:

- `expression` – a symbolic expression
- `v` – a variable or variable name
- `a` – lower endpoint of the sum
- `b` – upper endpoint of the sum
- `algorithm` – (default: `'maxima'`) one of
  - `'maxima'` – use Maxima (the default)
  - `'maple'` – (optional) use Maple
  - `'mathematica'` – (optional) use Mathematica
  - `'giac'` – (optional) use Giac
  - `'sympy'` – use SymPy
- `hold` – boolean (default: `False`); if `True`, don't evaluate

EXAMPLES:

```
sage: k, n = var('k,n')
sage: from sage.calculus.calculus import symbolic_sum
sage: symbolic_sum(k, k, 1, n).factor()
1/2*(n + 1)*n
```

```
>>> from sage.all import *
>>> k, n = var('k,n')
>>> from sage.calculus.calculus import symbolic_sum
>>> symbolic_sum(k, k, Integer(1), n).factor()
1/2*(n + 1)*n
```

```
sage: symbolic_sum(1/k^4, k, 1, oo)
1/90*pi^4
```

```
>>> from sage.all import *
>>> symbolic_sum(Integer(1)/k**Integer(4), k, Integer(1), oo)
1/90*pi^4
```

```
sage: symbolic_sum(1/k^5, k, 1, oo)
zeta(5)
```

```
>>> from sage.all import *
>>> symbolic_sum(Integer(1)/k**Integer(5), k, Integer(1), oo)
zeta(5)
```

A well known binomial identity:

```
sage: symbolic_sum(binomial(n,k), k, 0, n)
2^n
```

```
>>> from sage.all import *
>>> symbolic_sum(binomial(n,k), k, Integer(0), n)
2^n
```

And some truncations thereof:

```
sage: assume(n>1)
sage: symbolic_sum(binomial(n,k), k, 1, n)
2^n - 1
sage: symbolic_sum(binomial(n,k), k, 2, n)
2^n - n - 1
sage: symbolic_sum(binomial(n,k), k, 0, n-1)
2^n - 1
sage: symbolic_sum(binomial(n,k), k, 1, n-1)
2^n - 2
```

```
>>> from sage.all import *
>>> assume(n>Integer(1))
>>> symbolic_sum(binomial(n,k), k, Integer(1), n)
2^n - 1
>>> symbolic_sum(binomial(n,k), k, Integer(2), n)
2^n - n - 1
>>> symbolic_sum(binomial(n,k), k, Integer(0), n-Integer(1))
2^n - 1
>>> symbolic_sum(binomial(n,k), k, Integer(1), n-Integer(1))
2^n - 2
```

The binomial theorem:

```
sage: x, y = var('x, y')
sage: symbolic_sum(binomial(n,k) * x^k * y^(n-k), k, 0, n)
(x + y)^n
```

```
>>> from sage.all import *
>>> x, y = var('x, y')
>>> symbolic_sum(binomial(n,k) * x**k * y**(n-k), k, Integer(0), n)
(x + y)^n
```

```
sage: symbolic_sum(k * binomial(n, k), k, 1, n)
2^(n - 1)*n
```

```
>>> from sage.all import *
>>> symbolic_sum(k * binomial(n, k), k, Integer(1), n)
```

(continues on next page)

(continued from previous page)

```
2^(n - 1)*n
```

```
sage: symbolic_sum((-1)^k*binomial(n,k), k, 0, n)
0
```

```
>>> from sage.all import *
>>> symbolic_sum((-Integer(1))**k*binomial(n,k), k, Integer(0), n)
0
```

```
sage: symbolic_sum(2^(-k)/(k*(k+1)), k, 1, oo)
-log(2) + 1
```

```
>>> from sage.all import *
>>> symbolic_sum(Integer(2)**(-k)/(k*(k+Integer(1))), k, Integer(1), oo)
-log(2) + 1
```

Summing a hypergeometric term:

```
sage: symbolic_sum(binomial(n, k) * factorial(k) / factorial(n+1+k), k, 0, n)
1/2*sqrt(pi)/factorial(n + 1/2)
```

```
>>> from sage.all import *
>>> symbolic_sum(binomial(n, k) * factorial(k) / factorial(n+Integer(1)+k), k,␣
↪Integer(0), n)
1/2*sqrt(pi)/factorial(n + 1/2)
```

We check a well known identity:

```
sage: bool(symbolic_sum(k^3, k, 1, n) == symbolic_sum(k, k, 1, n)^2)
True
```

```
>>> from sage.all import *
>>> bool(symbolic_sum(k**Integer(3), k, Integer(1), n) == symbolic_sum(k, k,␣
↪Integer(1), n)**Integer(2))
True
```

A geometric sum:

```
sage: a, q = var('a, q')
sage: symbolic_sum(a*q^k, k, 0, n)
(a*q^(n + 1) - a)/(q - 1)
```

```
>>> from sage.all import *
>>> a, q = var('a, q')
>>> symbolic_sum(a*q**k, k, Integer(0), n)
(a*q^(n + 1) - a)/(q - 1)
```

For the geometric series, we will have to assume the right values for the sum to converge:

```
sage: assume(abs(q) < 1)
sage: symbolic_sum(a*q^k, k, 0, oo)
-a/(q - 1)
```

```
>>> from sage.all import *
>>> assume(abs(q) < Integer(1))
>>> symbolic_sum(a*q**k, k, Integer(0), oo)
-a/(q - 1)
```

A divergent geometric series. Don't forget to forget your assumptions:

```
sage: forget()
sage: assume(q > 1)
sage: symbolic_sum(a*q^k, k, 0, oo)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
sage: forget()
sage: assumptions()  # check the assumptions were really forgotten
[]
```

```
>>> from sage.all import *
>>> forget()
>>> assume(q > Integer(1))
>>> symbolic_sum(a*q**k, k, Integer(0), oo)
Traceback (most recent call last):
...
ValueError: Sum is divergent.
>>> forget()
>>> assumptions()  # check the assumptions were really forgotten
[]
```

A summation performed by Mathematica:

```
sage: symbolic_sum(1/(1+k^2), k, -oo, oo, algorithm='mathematica')      # optional␣
↪- mathematica
pi*coth(pi)
```

```
>>> from sage.all import *
>>> symbolic_sum(Integer(1)/(Integer(1)+k**Integer(2)), k, -oo, oo, algorithm=
↪'mathematica')      # optional - mathematica
pi*coth(pi)
```

An example of this summation with Giac:

```
sage: # needs giac
sage: symbolic_sum(1/(1+k^2), k, -oo, oo, algorithm='giac').factor()
pi*(e^(2*pi) + 1)/((e^pi + 1)*(e^pi - 1))
```

```
>>> from sage.all import *
>>> # needs giac
>>> symbolic_sum(Integer(1)/(Integer(1)+k**Integer(2)), k, -oo, oo, algorithm=
```

(continues on next page)

(continued from previous page)

```
↪'giac').factor()
pi*(e^(2*pi) + 1)/((e^pi + 1)*(e^pi - 1))
```

The same summation is solved by SymPy:

```
sage: symbolic_sum(1/(1+k^2), k, -oo, oo, algorithm='sympy')
pi/tanh(pi)
```

```
>>> from sage.all import *
>>> symbolic_sum(Integer(1)/(Integer(1)+k**Integer(2)), k, -oo, oo, algorithm=
↪'sympy')
pi/tanh(pi)
```

SymPy and Maxima 5.39.0 can do the following (see Issue #22005):

```
sage: sum(1/((2*n+1)^2-4)^2, n, 0, Infinity, algorithm='sympy')
1/64*pi^2
sage: sum(1/((2*n+1)^2-4)^2, n, 0, Infinity)
1/64*pi^2
```

```
>>> from sage.all import *
>>> sum(Integer(1)/((Integer(2)*n+Integer(1))**Integer(2)-Integer(4))**Integer(2),
↪ n, Integer(0), Infinity, algorithm='sympy')
1/64*pi^2
>>> sum(Integer(1)/((Integer(2)*n+Integer(1))**Integer(2)-Integer(4))**Integer(2),
↪ n, Integer(0), Infinity)
1/64*pi^2
```

Use Maple as a backend for summation:

```
sage: symbolic_sum(binomial(n,k)*x^k, k, 0, n, algorithm='maple')      # optional␣
↪- maple
(x + 1)^n
```

```
>>> from sage.all import *
>>> symbolic_sum(binomial(n,k)*x**k, k, Integer(0), n, algorithm='maple')      #␣
↪optional - maple
(x + 1)^n
```

If you don't want to evaluate immediately give the `hold` keyword:

```
sage: s = sum(n, n, 1, k, hold=True); s
sum(n, n, 1, k)
sage: s.unhold()
1/2*k^2 + 1/2*k
sage: s.subs(k == 10)
sum(n, n, 1, 10)
sage: s.subs(k == 10).unhold()
55
sage: s.subs(k == 10).n()
55.0000000000000
```

```
>>> from sage.all import *
>>> s = sum(n, n, Integer(1), k, hold=True); s
sum(n, n, 1, k)
>>> s.unhold()
1/2*k^2 + 1/2*k
>>> s.subs(k == Integer(10))
sum(n, n, 1, 10)
>>> s.subs(k == Integer(10)).unhold()
55
>>> s.subs(k == Integer(10)).n()
55.0000000000000
```

Note

Sage can currently only understand a subset of the output of Maxima, Maple and Mathematica, so even if the chosen backend can perform the summation the result might not be convertible into a Sage expression.

2.7 Units of measurement

This is the units package. It contains information about many units and conversions between them.

TUTORIAL:

To return a unit:

```
sage: units.length.meter
meter
```

```
>>> from sage.all import *
>>> units.length.meter
meter
```

This unit acts exactly like a symbolic variable:

```
sage: s = units.length.meter
sage: s^2
meter^2
sage: s + var('x')
meter + x
```

```
>>> from sage.all import *
>>> s = units.length.meter
>>> s**Integer(2)
meter^2
>>> s + var('x')
meter + x
```

Units have additional information in their docstring:

```
sage: # You would type: units.force.dyne?
sage: print(units.force.dyne.__doc__)
```

(continues on next page)

(continued from previous page)

```
CGS unit for force defined to be gram*centimeter/second^2.
Equal to 10^-5 newtons.
```

```
>>> from sage.all import *
>>> # You would type: units.force.dyne?
>>> print(units.force.dyne.__doc__)
CGS unit for force defined to be gram*centimeter/second^2.
Equal to 10^-5 newtons.
```

You may call the convert function with units:

```
sage: t = units.mass.gram*units.length.centimeter/units.time.second^2
sage: t.convert(units.mass.pound*units.length.foot/units.time.hour^2)
5400000000000/5760623099*(foot*pound/hour^2)
sage: t.convert(units.force.newton)
1/100000*newton
```

```
>>> from sage.all import *
>>> t = units.mass.gram*units.length.centimeter/units.time.second**Integer(2)
>>> t.convert(units.mass.pound*units.length.foot/units.time.hour**Integer(2))
5400000000000/5760623099*(foot*pound/hour^2)
>>> t.convert(units.force.newton)
1/100000*newton
```

Calling the convert function with no target returns base SI units:

```
sage: t.convert()
1/100000*kilogram*meter/second^2
```

```
>>> from sage.all import *
>>> t.convert()
1/100000*kilogram*meter/second^2
```

Giving improper units to convert to raises a `ValueError`:

```
sage: t.convert(units.charge.coulomb)
Traceback (most recent call last):
...
ValueError: Incompatible units
```

```
>>> from sage.all import *
>>> t.convert(units.charge.coulomb)
Traceback (most recent call last):
...
ValueError: Incompatible units
```

Converting temperatures works as well:

```
sage: s = 68*units.temperature.fahrenheit
sage: s.convert(units.temperature.celsius)
20*celsius
```

(continues on next page)

(continued from previous page)

```
sage: s.convert()
293.150000000000*kelvin
```

```
>>> from sage.all import *
>>> s = Integer(68)*units.temperature.fahrenheit
>>> s.convert(units.temperature.celsius)
20*celsius
>>> s.convert()
293.150000000000*kelvin
```

Trying to multiply temperatures by another unit then converting raises a `ValueError`:

```
sage: wrong = 50*units.temperature.celsius*units.length.foot
sage: wrong.convert()
Traceback (most recent call last):
...
ValueError: cannot convert
```

```
>>> from sage.all import *
>>> wrong = Integer(50)*units.temperature.celsius*units.length.foot
>>> wrong.convert()
Traceback (most recent call last):
...
ValueError: cannot convert
```

AUTHORS:

- David Ackerman
- William Stein

class `sage.symbolic.units.`**`UnitExpression`**

Bases: *`Expression`*

A symbolic unit.

EXAMPLES:

```
sage: acre = units.area.acre
sage: type(acre)
<class 'sage.symbolic.units.UnitExpression'>
```

```
>>> from sage.all import *
>>> acre = units.area.acre
>>> type(acre)
<class 'sage.symbolic.units.UnitExpression'>
```

class `sage.symbolic.units.`**`Units`**(*data*, *name=''*)

Bases: `ExtraTabCompletion`

A collection of units of some type.

EXAMPLES:

```
sage: units.power
Collection of units of power: cheval_vapeur horsepower watt
```

```
>>> from sage.all import *
>>> units.power
Collection of units of power: cheval_vapeur horsepower watt
```

sage.symbolic.units.**base_units**(*unit*)

Convert unit to base SI units.

INPUT:

- `unit` – a unit

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: sage.symbolic.units.base_units(units.length.foot)
381/1250*meter
```

```
>>> from sage.all import *
>>> sage.symbolic.units.base_units(units.length.foot)
381/1250*meter
```

If unit is already a base unit, it just returns that unit:

```
sage: sage.symbolic.units.base_units(units.length.meter)
meter
```

```
>>> from sage.all import *
>>> sage.symbolic.units.base_units(units.length.meter)
meter
```

Derived units get broken down into their base parts:

```
sage: sage.symbolic.units.base_units(units.force.newton)
kilogram*meter/second^2
sage: sage.symbolic.units.base_units(units.volume.liter)
1/1000*meter^3
```

```
>>> from sage.all import *
>>> sage.symbolic.units.base_units(units.force.newton)
kilogram*meter/second^2
>>> sage.symbolic.units.base_units(units.volume.liter)
1/1000*meter^3
```

Returns variable if `unit` is not a unit:

```
sage: sage.symbolic.units.base_units(var('x'))
x
```

```
>>> from sage.all import *
>>> sage.symbolic.units.base_units(var('x'))
x
```

sage.symbolic.units.**convert**(*expr*, *target*)

Convert units between `expr` and `target`. If `target` is `None` then converts to SI base units.

INPUT:

- `expr` – the symbolic expression converting from
- `target` – (default: `None`) the symbolic expression converting to

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: sage.symbolic.units.convert(units.length.foot, None)
381/1250*meter
sage: sage.symbolic.units.convert(units.mass.kilogram, units.mass.pound)
100000000/45359237*pound
```

```
>>> from sage.all import *
>>> sage.symbolic.units.convert(units.length.foot, None)
381/1250*meter
>>> sage.symbolic.units.convert(units.mass.kilogram, units.mass.pound)
100000000/45359237*pound
```

This raises `ValueError` if expr and target are not convertible:

```
sage: sage.symbolic.units.convert(units.mass.kilogram, units.length.foot)
Traceback (most recent call last):
...
ValueError: Incompatible units
sage: sage.symbolic.units.convert(units.length.meter^2, units.length.foot)
Traceback (most recent call last):
...
ValueError: Incompatible units
```

```
>>> from sage.all import *
>>> sage.symbolic.units.convert(units.mass.kilogram, units.length.foot)
Traceback (most recent call last):
...
ValueError: Incompatible units
>>> sage.symbolic.units.convert(units.length.meter**Integer(2), units.length.foot)
Traceback (most recent call last):
...
ValueError: Incompatible units
```

Recognizes derived unit relationships to base units and other derived units:

```
sage: sage.symbolic.units.convert(units.length.foot/units.time.second^2, units.
↪acceleration.galileo)
762/25*galileo
sage: sage.symbolic.units.convert(units.mass.kilogram*units.length.meter/units.
```

(continues on next page)

(continued from previous page)

```
↪time.second^2, units.force.newton)
newton
sage: sage.symbolic.units.convert(units.length.foot^3, units.area.acre*units.
↪length.inch)
1/3630*(acre*inch)
sage: sage.symbolic.units.convert(units.charge.coulomb, units.current.
↪ampere*units.time.second)
(ampere*second)
sage: sage.symbolic.units.convert(units.pressure.pascal*units.si_prefixes.kilo,␣
↪units.pressure.pounds_per_square_inch)
1290320000000/8896443230521*pounds_per_square_inch
```

```
>>> from sage.all import *
>>> sage.symbolic.units.convert(units.length.foot/units.time.second**Integer(2),␣
↪units.acceleration.galileo)
762/25*galileo
>>> sage.symbolic.units.convert(units.mass.kilogram*units.length.meter/units.time.
↪second**Integer(2), units.force.newton)
newton
>>> sage.symbolic.units.convert(units.length.foot**Integer(3), units.area.
↪acre*units.length.inch)
1/3630*(acre*inch)
>>> sage.symbolic.units.convert(units.charge.coulomb, units.current.ampere*units.
↪time.second)
(ampere*second)
>>> sage.symbolic.units.convert(units.pressure.pascal*units.si_prefixes.kilo,␣
↪units.pressure.pounds_per_square_inch)
1290320000000/8896443230521*pounds_per_square_inch
```

For decimal answers multiply 1.0:

```
sage: sage.symbolic.units.convert(units.pressure.pascal*units.si_prefixes.kilo,␣
↪units.pressure.pounds_per_square_inch)*1.0
0.145037737730209*pounds_per_square_inch
```

```
>>> from sage.all import *
>>> sage.symbolic.units.convert(units.pressure.pascal*units.si_prefixes.kilo,␣
↪units.pressure.pounds_per_square_inch)*RealNumber('1.0')
0.145037737730209*pounds_per_square_inch
```

You can also convert quantities of units:

```
sage: sage.symbolic.units.convert(cos(50) * units.angles.radian, units.angles.
↪degree)
degree*(180*cos(50)/pi)
sage: sage.symbolic.units.convert(cos(30) * units.angles.radian, units.angles.
↪degree).polynomial(RR)
8.83795706233228*degree
sage: sage.symbolic.units.convert(50 * units.length.light_year / units.time.year,␣
↪units.length.foot / units.time.second)
6249954068750/127*(foot/second)
```

```
>>> from sage.all import *
>>> sage.symbolic.units.convert(cos(Integer(50)) * units.angles.radian, units.
↪angles.degree)
degree*(180*cos(50)/pi)
>>> sage.symbolic.units.convert(cos(Integer(30)) * units.angles.radian, units.
↪angles.degree).polynomial(RR)
8.83795706233228*degree
>>> sage.symbolic.units.convert(Integer(50) * units.length.light_year / units.
↪time.year, units.length.foot / units.time.second)
6249954068750/127*(foot/second)
```

Quantities may contain variables (not for temperature conversion, though):

```
sage: sage.symbolic.units.convert(50 * x * units.area.square_meter, units.area.
↪acre)
acre*(1953125/158080329*x)
```

```
>>> from sage.all import *
>>> sage.symbolic.units.convert(Integer(50) * x * units.area.square_meter, units.
↪area.acre)
acre*(1953125/158080329*x)
```

sage.symbolic.units.**convert_temperature**(*expr*, *target*)

Function for converting between temperatures.

INPUT:

- `expr` – a unit of temperature
- `target` – a units of temperature

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: t = 32*units.temperature.fahrenheit
sage: t.convert(units.temperature.celsius)
0
sage: t.convert(units.temperature.kelvin)
273.150000000000*kelvin
```

```
>>> from sage.all import *
>>> t = Integer(32)*units.temperature.fahrenheit
>>> t.convert(units.temperature.celsius)
0
>>> t.convert(units.temperature.kelvin)
273.150000000000*kelvin
```

If target is `None` then it defaults to kelvin:

```
sage: t.convert()
273.150000000000*kelvin
```

```
>>> from sage.all import *
>>> t.convert()
273.150000000000*kelvin
```

This raises ValueError when either input is not a unit of temperature:

```
sage: t.convert(units.length.foot)
Traceback (most recent call last):
...
ValueError: cannot convert
sage: wrong = units.length.meter*units.temperature.fahrenheit
sage: wrong.convert()
Traceback (most recent call last):
...
ValueError: cannot convert
```

```
>>> from sage.all import *
>>> t.convert(units.length.foot)
Traceback (most recent call last):
...
ValueError: cannot convert
>>> wrong = units.length.meter*units.temperature.fahrenheit
>>> wrong.convert()
Traceback (most recent call last):
...
ValueError: cannot convert
```

We directly call the convert_temperature function:

```
sage: sage.symbolic.units.convert_temperature(37*units.temperature.celsius, units.
↪temperature.fahrenheit)
493/5*fahrenheit
sage: 493/5.0
98.6000000000000
```

```
>>> from sage.all import *
>>> sage.symbolic.units.convert_temperature(Integer(37)*units.temperature.celsius,
↪ units.temperature.fahrenheit)
493/5*fahrenheit
>>> Integer(493)/RealNumber('5.0')
98.6000000000000
```

sage.symbolic.units.**evalunitdict**()

Replace all the string values of the unitdict variable by their evaluated forms, and builds some other tables for ease of use. This function is mainly used internally, for efficiency (and flexibility) purposes, making it easier to describe the units.

EXAMPLES:

```
sage: sage.symbolic.units.evalunitdict()
```

```
>>> from sage.all import *
>>> sage.symbolic.units.evalunitdict()
```

sage.symbolic.units.**is_unit**(*s*)

Return a boolean when asked whether the input is in the list of units.

INPUT:

- s – an object

OUTPUT: boolean

EXAMPLES:

```
sage: sage.symbolic.units.is_unit(1)
False
sage: sage.symbolic.units.is_unit(units.length.meter)
True
```

```
>>> from sage.all import *
>>> sage.symbolic.units.is_unit(Integer(1))
False
>>> sage.symbolic.units.is_unit(units.length.meter)
True
```

The square of a unit is not a unit:

```
sage: sage.symbolic.units.is_unit(units.length.meter^2)
False
```

```
>>> from sage.all import *
>>> sage.symbolic.units.is_unit(units.length.meter**Integer(2))
False
```

You can also directly create units using var, though they won't have a nice docstring describing the unit:

```
sage: sage.symbolic.units.is_unit(var('meter'))
True
```

```
>>> from sage.all import *
>>> sage.symbolic.units.is_unit(var('meter'))
True
```

sage.symbolic.units.**str_to_unit**(*name*)

Create the symbolic unit with given name. A symbolic unit is a class that derives from symbolic expression, and has a specialized docstring.

INPUT:

- name – string

OUTPUT: a *UnitExpression*

EXAMPLES:

```
sage: sage.symbolic.units.str_to_unit('acre')
acre
sage: type(sage.symbolic.units.str_to_unit('acre'))
<class 'sage.symbolic.units.UnitExpression'>
```

```
>>> from sage.all import *
>>> sage.symbolic.units.str_to_unit('acre')
acre
>>> type(sage.symbolic.units.str_to_unit('acre'))
<class 'sage.symbolic.units.UnitExpression'>
```

sage.symbolic.units.**unit_derivations_expr**(*v*)

Given derived units name, returns the corresponding units expression.

For example, given `'acceleration'` output the symbolic expression `length/time^2`.

INPUT:

- `v` – string; name of a unit type such as `'area'`, `'volume'`, etc.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: sage.symbolic.units.unit_derivations_expr('volume')
length^3
sage: sage.symbolic.units.unit_derivations_expr('electric_potential')
length^2*mass/(current*time^3)
```

```
>>> from sage.all import *
>>> sage.symbolic.units.unit_derivations_expr('volume')
length^3
>>> sage.symbolic.units.unit_derivations_expr('electric_potential')
length^2*mass/(current*time^3)
```

If the unit name is unknown, a `KeyError` is raised:

```
sage: sage.symbolic.units.unit_derivations_expr('invalid')
Traceback (most recent call last):
...
KeyError: 'invalid'
```

```
>>> from sage.all import *
>>> sage.symbolic.units.unit_derivations_expr('invalid')
Traceback (most recent call last):
...
KeyError: 'invalid'
```

sage.symbolic.units.**unitdocs**(*unit*)

Return docstring for the given unit.

INPUT:

- `unit` – a unit

OUTPUT: string

EXAMPLES:

```
sage: sage.symbolic.units.unitdocs('meter')
'SI base unit of length.\nDefined to be the distance light travels in vacuum in 1/
↪299792458 of a second.'
```

(continues on next page)

(continued from previous page)

```
sage: sage.symbolic.units.unitdocs('amu')
'Abbreviation for atomic mass unit.\nApproximately equal to 1.660538782*10^-27␣
↪kilograms.'
```

```
>>> from sage.all import *
>>> sage.symbolic.units.unitdocs('meter')
'SI base unit of length.\nDefined to be the distance light travels in vacuum in 1/
↪299792458 of a second.'
>>> sage.symbolic.units.unitdocs('amu')
'Abbreviation for atomic mass unit.\nApproximately equal to 1.660538782*10^-27␣
↪kilograms.'
```

Units not in the list unit_docs will raise a `ValueError`:

```
sage: sage.symbolic.units.unitdocs('earth')
Traceback (most recent call last):
...
ValueError: no documentation exists for the unit earth
```

```
>>> from sage.all import *
>>> sage.symbolic.units.unitdocs('earth')
Traceback (most recent call last):
...
ValueError: no documentation exists for the unit earth
```

`sage.symbolic.units.`**`vars_in_str`**`(s)`

Given a string like `'mass/(length*time)'`, return the list `['mass', 'length', 'time']`.

INPUT:

- `s` – string

OUTPUT: list of strings (unit names)

EXAMPLES:

```
sage: sage.symbolic.units.vars_in_str('mass/(length*time)')
['mass', 'length', 'time']
```

```
>>> from sage.all import *
>>> sage.symbolic.units.vars_in_str('mass/(length*time)')
['mass', 'length', 'time']
```

2.8 The symbolic ring

class `sage.symbolic.ring.`**`NumpyToSRMorphism`**

Bases: `Morphism`

A morphism from numpy types to the symbolic ring.

class `sage.symbolic.ring.`**`SymbolicRing`**

Bases: `SymbolicRing`

Symbolic Ring, parent object for all symbolic expressions.

I()

The imaginary unit, viewed as an element of the symbolic ring.

EXAMPLES:

```
sage: SR.I()^2
-1
sage: SR.I().parent()
Symbolic Ring
```

```
>>> from sage.all import *
>>> SR.I()**Integer(2)
-1
>>> SR.I().parent()
Symbolic Ring
```

characteristic()

Return the characteristic of the symbolic ring, which is 0.

OUTPUT: a Sage integer

EXAMPLES:

```
sage: c = SR.characteristic(); c
0
sage: type(c)
<class 'sage.rings.integer.Integer'>
```

```
>>> from sage.all import *
>>> c = SR.characteristic(); c
0
>>> type(c)
<class 'sage.rings.integer.Integer'>
```

cleanup_var(*symbol*)

Cleans up a variable, removing assumptions about the variable and allowing for it to be garbage collected

INPUT:

- `symbol` – a variable or a list of variables

is_exact()

Return `False`, because there are approximate elements in the symbolic ring.

EXAMPLES:

```
sage: SR.is_exact()
False
```

```
>>> from sage.all import *
>>> SR.is_exact()
False
```

Here is an inexact element.

```
sage: SR(1.9393)
1.93930000000000
```

```
>>> from sage.all import *
>>> SR(RealNumber('1.9393'))
1.93930000000000
```

is_field(*proof=True*)

Return True, since the symbolic expression ring is (for the most part) a field.

EXAMPLES:

```
sage: SR.is_field()
True
```

```
>>> from sage.all import *
>>> SR.is_field()
True
```

is_finite()

Return False, since the Symbolic Ring is infinite.

EXAMPLES:

```
sage: SR.is_finite()
False
```

```
>>> from sage.all import *
>>> SR.is_finite()
False
```

pi()

EXAMPLES:

```
sage: SR.pi() is pi
True
```

```
>>> from sage.all import *
>>> SR.pi() is pi
True
```

subring(**args*, ***kwds*)

Create a subring of this symbolic ring.

INPUT:

Choose one of the following keywords to create a subring.

- `accepting_variables` – (default: `None`) a tuple or other iterable of variables. If specified, then a symbolic subring of expressions in only these variables is created.
- `rejecting_variables` – (default: `None`) a tuple or other iterable of variables. If specified, then a symbolic subring of expressions in variables distinct to these variables is created.
- `no_variables` – boolean (default: `False`); if set, then a symbolic subring of constant expressions (i.e., expressions without a variable) is created.

OUTPUT: a ring

EXAMPLES:

Let us create a couple of symbolic variables first:

```
sage: V = var('a, b, r, s, x, y')
```

```
>>> from sage.all import *
>>> V = var('a, b, r, s, x, y')
```

Now we create a symbolic subring only accepting expressions in the variables a and b:

```
sage: A = SR.subring(accepting_variables=(a, b)); A
Symbolic Subring accepting the variables a, b
```

```
>>> from sage.all import *
>>> A = SR.subring(accepting_variables=(a, b)); A
Symbolic Subring accepting the variables a, b
```

An element is

```
sage: A.an_element()
a
```

```
>>> from sage.all import *
>>> A.an_element()
a
```

From our variables in V the following are valid in A:

```
sage: tuple(v for v in V if v in A)
(a, b)
```

```
>>> from sage.all import *
>>> tuple(v for v in V if v in A)
(a, b)
```

Next, we create a symbolic subring rejecting expressions with given variables:

```
sage: R = SR.subring(rejecting_variables=(r, s)); R
Symbolic Subring rejecting the variables r, s
```

```
>>> from sage.all import *
>>> R = SR.subring(rejecting_variables=(r, s)); R
Symbolic Subring rejecting the variables r, s
```

An element is

```
sage: R.an_element()
some_variable
```

```
>>> from sage.all import *
>>> R.an_element()
some_variable
```

From our variables in V the following are valid in R:

```
sage: tuple(v for v in V if v in R)
(a, b, x, y)
```

```
>>> from sage.all import *
>>> tuple(v for v in V if v in R)
(a, b, x, y)
```

We have a third kind of subring, namely the subring of symbolic constants:

```
sage: C = SR.subring(no_variables=True); C
Symbolic Constants Subring
```

```
>>> from sage.all import *
>>> C = SR.subring(no_variables=True); C
Symbolic Constants Subring
```

Note that this subring can be considered as a special accepting subring; one without any variables.

An element is

```
sage: C.an_element()
I*pi*e
```

```
>>> from sage.all import *
>>> C.an_element()
I*pi*e
```

None of our variables in V is valid in C:

```
sage: tuple(v for v in V if v in C)
()
```

```
>>> from sage.all import *
>>> tuple(v for v in V if v in C)
()
```

See also

Subrings of the Symbolic Ring

symbol(*name=None*, *latex_name=None*, *domain=None*)

EXAMPLES:

```
sage: t0 = SR.symbol("t0")
sage: t0.conjugate()
```

(continues on next page)

(continued from previous page)

```
conjugate(t0)

sage: t1 = SR.symbol("t1", domain='real')
sage: t1.conjugate()
t1

sage: t0.abs()
abs(t0)

sage: t0_2 = SR.symbol("t0", domain='positive')
sage: t0_2.abs()
t0
sage: bool(t0_2 == t0)
True
sage: t0.conjugate()
t0

sage: SR.symbol() # temporary variable
symbol...
```

```
>>> from sage.all import *
>>> t0 = SR.symbol("t0")
>>> t0.conjugate()
conjugate(t0)

>>> t1 = SR.symbol("t1", domain='real')
>>> t1.conjugate()
t1

>>> t0.abs()
abs(t0)

>>> t0_2 = SR.symbol("t0", domain='positive')
>>> t0_2.abs()
t0
>>> bool(t0_2 == t0)
True
>>> t0.conjugate()
t0

>>> SR.symbol() # temporary variable
symbol...
```

We propagate the domain to the assumptions database:

```
sage: n = var('n', domain='integer')
sage: solve([n^2 == 3],n)
[]
```

```
>>> from sage.all import *
>>> n = var('n', domain='integer')
>>> solve([n**Integer(2) == Integer(3)],n)
```

(continues on next page)

(continued from previous page)

```
[]
```

symbols

temp_var(*n=None*, *domain=None*)

Return one or multiple new unique symbolic variables as an element of the symbolic ring. Use this instead of SR.var() if there is a possibility of name clashes occuring. Call SR.cleanup_var() once the variables are no longer needed or use a $withSR.temp_var()as...$ construct.

INPUT:

- n – (optional) positive integer; number of symbolic variables
- domain – (optional) specify the domain of the variable(s);

EXAMPLES:

Simple definition of a functional derivative:

```
sage: def functional_derivative(expr, f, x):
....:     with SR.temp_var() as a:
....:         return expr.subs({f(x):a}).diff(a).subs({a:f(x)})
sage: f = function('f')
sage: a = var('a')
sage: functional_derivative(f(a)^2+a,f,a)
2*f(a)
```

```
>>> from sage.all import *
>>> def functional_derivative(expr, f, x):
...     with SR.temp_var() as a:
...         return expr.subs({f(x):a}).diff(a).subs({a:f(x)})
>>> f = function('f')
>>> a = var('a')
>>> functional_derivative(f(a)**Integer(2)+a,f,a)
2*f(a)
```

Contrast this to a similar implementation using SR.var(), which gives a wrong result in our example:

```
sage: def functional_derivative(expr, f, x):
....:     a = SR.var('a')
....:     return expr.subs({f(x):a}).diff(a).subs({a:f(x)})
sage: f = function('f')
sage: a = var('a')
sage: functional_derivative(f(a)^2+a,f,a)
2*f(a) + 1
```

```
>>> from sage.all import *
>>> def functional_derivative(expr, f, x):
...     a = SR.var('a')
...     return expr.subs({f(x):a}).diff(a).subs({a:f(x)})
>>> f = function('f')
>>> a = var('a')
>>> functional_derivative(f(a)**Integer(2)+a,f,a)
2*f(a) + 1
```

var(*name*, *latex_name=None*, *n=None*, *domain=None*)

Return a symbolic variable as an element of the symbolic ring.

INPUT:

- `name` – string or list of strings with the name(s) of the symbolic variable(s)
- `latex_name` – (optional) string used when printing in latex mode, if not specified use `'name'`
- `n` – (optional) positive integer; number of symbolic variables, indexed from 0 to $n - 1$
- `domain` – (optional) specify the domain of the variable(s); it is None by default, and possible options are (non-exhaustive list, see note below): `'real'`, `'complex'`, `'positive'`, `'integer'` and `'noninteger'`

OUTPUT: symbolic expression or tuple of symbolic expressions

See also

This function does not inject the variable(s) into the global namespace. For that purpose see `var()`.

Note

For a comprehensive list of acceptable features type `'maxima('features')'`, and see also the documentation of *Assumptions*.

EXAMPLES:

Create a variable zz:

```
sage: zz = SR.var('zz'); zz
zz
```

```
>>> from sage.all import *
>>> zz = SR.var('zz'); zz
zz
```

The return type is a symbolic expression:

```
sage: type(zz)
<class 'sage.symbolic.expression.Expression'>
```

```
>>> from sage.all import *
>>> type(zz)
<class 'sage.symbolic.expression.Expression'>
```

We can specify the domain as well:

```
sage: zz = SR.var('zz', domain='real')
sage: zz.is_real()
True
```

```
>>> from sage.all import *
>>> zz = SR.var('zz', domain='real')
>>> zz.is_real()
True
```

The real domain is also set with the integer domain:

```
sage: SR.var('x', domain='integer').is_real()
True
```

```
>>> from sage.all import *
>>> SR.var('x', domain='integer').is_real()
True
```

The `name` argument does not have to match the left-hand side variable:

```
sage: t = SR.var('theta2'); t
theta2
```

```
>>> from sage.all import *
>>> t = SR.var('theta2'); t
theta2
```

Automatic indexing is available as well:

```
sage: x = SR.var('x', 4)
sage: x[0], x[3]
(x0, x3)
sage: sum(x)
x0 + x1 + x2 + x3
```

```
>>> from sage.all import *
>>> x = SR.var('x', Integer(4))
>>> x[Integer(0)], x[Integer(3)]
(x0, x3)
>>> sum(x)
x0 + x1 + x2 + x3
```

wild(*n=0*)

Return the n-th wild-card for pattern matching and substitution.

INPUT:

- `n` – nonnegative integer

OUTPUT: n-th wildcard expression

EXAMPLES:

```
sage: x,y = var('x,y')
sage: w0 = SR.wild(0); w1 = SR.wild(1)
sage: pattern = sin(x)*w0*w1^2; pattern
$1^2*$0*sin(x)
sage: f = atan(sin(x)*3*x^2); f
```

(continues on next page)

(continued from previous page)

```
arctan(3*x^2*sin(x))
sage: f.has(pattern)
True
sage: f.subs(pattern == x^2)
arctan(x^2)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> w0 = SR.wild(Integer(0)); w1 = SR.wild(Integer(1))
>>> pattern = sin(x)*w0*w1**Integer(2); pattern
$1^2*$0*sin(x)
>>> f = atan(sin(x)*Integer(3)*x**Integer(2)); f
arctan(3*x^2*sin(x))
>>> f.has(pattern)
True
>>> f.subs(pattern == x**Integer(2))
arctan(x^2)
```

class sage.symbolic.ring.**TemporaryVariables**(*iterable=()*, */*)

Bases: tuple

Instances of this class can be used with Python *with* to automatically clean up after themselves.

class sage.symbolic.ring.**UnderscoreSageMorphism**

Bases: Morphism

A Morphism which constructs Expressions from an arbitrary Python object by calling the `_sage_()` method on the object.

EXAMPLES:

```
sage: # needs sympy
sage: import sympy
sage: from sage.symbolic.ring import UnderscoreSageMorphism
sage: b = sympy.var('b')
sage: f = UnderscoreSageMorphism(type(b), SR)
sage: f(b)
b
sage: _.parent()
Symbolic Ring
```

```
>>> from sage.all import *
>>> # needs sympy
>>> import sympy
>>> from sage.symbolic.ring import UnderscoreSageMorphism
>>> b = sympy.var('b')
>>> f = UnderscoreSageMorphism(type(b), SR)
>>> f(b)
b
>>> _.parent()
Symbolic Ring
```

sage.symbolic.ring.**isidentifier**(*x*)

Return whether x is a valid identifier.

INPUT:

- `x` – string

OUTPUT: boolean; whether the string `x` can be used as a variable name

This function should return `False` for keywords, so we can not just use the `isidentifier` method of strings, because, for example, it returns `True` for "def" and for "None".

EXAMPLES:

```
sage: from sage.symbolic.ring import isidentifier
sage: isidentifier('x')
True
sage: isidentifier(' x')    # can't start with space
False
sage: isidentifier('ceci_n_est_pas_une_pipe')
True
sage: isidentifier('1 + x')
False
sage: isidentifier('2good')
False
sage: isidentifier('good2')
True
sage: isidentifier('lambda s:s+1')
False
sage: isidentifier('None')
False
sage: isidentifier('lambda')
False
sage: isidentifier('def')
False
```

```
>>> from sage.all import *
>>> from sage.symbolic.ring import isidentifier
>>> isidentifier('x')
True
>>> isidentifier(' x')    # can't start with space
False
>>> isidentifier('ceci_n_est_pas_une_pipe')
True
>>> isidentifier('1 + x')
False
>>> isidentifier('2good')
False
>>> isidentifier('good2')
True
>>> isidentifier('lambda s:s+1')
False
>>> isidentifier('None')
False
>>> isidentifier('lambda')
False
>>> isidentifier('def')
False
```

sage.symbolic.ring.**the_SymbolicRing**()

Return the unique symbolic ring object.

(This is mainly used for unpickling.)

EXAMPLES:

```
sage: sage.symbolic.ring.the_SymbolicRing()
Symbolic Ring
sage: sage.symbolic.ring.the_SymbolicRing() is sage.symbolic.ring.the_
↪SymbolicRing()
True
sage: sage.symbolic.ring.the_SymbolicRing() is SR
True
```

```
>>> from sage.all import *
>>> sage.symbolic.ring.the_SymbolicRing()
Symbolic Ring
>>> sage.symbolic.ring.the_SymbolicRing() is sage.symbolic.ring.the_SymbolicRing()
True
>>> sage.symbolic.ring.the_SymbolicRing() is SR
True
```

sage.symbolic.ring.**var**(*name*, ***kwds*)

EXAMPLES:

```
sage: from sage.symbolic.ring import var
sage: var("x y z")
(x, y, z)
sage: var("x,y,z")
(x, y, z)
sage: var("x , y , z")
(x, y, z)
sage: var("z")
z
```

```
>>> from sage.all import *
>>> from sage.symbolic.ring import var
>>> var("x y z")
(x, y, z)
>>> var("x,y,z")
(x, y, z)
>>> var("x , y , z")
(x, y, z)
>>> var("z")
z
```

2.9 Subrings of the Symbolic Ring

Subrings of the symbolic ring can be created via the *subring()* method of SR. This will call *SymbolicSubring* of this module.

The following kinds of subrings are supported:

- A symbolic subring of expressions, whose variables are contained in a given set of symbolic variables (see *SymbolicSubringAcceptingVars*). E.g.

```
sage: SR.subring(accepting_variables=('a', 'b'))
Symbolic Subring accepting the variables a, b
```

```
>>> from sage.all import *
>>> SR.subring(accepting_variables=('a', 'b'))
Symbolic Subring accepting the variables a, b
```

- A symbolic subring of expressions, whose variables are disjoint to a given set of symbolic variables (see *SymbolicSubringRejectingVars*). E.g.

```
sage: SR.subring(rejecting_variables=('r', 's'))
Symbolic Subring rejecting the variables r, s
```

```
>>> from sage.all import *
>>> SR.subring(rejecting_variables=('r', 's'))
Symbolic Subring rejecting the variables r, s
```

- The subring of symbolic constants (see *SymbolicConstantsSubring*). E.g.

```
sage: SR.subring(no_variables=True)
Symbolic Constants Subring
```

```
>>> from sage.all import *
>>> SR.subring(no_variables=True)
Symbolic Constants Subring
```

AUTHORS:

- Daniel Krenn (2015)

2.9.1 Classes and Methods

class sage.symbolic.subring.**GenericSymbolicSubring**(*vars*)

Bases: *SymbolicRing*

An abstract base class for a symbolic subring.

INPUT:

- `vars` – tuple of symbolic variables

has_valid_variable(*variable*)

Return whether the given `variable` is valid in this subring.

INPUT:

- `variable` – a symbolic variable

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.symbolic.subring import GenericSymbolicSubring
sage: GenericSymbolicSubring(vars=tuple()).has_valid_variable(x)
Traceback (most recent call last):
...
NotImplementedError: Not implemented in this abstract base class
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import GenericSymbolicSubring
>>> GenericSymbolicSubring(vars=tuple()).has_valid_variable(x)
Traceback (most recent call last):
...
NotImplementedError: Not implemented in this abstract base class
```

class sage.symbolic.subring.**GenericSymbolicSubringFunctor**(*vars*)

Bases: `ConstructionFunctor`

A base class for the functors constructing symbolic subrings.

INPUT:

- `vars` – tuple, set, or other iterable of symbolic variables

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: SymbolicSubring(no_variables=True).construction()[0]   # indirect doctest
Subring<accepting no variable>
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> SymbolicSubring(no_variables=True).construction()[Integer(0)]   # indirect␣
↪doctest
Subring<accepting no variable>
```

See also

`sage.categories.pushout.ConstructionFunctor`.

coercion_reversed = True

merge(*other*)

Merge this functor with `other` if possible.

INPUT:

- `other` – a functor

OUTPUT: a functor or `None`

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: F = SymbolicSubring(accepting_variables=('a',)).construction()[0]
sage: F.merge(F) is F
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> F = SymbolicSubring(accepting_variables=('a',)).construction()[Integer(0)]
>>> F.merge(F) is F
True
```

rank = 11

class sage.symbolic.subring.**SymbolicConstantsSubring**(*vars*)

Bases: *SymbolicSubringAcceptingVars*

The symbolic subring consisting of symbolic constants.

has_valid_variable(*variable*)

Return whether the given `variable` is valid in this subring.

INPUT:

- `variable` – a symbolic variable

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: S = SymbolicSubring(no_variables=True)
sage: S.has_valid_variable('a')
False
sage: S.has_valid_variable('r')
False
sage: S.has_valid_variable('x')
False
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> S = SymbolicSubring(no_variables=True)
>>> S.has_valid_variable('a')
False
>>> S.has_valid_variable('r')
False
>>> S.has_valid_variable('x')
False
```

class sage.symbolic.subring.**SymbolicSubringAcceptingVars**(*vars*)

Bases: *GenericSymbolicSubring*

The symbolic subring consisting of symbolic expressions in the given variables.

construction()

Return the functorial construction of this symbolic subring.

OUTPUT:

A tuple whose first entry is a construction functor and its second is the symbolic ring.

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: SymbolicSubring(accepting_variables=('a',)).construction()
(Subring<accepting a>, Symbolic Ring)
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> SymbolicSubring(accepting_variables=('a',)).construction()
(Subring<accepting a>, Symbolic Ring)
```

has_valid_variable(*variable*)

Return whether the given `variable` is valid in this subring.

INPUT:

- `variable` – a symbolic variable

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: S = SymbolicSubring(accepting_variables=('a',))
sage: S.has_valid_variable('a')
True
sage: S.has_valid_variable('r')
False
sage: S.has_valid_variable('x')
False
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> S = SymbolicSubring(accepting_variables=('a',))
>>> S.has_valid_variable('a')
True
>>> S.has_valid_variable('r')
False
>>> S.has_valid_variable('x')
False
```

class sage.symbolic.subring.**SymbolicSubringAcceptingVarsFunctor**(*vars*)

Bases: *GenericSymbolicSubringFunctor*

merge(*other*)

Merge this functor with `other` if possible.

INPUT:

- `other` – a functor

OUTPUT: a functor or `None`

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: F = SymbolicSubring(accepting_variables=('a',)).construction()[0]
sage: G = SymbolicSubring(rejecting_variables=('r',)).construction()[0]
```

(continues on next page)

(continued from previous page)

```
sage: F.merge(F) is F
True
sage: F.merge(G) is G
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> F = SymbolicSubring(accepting_variables=('a',)).construction()[Integer(0)]
>>> G = SymbolicSubring(rejecting_variables=('r',)).construction()[Integer(0)]
>>> F.merge(F) is F
True
>>> F.merge(G) is G
True
```

class sage.symbolic.subring.**SymbolicSubringFactory**

Bases: UniqueFactory

A factory creating a symbolic subring.

INPUT:

Specify one of the following keywords to create a subring.

- accepting_variables – (default: None) a tuple or other iterable of variables. If specified, then a symbolic subring of expressions in only these variables is created.
- rejecting_variables – (default: None) a tuple or other iterable of variables. If specified, then a symbolic subring of expressions in variables distinct to these variables is created.
- no_variables – boolean (default: False); if set, then a symbolic subring of constant expressions (i.e., expressions without a variable) is created.

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: V = var('a, b, c, r, s, t, x, y, z')
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> V = var('a, b, c, r, s, t, x, y, z')
```

```
sage: A = SymbolicSubring(accepting_variables=(a, b, c)); A
Symbolic Subring accepting the variables a, b, c
sage: tuple((v, v in A) for v in V)
((a, True), (b, True), (c, True),
 (r, False), (s, False), (t, False),
 (x, False), (y, False), (z, False))
```

```
>>> from sage.all import *
>>> A = SymbolicSubring(accepting_variables=(a, b, c)); A
Symbolic Subring accepting the variables a, b, c
>>> tuple((v, v in A) for v in V)
((a, True), (b, True), (c, True),
 (r, False), (s, False), (t, False),
 (x, False), (y, False), (z, False))
```

```
sage: R = SymbolicSubring(rejecting_variables=(r, s, t)); R
Symbolic Subring rejecting the variables r, s, t
sage: tuple((v, v in R) for v in V)
((a, True), (b, True), (c, True),
 (r, False), (s, False), (t, False),
 (x, True), (y, True), (z, True))
```

```
>>> from sage.all import *
>>> R = SymbolicSubring(rejecting_variables=(r, s, t)); R
Symbolic Subring rejecting the variables r, s, t
>>> tuple((v, v in R) for v in V)
((a, True), (b, True), (c, True),
 (r, False), (s, False), (t, False),
 (x, True), (y, True), (z, True))
```

```
sage: C = SymbolicSubring(no_variables=True); C
Symbolic Constants Subring
sage: tuple((v, v in C) for v in V)
((a, False), (b, False), (c, False),
 (r, False), (s, False), (t, False),
 (x, False), (y, False), (z, False))
```

```
>>> from sage.all import *
>>> C = SymbolicSubring(no_variables=True); C
Symbolic Constants Subring
>>> tuple((v, v in C) for v in V)
((a, False), (b, False), (c, False),
 (r, False), (s, False), (t, False),
 (x, False), (y, False), (z, False))
```

create_key_and_extra_args(*accepting_variables=None*, *rejecting_variables=None*, *no_variables=False*, ***kwds*)

Given the arguments and keyword, create a key that uniquely determines this object.

See *SymbolicSubringFactory* for details.

create_object(*version*, *key*, ***kwds*)

Create an object from the given arguments.

See *SymbolicSubringFactory* for details.

class sage.symbolic.subring.**SymbolicSubringRejectingVars**(*vars*)

Bases: *GenericSymbolicSubring*

The symbolic subring consisting of symbolic expressions whose variables are none of the given variables.

construction()

Return the functorial construction of this symbolic subring.

OUTPUT:

A tuple whose first entry is a construction functor and its second is the symbolic ring.

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: SymbolicSubring(rejecting_variables=('r',)).construction()
(Subring<rejecting r>, Symbolic Ring)
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> SymbolicSubring(rejecting_variables=('r',)).construction()
(Subring<rejecting r>, Symbolic Ring)
```

has_valid_variable(*variable*)

Return whether the given `variable` is valid in this subring.

INPUT:

- `variable` – a symbolic variable

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: S = SymbolicSubring(rejecting_variables=('r',))
sage: S.has_valid_variable('a')
True
sage: S.has_valid_variable('r')
False
sage: S.has_valid_variable('x')
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> S = SymbolicSubring(rejecting_variables=('r',))
>>> S.has_valid_variable('a')
True
>>> S.has_valid_variable('r')
False
>>> S.has_valid_variable('x')
True
```

class sage.symbolic.subring.**SymbolicSubringRejectingVarsFunctor**(*vars*)

Bases: *GenericSymbolicSubringFunctor*

merge(*other*)

Merge this functor with `other` if possible.

INPUT:

- `other` – a functor

OUTPUT: a functor or `None`

EXAMPLES:

```
sage: from sage.symbolic.subring import SymbolicSubring
sage: F = SymbolicSubring(accepting_variables=('a',)).construction()[0]
sage: G = SymbolicSubring(rejecting_variables=('r',)).construction()[0]
```

(continues on next page)

(continued from previous page)

```
sage: G.merge(G) is G
True
sage: G.merge(F) is G
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.subring import SymbolicSubring
>>> F = SymbolicSubring(accepting_variables=('a',)).construction()[Integer(0)]
>>> G = SymbolicSubring(rejecting_variables=('r',)).construction()[Integer(0)]
>>> G.merge(G) is G
True
>>> G.merge(F) is G
True
```

2.10 Classes for symbolic functions

To enable their usage as part of symbolic expressions, symbolic function classes are derived from one of the subclasses of *Function*:

- *BuiltinFunction*: the code of these functions is written in Python; many special functions are of this type
- *GinacFunction*: the code of these functions is written in C++ and part of the Pynac support library; most elementary functions are of this type
- *SymbolicFunction*: symbolic functions defined on the Sage command line are of this type

Sage uses `BuiltinFunction` and `GinacFunction` for its symbolic builtin functions. Users can define any other additional `SymbolicFunction` through the `function()` factory, see *Factory for symbolic functions*

Several parameters are supported by the superclass' `__init__()` method. Examples follow below.

- `nargs`: the number of arguments
- `name`: the string that is printed on the CLI; the name of the member functions that are attempted for evaluation of Sage element arguments; also the name of the Pynac function that is associated with a `GinacFunction`
- `alt_name`: the second name of the member functions that are attempted for evaluation of Sage element arguments
- `latex_name`: what is printed when `latex(f(...))` is called
- `conversions`: a dict containing the function's name in other CAS
- `evalf_params_first`: if `False`, when floating-point evaluating the expression do not evaluate function arguments before calling the `_evalf_()` member of the function
- `preserved_arg`: if nonzero, the index (starting with `1`) of the function argument that determines the return type. Note that, e.g, `atan2()` uses both arguments to determine return type, through a different mechanism

Function classes can define the following Python member functions:

- `_eval_(*args)`: the only mandatory member function, evaluating the argument and returning the result; if `None` is returned the expression stays unevaluated
- `_eval_numpy_(*args)`: evaluation of `f(args)` with arguments of numpy type
- `_evalf_(*args, **kwds)`: called when the expression is floating-point evaluated; may receive a `parent` keyword specifying the expected parent of the result. If not defined an attempt is made to convert the result of `_eval_()`.

- `_conjugate_(*args)`, `_real_part_(*args)`, `_imag_part_(*args)`: return conjugate, real part, imaginary part of the expression `f(args)`
- `_derivative_(*args, index)`: return derivative with respect to the parameter indexed by `index` (starting with 0) of `f(args)`
- `_tderivative_()`: same as `_derivative_()` but don't apply chain rule; only one of the two functions may be defined
- `_power_(*args, expo)`: return `f(args)^expo`
- `_series_(*args, **kwds)`: return the power series at `at` up to `order` with respect to `var` of `f(args)`; these three values are received in `kwds`. If not defined the series is attempted to be computed by differentiation.
- `print(*args)`: return what should be printed on the CLI with `f(args)`
- `print_latex(*args)`: return what should be output with `latex(f(args))`

The following examples are intended for Sage developers. Users can define functions interactively through the `function()` factory, see *Factory for symbolic functions*.

EXAMPLES:

The simplest example is a function returning nothing, it practically behaves like a symbol. Setting `nargs=0` allows any number of arguments:

```
sage: from sage.symbolic.function import BuiltinFunction
sage: class Test1(BuiltinFunction):
....:     def __init__(self):
....:         BuiltinFunction.__init__(self, 'test', nargs=0)
....:     def _eval_(self, *args):
....:         pass
sage: f = Test1()
sage: f()                                                                   #␣
↪needs sage.symbolic
test()
sage: f(1,2,3)*f(1,2,3)                                                     #␣
↪needs sage.symbolic
test(1, 2, 3)^2
```

```
>>> from sage.all import *
>>> from sage.symbolic.function import BuiltinFunction
>>> class Test1(BuiltinFunction):
...     def __init__(self):
...         BuiltinFunction.__init__(self, 'test', nargs=Integer(0))
...     def _eval_(self, *args):
...         pass
>>> f = Test1()
>>> f()                                                                     #␣
↪needs sage.symbolic
test()
>>> f(Integer(1),Integer(2),Integer(3))*f(Integer(1),Integer(2),Integer(3))       ␣
↪                                                   # needs sage.symbolic
test(1, 2, 3)^2
```

In the following the `sin` function of `CBF(0)` is called because with floating point arguments the `CBF` element's `my_sin()` member function is attempted, and after that `sin()` which succeeds:

```
sage: class Test2(BuiltinFunction):
....:     def __init__(self):
....:         BuiltinFunction.__init__(self, 'my_sin', alt_name='sin',
....:                                  latex_name=r'\SIN', nargs=1)
....:     def _eval_(self, x):
....:         return 5
....:     def _evalf_(self, x, **kwds):
....:         return 3.5
sage: f = Test2()
sage: f(0)
5
sage: f(0, hold=True)                                                    #␣
↪needs sage.symbolic
my_sin(0)
sage: f(0, hold=True).n()                                                #␣
↪needs sage.rings.real_mpfr
3.50000000000000
sage: f(CBF(0))                                                          #␣
↪needs sage.libs.flint
0

sage: latex(f(0, hold=True))                                             #␣
↪needs sage.symbolic
\SIN\left(0\right)
sage: f(1,2)
Traceback (most recent call last):
...
TypeError: Symbolic function my_sin takes exactly 1 arguments (2 given)
```

```
>>> from sage.all import *
>>> class Test2(BuiltinFunction):
...     def __init__(self):
...         BuiltinFunction.__init__(self, 'my_sin', alt_name='sin',
...                                  latex_name=r'\SIN', nargs=Integer(1))
...     def _eval_(self, x):
...         return Integer(5)
...     def _evalf_(self, x, **kwds):
...         return RealNumber('3.5')
>>> f = Test2()
>>> f(Integer(0))
5
>>> f(Integer(0), hold=True)                                             ␣
↪     # needs sage.symbolic
my_sin(0)
>>> f(Integer(0), hold=True).n()                                         ␣
↪     # needs sage.rings.real_mpfr
3.50000000000000
>>> f(CBF(Integer(0)))                                                   ␣
↪     # needs sage.libs.flint
0

>>> latex(f(Integer(0), hold=True))                                      ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.symbolic
\SIN\left(0\right)
>>> f(Integer(1),Integer(2))
Traceback (most recent call last):
...
TypeError: Symbolic function my_sin takes exactly 1 arguments (2 given)
```

class sage.symbolic.function.**BuiltinFunction**

Bases: *Function*

This is the base class for symbolic functions defined in Sage.

If a function is provided by the Sage library, we don't need to pickle the custom methods, since we can just initialize the same library function again. This allows us to use Cython for custom methods.

We assume that each subclass of this class will define one symbolic function. Make sure you use subclasses and not just call the initializer of this class.

class sage.symbolic.function.**Function**

Bases: SageObject

Base class for symbolic functions defined through Pynac in Sage.

This is an abstract base class, with generic code for the interfaces and a `__call__()` method. Subclasses should implement the `_is_registered()` and `_register_function()` methods.

This class is not intended for direct use, instead use one of the subclasses *BuiltinFunction* or *SymbolicFunction*.

default_variable()

Return a default variable.

EXAMPLES:

```
sage: sin.default_variable()                                          #␣
↪needs sage.symbolic
x
```

```
>>> from sage.all import *
>>> sin.default_variable()                                        #␣
↪needs sage.symbolic
x
```

name()

Return the name of this function.

EXAMPLES:

```
sage: foo = function("foo", nargs=2)                                  #␣
↪needs sage.symbolic
sage: foo.name()                                                      #␣
↪needs sage.symbolic
'foo'
```

```
>>> from sage.all import *
>>> foo = function("foo", nargs=Integer(2))                            ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.symbolic
>>> foo.name()                                                          #␣
↪needs sage.symbolic
'foo'
```

number_of_arguments()

Return the number of arguments that this function takes.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: foo = function("foo", nargs=2)
sage: foo.number_of_arguments()
2
sage: foo(x, x)
foo(x, x)
sage: foo(x)
Traceback (most recent call last):
...
TypeError: Symbolic function foo takes exactly 2 arguments (1 given)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> foo = function("foo", nargs=Integer(2))
>>> foo.number_of_arguments()
2
>>> foo(x, x)
foo(x, x)
>>> foo(x)
Traceback (most recent call last):
...
TypeError: Symbolic function foo takes exactly 2 arguments (1 given)
```

variables()

Return the variables (of which there are none) present in this function.

EXAMPLES:

```
sage: sin.variables()
()
```

```
>>> from sage.all import *
>>> sin.variables()
()
```

class sage.symbolic.function.**GinacFunction**

Bases: *BuiltinFunction*

This class provides a wrapper around symbolic functions already defined in Pynac/GiNaC.

GiNaC provides custom methods for these functions defined at the C++ level. It is still possible to define new custom functionality or override those already defined.

There is also no need to register these functions.

class sage.symbolic.function.**SymbolicFunction**

Bases: *Function*

This is the basis for user defined symbolic functions. We try to pickle or hash the custom methods, so subclasses must be defined in Python not Cython.

sage.symbolic.function.**pickle_wrapper**(*f*)

Return a pickled version of the function f.

If f is None, just return None.

This is a wrapper around pickle_function().

EXAMPLES:

```
sage: from sage.symbolic.function import pickle_wrapper
sage: def f(x): return x*x
sage: isinstance(pickle_wrapper(f), bytes)
True
sage: pickle_wrapper(None) is None
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.function import pickle_wrapper
>>> def f(x): return x*x
>>> isinstance(pickle_wrapper(f), bytes)
True
>>> pickle_wrapper(None) is None
True
```

sage.symbolic.function.**unpickle_wrapper**(*p*)

Return a unpickled version of the function defined by p.

If p is None, just return None.

This is a wrapper around unpickle_function().

EXAMPLES:

```
sage: from sage.symbolic.function import pickle_wrapper, unpickle_wrapper
sage: def f(x): return x*x
sage: s = pickle_wrapper(f)
sage: g = unpickle_wrapper(s)
sage: g(2)
4
sage: unpickle_wrapper(None) is None
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.function import pickle_wrapper, unpickle_wrapper
>>> def f(x): return x*x
>>> s = pickle_wrapper(f)
>>> g = unpickle_wrapper(s)
>>> g(Integer(2))
4
>>> unpickle_wrapper(None) is None
True
```

2.11 Factory for symbolic functions

`sage.symbolic.function_factory.function(s, **kwds)`

Create a formal symbolic function with the name *s*.

INPUT:

- `nargs=0` – number of arguments the function accepts, defaults to variable number of arguments, or 0
- `latex_name` – name used when printing in latex mode
- `conversions` – dictionary specifying names of this function in other systems, this is used by the interfaces internally during conversion
- `eval_func` – method used for automatic evaluation
- `evalf_func` – method used for numeric evaluation
- `evalf_params_first` – boolean to indicate if parameters should be evaluated numerically before calling the custom evalf function
- `conjugate_func` – method used for complex conjugation
- `real_part_func` – method used when taking real parts
- `imag_part_func` – method used when taking imaginary parts
- `derivative_func` – method to be used for (partial) derivation This method should take a keyword argument deriv_param specifying the index of the argument to differentiate w.r.t
- `tderivative_func` – method to be used for derivatives
- `power_func` – method used when taking powers This method should take a keyword argument power_param specifying the exponent
- `series_func` – method used for series expansion This method should expect keyword arguments - `order` – order for the expansion to be computed - `var` – variable to expand w.r.t. - `at` – expand at this value
- `print_func` – method for custom printing
- `print_latex_func` – method for custom printing in latex mode

Note that custom methods must be instance methods, i.e., expect the instance of the symbolic function as the first argument.

EXAMPLES:

```
sage: from sage.symbolic.function_factory import function
sage: var('a, b')
(a, b)
sage: cr = function('cr')
sage: f = cr(a)
sage: g = f.diff(a).integral(b); g
b*diff(cr(a), a)
sage: foo = function("foo", nargs=2)
sage: x,y,z = var("x y z")
sage: foo(x, y) + foo(y, z)^2
foo(y, z)^2 + foo(x, y)
```

```
>>> from sage.all import *
>>> from sage.symbolic.function_factory import function
```

(continues on next page)

(continued from previous page)

```
>>> var('a, b')
(a, b)
>>> cr = function('cr')
>>> f = cr(a)
>>> g = f.diff(a).integral(b); g
b*diff(cr(a), a)
>>> foo = function("foo", nargs=Integer(2))
>>> x,y,z = var("x y z")
>>> foo(x, y) + foo(y, z)**Integer(2)
foo(y, z)^2 + foo(x, y)
```

You need to use `substitute_function()` to replace all occurrences of a function with another:

```
sage: g.substitute_function(cr, cos)
-b*sin(a)

sage: g.substitute_function(cr, (sin(x) + cos(x)).function(x))
b*(cos(a) - sin(a))
```

```
>>> from sage.all import *
>>> g.substitute_function(cr, cos)
-b*sin(a)

>>> g.substitute_function(cr, (sin(x) + cos(x)).function(x))
b*(cos(a) - sin(a))
```

Basic arithmetic with unevaluated functions is no longer supported:

```
sage: x = var('x')
sage: f = function('f')
sage: 2*f
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring' and
'<class 'sage.symbolic.function_factory...NewSymbolicFunction'>'
```

```
>>> from sage.all import *
>>> x = var('x')
>>> f = function('f')
>>> Integer(2)*f
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring' and
'<class 'sage.symbolic.function_factory...NewSymbolicFunction'>'
```

You now need to evaluate the function in order to do the arithmetic:

```
sage: 2*f(x)
2*f(x)
```

```
>>> from sage.all import *
>>> Integer(2)*f(x)
```

(continues on next page)

(continued from previous page)

```
2*f(x)
```

We create a formal function of one variable, write down an expression that involves first and second derivatives, and extract off coefficients.

```
sage: r, kappa = var('r,kappa')
sage: psi = function('psi', nargs=1)(r); psi
psi(r)
sage: g = 1/r^2*(2*r*psi.derivative(r,1) + r^2*psi.derivative(r,2)); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
sage: g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
sage: g.coefficient(psi.derivative(r,2))
1
sage: g.coefficient(psi.derivative(r,1))
2/r
```

```
>>> from sage.all import *
>>> r, kappa = var('r,kappa')
>>> psi = function('psi', nargs=Integer(1))(r); psi
psi(r)
>>> g = Integer(1)/r**Integer(2)*(Integer(2)*r*psi.derivative(r,Integer(1)) +␣
→r**Integer(2)*psi.derivative(r,Integer(2))); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
>>> g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
>>> g.coefficient(psi.derivative(r,Integer(2)))
1
>>> g.coefficient(psi.derivative(r,Integer(1)))
2/r
```

Defining custom methods for automatic or numeric evaluation, derivation, conjugation, etc. is supported:

```
sage: def ev(self, x): return 2*x
sage: foo = function("foo", nargs=1, eval_func=ev)
sage: foo(x)
2*x
sage: foo = function("foo", nargs=1, eval_func=lambda self, x: 5)
sage: foo(x)
5
sage: def ef(self, x): pass
sage: bar = function("bar", nargs=1, eval_func=ef)
sage: bar(x)
bar(x)

sage: def evalf_f(self, x, parent=None, algorithm=None): return 6
sage: foo = function("foo", nargs=1, evalf_func=evalf_f)
sage: foo(x)
foo(x)
sage: foo(x).n()
6
```

(continues on next page)

(continued from previous page)

```
sage: foo = function("foo", nargs=1, conjugate_func=ev)
sage: foo(x).conjugate()
2*x

sage: def deriv(self, *args, **kwds):
....:     print("{} {}".format(args, kwds))
....:     return args[kwds['diff_param']]^2
sage: foo = function("foo", nargs=2, derivative_func=deriv)
sage: foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2

sage: def pow(self, x, power_param=None):
....:     print("{} {}".format(x, power_param))
....:     return x*power_param
sage: foo = function("foo", nargs=1, power_func=pow)
sage: foo(y)^(x+y)
y x + y
(x + y)*y

sage: def expand(self, *args, **kwds):
....:     print("{} {}".format(args, sorted(kwds.items())))
....:     return sum(args[0]^i for i in range(kwds['order']))
sage: foo = function("foo", nargs=1, series_func=expand)
sage: foo(y).series(y, 5)
(y,) [('at', 0), ('options', 0), ('order', 5), ('var', y)]
y^4 + y^3 + y^2 + y + 1

sage: def my_print(self, *args): return "my args are: " + ', '.join(map(repr,␣
→args))
sage: foo = function('t', nargs=2, print_func=my_print)
sage: foo(x,y^z)
my args are: x, y^z

sage: latex(foo(x,y^z))
t\left(x, y^{z}\right)
sage: foo = function('t', nargs=2, print_latex_func=my_print)
sage: foo(x,y^z)
t(x, y^z)
sage: latex(foo(x,y^z))
my args are: x, y^z
sage: foo = function('t', nargs=2, latex_name='foo')
sage: latex(foo(x,y^z))
foo\left(x, y^{z}\right)
```

```
>>> from sage.all import *
>>> def ev(self, x): return Integer(2)*x
>>> foo = function("foo", nargs=Integer(1), eval_func=ev)
>>> foo(x)
2*x
>>> foo = function("foo", nargs=Integer(1), eval_func=lambda self, x: Integer(5))
>>> foo(x)
```

(continues on next page)

(continued from previous page)

```
5
>>> def ef(self, x): pass
>>> bar = function("bar", nargs=Integer(1), eval_func=ef)
>>> bar(x)
bar(x)

>>> def evalf_f(self, x, parent=None, algorithm=None): return Integer(6)
>>> foo = function("foo", nargs=Integer(1), evalf_func=evalf_f)
>>> foo(x)
foo(x)
>>> foo(x).n()
6

>>> foo = function("foo", nargs=Integer(1), conjugate_func=ev)
>>> foo(x).conjugate()
2*x

>>> def deriv(self, *args, **kwds):
...     print("{} {}".format(args, kwds))
...     return args[kwds['diff_param']]**Integer(2)
>>> foo = function("foo", nargs=Integer(2), derivative_func=deriv)
>>> foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2

>>> def pow(self, x, power_param=None):
...     print("{} {}".format(x, power_param))
...     return x*power_param
>>> foo = function("foo", nargs=Integer(1), power_func=pow)
>>> foo(y)**(x+y)
y x + y
(x + y)*y

>>> def expand(self, *args, **kwds):
...     print("{} {}".format(args, sorted(kwds.items())))
...     return sum(args[Integer(0)]**i for i in range(kwds['order']))
>>> foo = function("foo", nargs=Integer(1), series_func=expand)
>>> foo(y).series(y, Integer(5))
(y,) [('at', 0), ('options', 0), ('order', 5), ('var', y)]
y^4 + y^3 + y^2 + y + 1

>>> def my_print(self, *args): return "my args are: " + ', '.join(map(repr, args))
>>> foo = function('t', nargs=Integer(2), print_func=my_print)
>>> foo(x,y**z)
my args are: x, y^z

>>> latex(foo(x,y**z))
t\left(x, y^{z}\right)
>>> foo = function('t', nargs=Integer(2), print_latex_func=my_print)
>>> foo(x,y**z)
t(x, y^z)
>>> latex(foo(x,y**z))
```

(continues on next page)

(continued from previous page)

```
my args are: x, y^z
>>> foo = function('t', nargs=Integer(2), latex_name='foo')
>>> latex(foo(x,y**z))
foo\left(x, y^{z}\right)
```

Chain rule:

```
sage: def print_args(self, *args, **kwds): print("args: {}".format(args)); print(
↪"kwds: {}".format(kwds)); return args[0]
sage: foo = function('t', nargs=2, tderivative_func=print_args)
sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': x}
x
sage: foo = function('t', nargs=2, derivative_func=print_args)
sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
args: (x, x)
kwds: {'diff_param': 1}
2*x
```

```
>>> from sage.all import *
>>> def print_args(self, *args, **kwds): print("args: {}".format(args)); print(
↪"kwds: {}".format(kwds)); return args[Integer(0)]
>>> foo = function('t', nargs=Integer(2), tderivative_func=print_args)
>>> foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': x}
x
>>> foo = function('t', nargs=Integer(2), derivative_func=print_args)
>>> foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
args: (x, x)
kwds: {'diff_param': 1}
2*x
```

sage.symbolic.function_factory.**function_factory**(*name, nargs=0, latex_name=None, conversions=None, evalf_params_first=True, eval_func=None, evalf_func=None, conjugate_func=None, real_part_func=None, imag_part_func=None, derivative_func=None, tderivative_func=None, power_func=None, series_func=None, print_func=None, print_latex_func=None*)

Create a formal symbolic function. For an explanation of the arguments see the documentation for the method `function()`.

EXAMPLES:

```
sage: from sage.symbolic.function_factory import function_factory
```

(continues on next page)

(continued from previous page)

```
sage: f = function_factory('f', 2, '\\foo', {'mathematica':'Foo'})
sage: f(2,4)
f(2, 4)
sage: latex(f(1,2))
\foo\left(1, 2\right)
sage: f._mathematica_init_()
'Foo'

sage: def evalf_f(self, x, parent=None, algorithm=None): return x*.5r
sage: g = function_factory('g',1,evalf_func=evalf_f)
sage: g(2)
g(2)
sage: g(2).n()
1.00000000000000
```

```
>>> from sage.all import *
>>> from sage.symbolic.function_factory import function_factory
>>> f = function_factory('f', Integer(2), '\\foo', {'mathematica':'Foo'})
>>> f(Integer(2),Integer(4))
f(2, 4)
>>> latex(f(Integer(1),Integer(2)))
\foo\left(1, 2\right)
>>> f._mathematica_init_()
'Foo'

>>> def evalf_f(self, x, parent=None, algorithm=None): return x*.5
>>> g = function_factory('g',Integer(1),evalf_func=evalf_f)
>>> g(Integer(2))
g(2)
>>> g(Integer(2)).n()
1.00000000000000
```

`sage.symbolic.function_factory.`**`unpickle_function`**`(`*name*, *nargs*, *latex_name*, *conversions*, *evalf_params_first*, *pickled_funcs*`)`

This is returned by the `__reduce__` method of symbolic functions to be called during unpickling to recreate the given function.

It calls `function_factory()` with the supplied arguments.

EXAMPLES:

```
sage: from sage.symbolic.function_factory import unpickle_function
sage: nf = unpickle_function('f', 2, '\\foo', {'mathematica':'Foo'}, True, [])
sage: nf
f
sage: nf(1,2)
f(1, 2)
sage: latex(nf(x,x))
\foo\left(x, x\right)
sage: nf._mathematica_init_()
'Foo'

sage: from sage.symbolic.function import pickle_wrapper
```

(continues on next page)

(continued from previous page)

```
sage: def evalf_f(self, x, parent=None, algorithm=None): return 2r*x + 5r
sage: def conjugate_f(self, x): return x/2r
sage: nf = unpickle_function('g', 1, None, None, True, [None, pickle_
↪wrapper(evalf_f), pickle_wrapper(conjugate_f)] + [None]*8)
sage: nf
g
sage: nf(2)
g(2)
sage: nf(2).n()
9.00000000000000
sage: nf(2).conjugate()
1
```

```
>>> from sage.all import *
>>> from sage.symbolic.function_factory import unpickle_function
>>> nf = unpickle_function('f', Integer(2), '\\foo', {'mathematica':'Foo'}, True,␣
↪[])
>>> nf
f
>>> nf(Integer(1),Integer(2))
f(1, 2)
>>> latex(nf(x,x))
\foo\left(x, x\right)
>>> nf._mathematica_init_()
'Foo'

>>> from sage.symbolic.function import pickle_wrapper
>>> def evalf_f(self, x, parent=None, algorithm=None): return 2*x + 5
>>> def conjugate_f(self, x): return x/2
>>> nf = unpickle_function('g', Integer(1), None, None, True, [None, pickle_
↪wrapper(evalf_f), pickle_wrapper(conjugate_f)] + [None]*Integer(8))
>>> nf
g
>>> nf(Integer(2))
g(2)
>>> nf(Integer(2)).n()
9.00000000000000
>>> nf(Integer(2)).conjugate()
1
```

2.12 Functional notation support for common calculus methods

EXAMPLES: We illustrate each of the calculus functional functions.

```
sage: simplify(x - x)
0
sage: a = var('a')
sage: derivative(x^a + sin(x), x)
a*x^(a - 1) + cos(x)
sage: diff(x^a + sin(x), x)
a*x^(a - 1) + cos(x)
```

(continues on next page)

(continued from previous page)

```
sage: derivative(x^a + sin(x), x)
a*x^(a - 1) + cos(x)
sage: integral(a*x*sin(x), x)
-(x*cos(x) - sin(x))*a
sage: integrate(a*x*sin(x), x)
-(x*cos(x) - sin(x))*a
sage: limit(a*sin(x)/x, x=0)
a
sage: taylor(a*sin(x)/x, x, 0, 4)
1/120*a*x^4 - 1/6*a*x^2 + a
sage: expand((x - a)^3)
-a^3 + 3*a^2*x - 3*a*x^2 + x^3
```

```
>>> from sage.all import *
>>> simplify(x - x)
0
>>> a = var('a')
>>> derivative(x**a + sin(x), x)
a*x^(a - 1) + cos(x)
>>> diff(x**a + sin(x), x)
a*x^(a - 1) + cos(x)
>>> derivative(x**a + sin(x), x)
a*x^(a - 1) + cos(x)
>>> integral(a*x*sin(x), x)
-(x*cos(x) - sin(x))*a
>>> integrate(a*x*sin(x), x)
-(x*cos(x) - sin(x))*a
>>> limit(a*sin(x)/x, x=Integer(0))
a
>>> taylor(a*sin(x)/x, x, Integer(0), Integer(4))
1/120*a*x^4 - 1/6*a*x^2 + a
>>> expand((x - a)**Integer(3))
-a^3 + 3*a^2*x - 3*a*x^2 + x^3
```

sage.calculus.functional.**derivative**(*f*, **args*, ***kwds*)

The derivative of f.

Repeated differentiation is supported by the syntax given in the examples below.

ALIAS: diff

EXAMPLES: We differentiate a callable symbolic function:

```
sage: f(x,y) = x*y + sin(x^2) + e^(-x)
sage: f
(x, y) |--> x*y + e^(-x) + sin(x^2)
sage: derivative(f, x)
(x, y) |--> 2*x*cos(x^2) + y - e^(-x)
sage: derivative(f, y)
(x, y) |--> x
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(x*y + sin(x**Integer(2)) + e**(-
```

(continues on next page)

(continued from previous page)

```
↪x)).function(x,y)
>>> f
(x, y) |--> x*y + e^(-x) + sin(x^2)
>>> derivative(f, x)
(x, y) |--> 2*x*cos(x^2) + y - e^(-x)
>>> derivative(f, y)
(x, y) |--> x
```

We differentiate a polynomial:

```
sage: t = polygen(QQ, 't')
sage: f = (1-t)^5; f
-t^5 + 5*t^4 - 10*t^3 + 10*t^2 - 5*t + 1
sage: derivative(f)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
sage: derivative(f, t)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
sage: derivative(f, t, t)
-20*t^3 + 60*t^2 - 60*t + 20
sage: derivative(f, t, 2)
-20*t^3 + 60*t^2 - 60*t + 20
sage: derivative(f, 2)
-20*t^3 + 60*t^2 - 60*t + 20
```

```
>>> from sage.all import *
>>> t = polygen(QQ, 't')
>>> f = (Integer(1)-t)**Integer(5); f
-t^5 + 5*t^4 - 10*t^3 + 10*t^2 - 5*t + 1
>>> derivative(f)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
>>> derivative(f, t)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
>>> derivative(f, t, t)
-20*t^3 + 60*t^2 - 60*t + 20
>>> derivative(f, t, Integer(2))
-20*t^3 + 60*t^2 - 60*t + 20
>>> derivative(f, Integer(2))
-20*t^3 + 60*t^2 - 60*t + 20
```

We differentiate a symbolic expression:

```
sage: var('a x')
(a, x)
sage: f = exp(sin(a - x^2))/x
sage: derivative(f, x)
-2*cos(-x^2 + a)*e^(sin(-x^2 + a)) - e^(sin(-x^2 + a))/x^2
sage: derivative(f, a)
cos(-x^2 + a)*e^(sin(-x^2 + a))/x
```

```
>>> from sage.all import *
>>> var('a x')
(a, x)
```

(continues on next page)

(continued from previous page)

```
>>> f = exp(sin(a - x**Integer(2)))/x
>>> derivative(f, x)
-2*cos(-x^2 + a)*e^(sin(-x^2 + a)) - e^(sin(-x^2 + a))/x^2
>>> derivative(f, a)
cos(-x^2 + a)*e^(sin(-x^2 + a))/x
```

Syntax for repeated differentiation:

```
sage: R.<u, v> = PolynomialRing(QQ)
sage: f = u^4*v^5
sage: derivative(f, u)
4*u^3*v^5
sage: f.derivative(u)   # can always use method notation too
4*u^3*v^5
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('u', 'v',)); (u, v,) = R._first_ngens(2)
>>> f = u**Integer(4)*v**Integer(5)
>>> derivative(f, u)
4*u^3*v^5
>>> f.derivative(u)   # can always use method notation too
4*u^3*v^5
```

```
sage: derivative(f, u, u)
12*u^2*v^5
sage: derivative(f, u, u, u)
24*u*v^5
sage: derivative(f, u, 3)
24*u*v^5
```

```
>>> from sage.all import *
>>> derivative(f, u, u)
12*u^2*v^5
>>> derivative(f, u, u, u)
24*u*v^5
>>> derivative(f, u, Integer(3))
24*u*v^5
```

```
sage: derivative(f, u, v)
20*u^3*v^4
sage: derivative(f, u, 2, v)
60*u^2*v^4
sage: derivative(f, u, v, 2)
80*u^3*v^3
sage: derivative(f, [u, v, v])
80*u^3*v^3
```

```
>>> from sage.all import *
>>> derivative(f, u, v)
20*u^3*v^4
>>> derivative(f, u, Integer(2), v)
```

(continues on next page)

(continued from previous page)

```
60*u^2*v^4
>>> derivative(f, u, v, Integer(2))
80*u^3*v^3
>>> derivative(f, [u, v, v])
80*u^3*v^3
```

We differentiate a scalar field on a manifold:

```
sage: M = Manifold(2, 'M')
sage: X.<x,y> = M.chart()
sage: f = M.scalar_field(x^2*y, name='f')
sage: derivative(f)
1-form df on the 2-dimensional differentiable manifold M
sage: derivative(f).display()
df = 2*x*y dx + x^2 dy
```

```
>>> from sage.all import *
>>> M = Manifold(Integer(2), 'M')
>>> X = M.chart(names=('x', 'y',)); (x, y,) = X._first_ngens(2)
>>> f = M.scalar_field(x**Integer(2)*y, name='f')
>>> derivative(f)
1-form df on the 2-dimensional differentiable manifold M
>>> derivative(f).display()
df = 2*x*y dx + x^2 dy
```

We differentiate a differentiable form, getting its exterior derivative:

```
sage: a = M.one_form(-y, x, name='a'); a.display()
a = -y dx + x dy
sage: derivative(a)
2-form da on the 2-dimensional differentiable manifold M
sage: derivative(a).display()
da = 2 dx∧dy
```

```
>>> from sage.all import *
>>> a = M.one_form(-y, x, name='a'); a.display()
a = -y dx + x dy
>>> derivative(a)
2-form da on the 2-dimensional differentiable manifold M
>>> derivative(a).display()
da = 2 dx∧dy
```

sage.calculus.functional.**diff**(*f*, **args*, ***kwds*)

The derivative of f.

Repeated differentiation is supported by the syntax given in the examples below.

ALIAS: diff

EXAMPLES: We differentiate a callable symbolic function:

```
sage: f(x,y) = x*y + sin(x^2) + e^(-x)
sage: f
```

(continues on next page)

(continued from previous page)

```
(x, y) |--> x*y + e^(-x) + sin(x^2)
sage: derivative(f, x)
(x, y) |--> 2*x*cos(x^2) + y - e^(-x)
sage: derivative(f, y)
(x, y) |--> x
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(x*y + sin(x**Integer(2)) + e**(-
↪x)).function(x,y)
>>> f
(x, y) |--> x*y + e^(-x) + sin(x^2)
>>> derivative(f, x)
(x, y) |--> 2*x*cos(x^2) + y - e^(-x)
>>> derivative(f, y)
(x, y) |--> x
```

We differentiate a polynomial:

```
sage: t = polygen(QQ, 't')
sage: f = (1-t)^5; f
-t^5 + 5*t^4 - 10*t^3 + 10*t^2 - 5*t + 1
sage: derivative(f)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
sage: derivative(f, t)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
sage: derivative(f, t, t)
-20*t^3 + 60*t^2 - 60*t + 20
sage: derivative(f, t, 2)
-20*t^3 + 60*t^2 - 60*t + 20
sage: derivative(f, 2)
-20*t^3 + 60*t^2 - 60*t + 20
```

```
>>> from sage.all import *
>>> t = polygen(QQ, 't')
>>> f = (Integer(1)-t)**Integer(5); f
-t^5 + 5*t^4 - 10*t^3 + 10*t^2 - 5*t + 1
>>> derivative(f)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
>>> derivative(f, t)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
>>> derivative(f, t, t)
-20*t^3 + 60*t^2 - 60*t + 20
>>> derivative(f, t, Integer(2))
-20*t^3 + 60*t^2 - 60*t + 20
>>> derivative(f, Integer(2))
-20*t^3 + 60*t^2 - 60*t + 20
```

We differentiate a symbolic expression:

```
sage: var('a x')
(a, x)
sage: f = exp(sin(a - x^2))/x
```

(continues on next page)

(continued from previous page)

```
sage: derivative(f, x)
-2*cos(-x^2 + a)*e^(sin(-x^2 + a)) - e^(sin(-x^2 + a))/x^2
sage: derivative(f, a)
cos(-x^2 + a)*e^(sin(-x^2 + a))/x
```

```
>>> from sage.all import *
>>> var('a x')
(a, x)
>>> f = exp(sin(a - x**Integer(2)))/x
>>> derivative(f, x)
-2*cos(-x^2 + a)*e^(sin(-x^2 + a)) - e^(sin(-x^2 + a))/x^2
>>> derivative(f, a)
cos(-x^2 + a)*e^(sin(-x^2 + a))/x
```

Syntax for repeated differentiation:

```
sage: R.<u, v> = PolynomialRing(QQ)
sage: f = u^4*v^5
sage: derivative(f, u)
4*u^3*v^5
sage: f.derivative(u)   # can always use method notation too
4*u^3*v^5
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, names=('u', 'v',)); (u, v,) = R._first_ngens(2)
>>> f = u**Integer(4)*v**Integer(5)
>>> derivative(f, u)
4*u^3*v^5
>>> f.derivative(u)   # can always use method notation too
4*u^3*v^5
```

```
sage: derivative(f, u, u)
12*u^2*v^5
sage: derivative(f, u, u, u)
24*u*v^5
sage: derivative(f, u, 3)
24*u*v^5
```

```
>>> from sage.all import *
>>> derivative(f, u, u)
12*u^2*v^5
>>> derivative(f, u, u, u)
24*u*v^5
>>> derivative(f, u, Integer(3))
24*u*v^5
```

```
sage: derivative(f, u, v)
20*u^3*v^4
sage: derivative(f, u, 2, v)
60*u^2*v^4
sage: derivative(f, u, v, 2)
```

(continues on next page)

(continued from previous page)

```
80*u^3*v^3
sage: derivative(f, [u, v, v])
80*u^3*v^3
```

```
>>> from sage.all import *
>>> derivative(f, u, v)
20*u^3*v^4
>>> derivative(f, u, Integer(2), v)
60*u^2*v^4
>>> derivative(f, u, v, Integer(2))
80*u^3*v^3
>>> derivative(f, [u, v, v])
80*u^3*v^3
```

We differentiate a scalar field on a manifold:

```
sage: M = Manifold(2, 'M')
sage: X.<x,y> = M.chart()
sage: f = M.scalar_field(x^2*y, name='f')
sage: derivative(f)
1-form df on the 2-dimensional differentiable manifold M
sage: derivative(f).display()
df = 2*x*y dx + x^2 dy
```

```
>>> from sage.all import *
>>> M = Manifold(Integer(2), 'M')
>>> X = M.chart(names=('x', 'y',)); (x, y,) = X._first_ngens(2)
>>> f = M.scalar_field(x**Integer(2)*y, name='f')
>>> derivative(f)
1-form df on the 2-dimensional differentiable manifold M
>>> derivative(f).display()
df = 2*x*y dx + x^2 dy
```

We differentiate a differentiable form, getting its exterior derivative:

```
sage: a = M.one_form(-y, x, name='a'); a.display()
a = -y dx + x dy
sage: derivative(a)
2-form da on the 2-dimensional differentiable manifold M
sage: derivative(a).display()
da = 2 dx∧dy
```

```
>>> from sage.all import *
>>> a = M.one_form(-y, x, name='a'); a.display()
a = -y dx + x dy
>>> derivative(a)
2-form da on the 2-dimensional differentiable manifold M
>>> derivative(a).display()
da = 2 dx∧dy
```

`sage.calculus.functional.`**`expand`**(*x*, **args*, ***kwds*)

EXAMPLES:

```
sage: a = (x-1)*(x^2 - 1); a
(x^2 - 1)*(x - 1)
sage: expand(a)
x^3 - x^2 - x + 1
```

```
>>> from sage.all import *
>>> a = (x-Integer(1))*(x**Integer(2) - Integer(1)); a
(x^2 - 1)*(x - 1)
>>> expand(a)
x^3 - x^2 - x + 1
```

You can also use expand on polynomial, integer, and other factorizations:

```
sage: x = polygen(ZZ)
sage: F = factor(x^12 - 1); F
(x - 1) * (x + 1) * (x^2 - x + 1) * (x^2 + 1) * (x^2 + x + 1) * (x^4 - x^2 + 1)
sage: expand(F)
x^12 - 1
sage: F.expand()
x^12 - 1
sage: F = factor(2007); F
3^2 * 223
sage: expand(F)
2007
```

```
>>> from sage.all import *
>>> x = polygen(ZZ)
>>> F = factor(x**Integer(12) - Integer(1)); F
(x - 1) * (x + 1) * (x^2 - x + 1) * (x^2 + 1) * (x^2 + x + 1) * (x^4 - x^2 + 1)
>>> expand(F)
x^12 - 1
>>> F.expand()
x^12 - 1
>>> F = factor(Integer(2007)); F
3^2 * 223
>>> expand(F)
2007
```

Note: If you want to compute the expanded form of a polynomial arithmetic operation quickly and the coefficients of the polynomial all lie in some ring, e.g., the integers, it is vastly faster to create a polynomial ring and do the arithmetic there.

```
sage: x = polygen(ZZ)      # polynomial over a given base ring.
sage: f = sum(x^n for n in range(5))
sage: f*f                  # much faster, even if the degree is huge
x^8 + 2*x^7 + 3*x^6 + 4*x^5 + 5*x^4 + 4*x^3 + 3*x^2 + 2*x + 1
```

```
>>> from sage.all import *
>>> x = polygen(ZZ)      # polynomial over a given base ring.
>>> f = sum(x**n for n in range(Integer(5)))
>>> f*f                  # much faster, even if the degree is huge
x^8 + 2*x^7 + 3*x^6 + 4*x^5 + 5*x^4 + 4*x^3 + 3*x^2 + 2*x + 1
```

sage.calculus.functional.**integral**(*f*, **args*, ***kwds*)

The integral of f.

EXAMPLES:

```
sage: integral(sin(x), x)
-cos(x)
sage: integral(sin(x)^2, x, pi, 123*pi/2)
121/4*pi
sage: integral( sin(x), x, 0, pi)
2
```

```
>>> from sage.all import *
>>> integral(sin(x), x)
-cos(x)
>>> integral(sin(x)**Integer(2), x, pi, Integer(123)*pi/Integer(2))
121/4*pi
>>> integral( sin(x), x, Integer(0), pi)
2
```

We integrate a symbolic function:

```
sage: f(x,y,z) = x*y/z + sin(z)
sage: integral(f, z)
(x, y, z) |--> x*y*log(z) - cos(z)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y,z"); f = symbolic_expression(x*y/z + sin(z)).function(x,y,z)
>>> integral(f, z)
(x, y, z) |--> x*y*log(z) - cos(z)
```

```
sage: var('a,b')
(a, b)
sage: assume(b-a>0)
sage: integral( sin(x), x, a, b)
cos(a) - cos(b)
sage: forget()
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> assume(b-a>Integer(0))
>>> integral( sin(x), x, a, b)
cos(a) - cos(b)
>>> forget()
```

```
sage: integral(x/(x^3-1), x)
1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x + 1)) - 1/6*log(x^2 + x + 1) + 1/3*log(x - 1)
```

```
>>> from sage.all import *
>>> integral(x/(x**Integer(3)-Integer(1)), x)
1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x + 1)) - 1/6*log(x^2 + x + 1) + 1/3*log(x - 1)
```

```
sage: integral( exp(-x^2), x )
1/2*sqrt(pi)*erf(x)
```

```
>>> from sage.all import *
>>> integral( exp(-x**Integer(2)), x )
1/2*sqrt(pi)*erf(x)
```

We define the Gaussian, plot and integrate it numerically and symbolically:

```
sage: f(x) = 1/(sqrt(2*pi)) * e^(-x^2/2)
sage: P = plot(f, -4, 4, hue=0.8, thickness=2)                                   #␣
↪needs sage.plot
sage: P.show(ymin=0, ymax=0.4)                                                   #␣
↪needs sage.plot
sage: numerical_integral(f, -4, 4)                    # random output
(0.99993665751633376, 1.1101527003413533e-14)
sage: integrate(f, x)
x |--> 1/2*erf(1/2*sqrt(2)*x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(Integer(1)/(sqrt(Integer(2)*pi)) *␣
↪e**(-x**Integer(2)/Integer(2))).function(x)
>>> P = plot(f, -Integer(4), Integer(4), hue=RealNumber('0.8'),␣
↪thickness=Integer(2))                                # needs sage.plot
>>> P.show(ymin=Integer(0), ymax=RealNumber('0.4'))                              ␣
↪                     # needs sage.plot
>>> numerical_integral(f, -Integer(4), Integer(4))                    # random␣
↪output
(0.99993665751633376, 1.1101527003413533e-14)
>>> integrate(f, x)
x |--> 1/2*erf(1/2*sqrt(2)*x)
```

You can have Sage calculate multiple integrals. For example, consider the function $exp(y^2)$ on the region between the lines $x = y$, $x = 1$, and $y = 0$. We find the value of the integral on this region using the command:

```
sage: area = integral(integral(exp(y^2),x,0,y),y,0,1); area
1/2*e - 1/2
sage: float(area)
0.859140914229522...
```

```
>>> from sage.all import *
>>> area = integral(integral(exp(y**Integer(2)),x,Integer(0),y),y,Integer(0),
↪Integer(1)); area
1/2*e - 1/2
>>> float(area)
0.859140914229522...
```

We compute the line integral of $\sin(x)$ along the arc of the curve $x = y^4$ from $(1, -1)$ to $(1, 1)$:

```
sage: t = var('t')
sage: (x,y) = (t^4,t)
sage: (dx,dy) = (diff(x,t), diff(y,t))
```

(continues on next page)

(continued from previous page)

```
sage: integral(sin(x)*dx, t,-1, 1)
0
sage: restore('x,y')   # restore the symbolic variables x and y
```

```
>>> from sage.all import *
>>> t = var('t')
>>> (x,y) = (t**Integer(4),t)
>>> (dx,dy) = (diff(x,t), diff(y,t))
>>> integral(sin(x)*dx, t,-Integer(1), Integer(1))
0
>>> restore('x,y')   # restore the symbolic variables x and y
```

Sage is now (Issue #27958) able to compute the following integral:

```
sage: integral(exp(-x^2)*log(x), x)  # long time
1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)
```

```
>>> from sage.all import *
>>> integral(exp(-x**Integer(2))*log(x), x)  # long time
1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)
```

and its value:

```
sage: integral( exp(-x^2)*ln(x), x, 0, oo)
-1/4*sqrt(pi)*(euler_gamma + 2*log(2))
```

```
>>> from sage.all import *
>>> integral( exp(-x**Integer(2))*ln(x), x, Integer(0), oo)
-1/4*sqrt(pi)*(euler_gamma + 2*log(2))
```

This definite integral is easy:

```
sage: integral( ln(x)/x, x, 1, 2)
1/2*log(2)^2
```

```
>>> from sage.all import *
>>> integral( ln(x)/x, x, Integer(1), Integer(2))
1/2*log(2)^2
```

Sage cannot do this elliptic integral (yet):

```
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
integrate(1/(sqrt(2*t^2 + 1)*sqrt(t^2 - 2)), t, 2, 3)
```

```
>>> from sage.all import *
>>> integral(Integer(1)/sqrt(Integer(2)*t**Integer(4) - Integer(3)*t**Integer(2) -
↪ Integer(2)), t, Integer(2), Integer(3))
integrate(1/(sqrt(2*t^2 + 1)*sqrt(t^2 - 2)), t, 2, 3)
```

A double integral:

```
sage: y = var('y')
sage: integral(integral(x*y^2, x, 0, y), y, -2, 2)
32/5
```

```
>>> from sage.all import *
>>> y = var('y')
>>> integral(integral(x*y**Integer(2), x, Integer(0), y), y, -Integer(2),␣
→Integer(2))
32/5
```

This illustrates using assumptions:

```
sage: integral(abs(x), x, 0, 5)
25/2
sage: a = var("a")
sage: integral(abs(x), x, 0, a)
1/2*a*abs(a)
sage: integral(abs(x)*x, x, 0, a)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)',
see `assume?` for more details)
Is a positive, negative or zero?
sage: assume(a>0)
sage: integral(abs(x)*x, x, 0, a)
1/3*a^3
sage: forget()      # forget the assumptions.
```

```
>>> from sage.all import *
>>> integral(abs(x), x, Integer(0), Integer(5))
25/2
>>> a = var("a")
>>> integral(abs(x), x, Integer(0), a)
1/2*a*abs(a)
>>> integral(abs(x)*x, x, Integer(0), a)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)',
see `assume?` for more details)
Is a positive, negative or zero?
>>> assume(a>Integer(0))
>>> integral(abs(x)*x, x, Integer(0), a)
1/3*a^3
>>> forget()      # forget the assumptions.
```

We integrate and differentiate a huge mess:

```
sage: f = (x^2-1+3*(1+x^2)^(1/3))/(1+x^2)^(2/3)*x/(x^2+2)^2
sage: g = integral(f, x)
```

(continues on next page)

(continued from previous page)

```
sage: h = f - diff(g, x)
```

```
>>> from sage.all import *
>>> f = (x**Integer(2)-
↪Integer(1)+Integer(3)*(Integer(1)+x**Integer(2))**(Integer(1)/Integer(3)))/
↪(Integer(1)+x**Integer(2))**(Integer(2)/Integer(3))*x/
↪(x**Integer(2)+Integer(2))**Integer(2)
>>> g = integral(f, x)
>>> h = f - diff(g, x)
```

```
sage: [float(h(x=i)) for i in range(5)] #random

[0.0,
 -1.1102230246251565e-16,
 -5.551115123125783e-17,
 -5.551115123125783e-17,
 -6.938893903907228e-17]
sage: h.factor()
0
sage: bool(h == 0)
True
```

```
>>> from sage.all import *
>>> [float(h(x=i)) for i in range(Integer(5))] #random

[0.0,
 -1.1102230246251565e-16,
 -5.551115123125783e-17,
 -5.551115123125783e-17,
 -6.938893903907228e-17]
>>> h.factor()
0
>>> bool(h == Integer(0))
True
```

`sage.calculus.functional.`**`integrate`**(f, **args*, ***kwds*)

The integral of f.

EXAMPLES:

```
sage: integral(sin(x), x)
-cos(x)
sage: integral(sin(x)^2, x, pi, 123*pi/2)
121/4*pi
sage: integral( sin(x), x, 0, pi)
2
```

```
>>> from sage.all import *
>>> integral(sin(x), x)
-cos(x)
>>> integral(sin(x)**Integer(2), x, pi, Integer(123)*pi/Integer(2))
```

(continues on next page)

(continued from previous page)

```
121/4*pi
>>> integral( sin(x), x, Integer(0), pi)
2
```

We integrate a symbolic function:

```
sage: f(x,y,z) = x*y/z + sin(z)
sage: integral(f, z)
(x, y, z) |--> x*y*log(z) - cos(z)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y,z"); f = symbolic_expression(x*y/z + sin(z)).function(x,y,z)
>>> integral(f, z)
(x, y, z) |--> x*y*log(z) - cos(z)
```

```
sage: var('a,b')
(a, b)
sage: assume(b-a>0)
sage: integral( sin(x), x, a, b)
cos(a) - cos(b)
sage: forget()
```

```
>>> from sage.all import *
>>> var('a,b')
(a, b)
>>> assume(b-a>Integer(0))
>>> integral( sin(x), x, a, b)
cos(a) - cos(b)
>>> forget()
```

```
sage: integral(x/(x^3-1), x)
1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x + 1)) - 1/6*log(x^2 + x + 1) + 1/3*log(x - 1)
```

```
>>> from sage.all import *
>>> integral(x/(x**Integer(3)-Integer(1)), x)
1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x + 1)) - 1/6*log(x^2 + x + 1) + 1/3*log(x - 1)
```

```
sage: integral( exp(-x^2), x )
1/2*sqrt(pi)*erf(x)
```

```
>>> from sage.all import *
>>> integral( exp(-x**Integer(2)), x )
1/2*sqrt(pi)*erf(x)
```

We define the Gaussian, plot and integrate it numerically and symbolically:

```
sage: f(x) = 1/(sqrt(2*pi)) * e^(-x^2/2)
sage: P = plot(f, -4, 4, hue=0.8, thickness=2)                               #␣
↪needs sage.plot
sage: P.show(ymin=0, ymax=0.4)                                               #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
sage: numerical_integral(f, -4, 4)                    # random output
(0.99993665751633376, 1.1101527003413533e-14)
sage: integrate(f, x)
x |--> 1/2*erf(1/2*sqrt(2)*x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(Integer(1)/(sqrt(Integer(2)*pi)) *␣
↪e**(-x**Integer(2)/Integer(2))).function(x)
>>> P = plot(f, -Integer(4), Integer(4), hue=RealNumber('0.8'),␣
↪thickness=Integer(2))                                 # needs sage.plot
>>> P.show(ymin=Integer(0), ymax=RealNumber('0.4'))                          ␣
↪                   # needs sage.plot
>>> numerical_integral(f, -Integer(4), Integer(4))                # random␣
↪output
(0.99993665751633376, 1.1101527003413533e-14)
>>> integrate(f, x)
x |--> 1/2*erf(1/2*sqrt(2)*x)
```

You can have Sage calculate multiple integrals. For example, consider the function $exp(y^2)$ on the region between the lines $x = y$, $x = 1$, and $y = 0$. We find the value of the integral on this region using the command:

```
sage: area = integral(integral(exp(y^2),x,0,y),y,0,1); area
1/2*e - 1/2
sage: float(area)
0.859140914229522...
```

```
>>> from sage.all import *
>>> area = integral(integral(exp(y**Integer(2)),x,Integer(0),y),y,Integer(0),
↪Integer(1)); area
1/2*e - 1/2
>>> float(area)
0.859140914229522...
```

We compute the line integral of $\sin(x)$ along the arc of the curve $x = y^4$ from $(1, -1)$ to $(1, 1)$:

```
sage: t = var('t')
sage: (x,y) = (t^4,t)
sage: (dx,dy) = (diff(x,t), diff(y,t))
sage: integral(sin(x)*dx, t,-1, 1)
0
sage: restore('x,y')   # restore the symbolic variables x and y
```

```
>>> from sage.all import *
>>> t = var('t')
>>> (x,y) = (t**Integer(4),t)
>>> (dx,dy) = (diff(x,t), diff(y,t))
>>> integral(sin(x)*dx, t,-Integer(1), Integer(1))
0
>>> restore('x,y')   # restore the symbolic variables x and y
```

Sage is now (Issue #27958) able to compute the following integral:

```
sage: integral(exp(-x^2)*log(x), x)   # long time
1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)
```

```
>>> from sage.all import *
>>> integral(exp(-x**Integer(2))*log(x), x)   # long time
1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)
```

and its value:

```
sage: integral( exp(-x^2)*ln(x), x, 0, oo)
-1/4*sqrt(pi)*(euler_gamma + 2*log(2))
```

```
>>> from sage.all import *
>>> integral( exp(-x**Integer(2))*ln(x), x, Integer(0), oo)
-1/4*sqrt(pi)*(euler_gamma + 2*log(2))
```

This definite integral is easy:

```
sage: integral( ln(x)/x, x, 1, 2)
1/2*log(2)^2
```

```
>>> from sage.all import *
>>> integral( ln(x)/x, x, Integer(1), Integer(2))
1/2*log(2)^2
```

Sage cannot do this elliptic integral (yet):

```
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
integrate(1/(sqrt(2*t^2 + 1)*sqrt(t^2 - 2)), t, 2, 3)
```

```
>>> from sage.all import *
>>> integral(Integer(1)/sqrt(Integer(2)*t**Integer(4) - Integer(3)*t**Integer(2) -
↪ Integer(2)), t, Integer(2), Integer(3))
integrate(1/(sqrt(2*t^2 + 1)*sqrt(t^2 - 2)), t, 2, 3)
```

A double integral:

```
sage: y = var('y')
sage: integral(integral(x*y^2, x, 0, y), y, -2, 2)
32/5
```

```
>>> from sage.all import *
>>> y = var('y')
>>> integral(integral(x*y**Integer(2), x, Integer(0), y), y, -Integer(2),␣
↪Integer(2))
32/5
```

This illustrates using assumptions:

```
sage: integral(abs(x), x, 0, 5)
25/2
sage: a = var("a")
```

(continues on next page)

(continued from previous page)

```
sage: integral(abs(x), x, 0, a)
1/2*a*abs(a)
sage: integral(abs(x)*x, x, 0, a)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)',
see `assume?` for more details)
Is a positive, negative or zero?
sage: assume(a>0)
sage: integral(abs(x)*x, x, 0, a)
1/3*a^3
sage: forget()      # forget the assumptions.
```

```
>>> from sage.all import *
>>> integral(abs(x), x, Integer(0), Integer(5))
25/2
>>> a = var("a")
>>> integral(abs(x), x, Integer(0), a)
1/2*a*abs(a)
>>> integral(abs(x)*x, x, Integer(0), a)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)',
see `assume?` for more details)
Is a positive, negative or zero?
>>> assume(a>Integer(0))
>>> integral(abs(x)*x, x, Integer(0), a)
1/3*a^3
>>> forget()      # forget the assumptions.
```

We integrate and differentiate a huge mess:

```
sage: f = (x^2-1+3*(1+x^2)^(1/3))/(1+x^2)^(2/3)*x/(x^2+2)^2
sage: g = integral(f, x)
sage: h = f - diff(g, x)
```

```
>>> from sage.all import *
>>> f = (x**Integer(2)-
↪Integer(1)+Integer(3)*(Integer(1)+x**Integer(2))**(Integer(1)/Integer(3)))/
↪(Integer(1)+x**Integer(2))**(Integer(2)/Integer(3))*x/
↪(x**Integer(2)+Integer(2))**Integer(2)
>>> g = integral(f, x)
>>> h = f - diff(g, x)
```

```
sage: [float(h(x=i)) for i in range(5)] #random

[0.0,
```

(continues on next page)

(continued from previous page)

```
 -1.1102230246251565e-16,
 -5.551115123125783e-17,
 -5.551115123125783e-17,
 -6.938893903907228e-17]
sage: h.factor()
0
sage: bool(h == 0)
True
```

```
>>> from sage.all import *
>>> [float(h(x=i)) for i in range(Integer(5))] #random

[0.0,
 -1.1102230246251565e-16,
 -5.551115123125783e-17,
 -5.551115123125783e-17,
 -6.938893903907228e-17]
>>> h.factor()
0
>>> bool(h == Integer(0))
True
```

`sage.calculus.functional.lim(f, dir=None, taylor=False, **argv)`

Return the limit as the variable v approaches a from the given direction.

```
limit(expr, x = a)
limit(expr, x = a, dir='above')
```

INPUT:

- `dir` – (default: `None`) dir may have the value `'plus'` (or `'above'`) for a limit from above, `'minus'` (or `'below'`) for a limit from below, or may be omitted (implying a two-sided limit is to be computed).
- `taylor` – (default: `False`) if `True`, use Taylor series, which allows more limits to be computed (but may also crash in some obscure cases due to bugs in Maxima).
- `\*\*argv` – 1 named parameter

ALIAS: You can also use lim instead of limit.

EXAMPLES:

```
sage: limit(sin(x)/x, x=0)
1
sage: limit(exp(x), x=oo)
+Infinity
sage: lim(exp(x), x=-oo)
0
sage: lim(1/x, x=0)
Infinity
sage: limit(sqrt(x^2+x+1)+x, taylor=True, x=-oo)
-1/2
sage: limit((tan(sin(x)) - sin(tan(x)))/x^7, taylor=True, x=0)
1/30
```

```
>>> from sage.all import *
>>> limit(sin(x)/x, x=Integer(0))
1
>>> limit(exp(x), x=oo)
+Infinity
>>> lim(exp(x), x=-oo)
0
>>> lim(Integer(1)/x, x=Integer(0))
Infinity
>>> limit(sqrt(x**Integer(2)+x+Integer(1))+x, taylor=True, x=-oo)
-1/2
>>> limit((tan(sin(x)) - sin(tan(x)))/x**Integer(7), taylor=True, x=Integer(0))
1/30
```

Sage does not know how to do this limit (which is 0), so it returns it unevaluated:

```
sage: lim(exp(x^2)*(1-erf(x)), x=infinity)
-limit((erf(x) - 1)*e^(x^2), x, +Infinity)
```

```
>>> from sage.all import *
>>> lim(exp(x**Integer(2))*(Integer(1)-erf(x)), x=infinity)
-limit((erf(x) - 1)*e^(x^2), x, +Infinity)
```

sage.calculus.functional.**limit**(*f*, *dir=None*, *taylor=False*, ***argv*)

Return the limit as the variable v approaches a from the given direction.

```
limit(expr, x = a)
limit(expr, x = a, dir='above')
```

INPUT:

- `dir` – (default: `None`) dir may have the value `'plus'` (or `'above'`) for a limit from above, `'minus'` (or `'below'`) for a limit from below, or may be omitted (implying a two-sided limit is to be computed).
- `taylor` – (default: `False`) if `True`, use Taylor series, which allows more limits to be computed (but may also crash in some obscure cases due to bugs in Maxima).
- `\*\*argv` – 1 named parameter

ALIAS: You can also use lim instead of limit.

EXAMPLES:

```
sage: limit(sin(x)/x, x=0)
1
sage: limit(exp(x), x=oo)
+Infinity
sage: lim(exp(x), x=-oo)
0
sage: lim(1/x, x=0)
Infinity
sage: limit(sqrt(x^2+x+1)+x, taylor=True, x=-oo)
-1/2
sage: limit((tan(sin(x)) - sin(tan(x)))/x^7, taylor=True, x=0)
1/30
```

```
>>> from sage.all import *
>>> limit(sin(x)/x, x=Integer(0))
1
>>> limit(exp(x), x=oo)
+Infinity
>>> lim(exp(x), x=-oo)
0
>>> lim(Integer(1)/x, x=Integer(0))
Infinity
>>> limit(sqrt(x**Integer(2)+x+Integer(1))+x, taylor=True, x=-oo)
-1/2
>>> limit((tan(sin(x)) - sin(tan(x)))/x**Integer(7), taylor=True, x=Integer(0))
1/30
```

Sage does not know how to do this limit (which is 0), so it returns it unevaluated:

```
sage: lim(exp(x^2)*(1-erf(x)), x=infinity)
-limit((erf(x) - 1)*e^(x^2), x, +Infinity)
```

```
>>> from sage.all import *
>>> lim(exp(x**Integer(2))*(Integer(1)-erf(x)), x=infinity)
-limit((erf(x) - 1)*e^(x^2), x, +Infinity)
```

sage.calculus.functional.**simplify**(*f*, *algorithm='maxima'*, ***kwds*)

Simplify the expression f.

See the documentation of the `simplify()` method of symbolic expressions for details on options.

EXAMPLES:

We simplify the expression $i + x - x$:

```
sage: f = I + x - x; simplify(f)
I
```

```
>>> from sage.all import *
>>> f = I + x - x; simplify(f)
I
```

In fact, printing f yields the same thing - i.e., the simplified form.

Some simplifications are algorithm-specific:

```
sage: x, t = var("x, t")
sage: ex = 1/2*I*x + 1/2*I*sqrt(x^2 - 1) + 1/2/(I*x + I*sqrt(x^2 - 1))
sage: simplify(ex)
1/2*I*x + 1/2*I*sqrt(x^2 - 1) + 1/(2*I*x + 2*I*sqrt(x^2 - 1))
sage: simplify(ex, algorithm='giac')   # needs giac
I*sqrt(x^2 - 1)
```

```
>>> from sage.all import *
>>> x, t = var("x, t")
>>> ex = Integer(1)/Integer(2)*I*x + Integer(1)/Integer(2)*I*sqrt(x**Integer(2) -
↪Integer(1)) + Integer(1)/Integer(2)/(I*x + I*sqrt(x**Integer(2) - Integer(1)))
```

(continues on next page)

(continued from previous page)

```
>>> simplify(ex)
1/2*I*x + 1/2*I*sqrt(x^2 - 1) + 1/(2*I*x + 2*I*sqrt(x^2 - 1))
>>> simplify(ex, algorithm='giac')  # needs giac
I*sqrt(x^2 - 1)
```

sage.calculus.functional.**taylor**(*f*, **args*)

Expand `self` in a truncated Taylor or Laurent series in the variable v around the point a, containing terms through $(x-a)^n$. Functions in more variables are also supported.

INPUT:

- `*args` – the following notation is supported
- `x`, `a`, `n` – variable, point, degree
- `(x, a)`, `(y, b)`, …, `n` – variables with points, degree of polynomial

EXAMPLES:

```
sage: var('x,k,n')
(x, k, n)
sage: taylor (sqrt (1 - k^2*sin(x)^2), x, 0, 6)
-1/720*(45*k^6 - 60*k^4 + 16*k^2)*x^6 - 1/24*(3*k^4 - 4*k^2)*x^4 - 1/2*k^2*x^2 + 1
```

```
>>> from sage.all import *
>>> var('x,k,n')
(x, k, n)
>>> taylor (sqrt (Integer(1) - k**Integer(2)*sin(x)**Integer(2)), x, Integer(0),␣
↪Integer(6))
-1/720*(45*k^6 - 60*k^4 + 16*k^2)*x^6 - 1/24*(3*k^4 - 4*k^2)*x^4 - 1/2*k^2*x^2 + 1
```

```
sage: taylor ((x + 1)^n, x, 0, 4)
1/24*(n^4 - 6*n^3 + 11*n^2 - 6*n)*x^4 + 1/6*(n^3 - 3*n^2 + 2*n)*x^3 + 1/2*(n^2 -␣
↪n)*x^2 + n*x + 1
```

```
>>> from sage.all import *
>>> taylor ((x + Integer(1))**n, x, Integer(0), Integer(4))
1/24*(n^4 - 6*n^3 + 11*n^2 - 6*n)*x^4 + 1/6*(n^3 - 3*n^2 + 2*n)*x^3 + 1/2*(n^2 -␣
↪n)*x^2 + n*x + 1
```

```
sage: taylor ((x + 1)^n, x, 0, 4)
1/24*(n^4 - 6*n^3 + 11*n^2 - 6*n)*x^4 + 1/6*(n^3 - 3*n^2 + 2*n)*x^3 + 1/2*(n^2 -␣
↪n)*x^2 + n*x + 1
```

```
>>> from sage.all import *
>>> taylor ((x + Integer(1))**n, x, Integer(0), Integer(4))
1/24*(n^4 - 6*n^3 + 11*n^2 - 6*n)*x^4 + 1/6*(n^3 - 3*n^2 + 2*n)*x^3 + 1/2*(n^2 -␣
↪n)*x^2 + n*x + 1
```

Taylor polynomial in two variables:

```
sage: x,y=var('x y'); taylor(x*y^3,(x,1),(y,-1),4)
(x - 1)*(y + 1)^3 - 3*(x - 1)*(y + 1)^2 + (y + 1)^3 + 3*(x - 1)*(y + 1) - 3*(y +␣
↪1)^2 - x + 3*y + 3
```

```
>>> from sage.all import *
>>> x,y=var('x y'); taylor(x*y**Integer(3),(x,Integer(1)),(y,-Integer(1)),
↪Integer(4))
(x - 1)*(y + 1)^3 - 3*(x - 1)*(y + 1)^2 + (y + 1)^3 + 3*(x - 1)*(y + 1) - 3*(y +␣
↪1)^2 - x + 3*y + 3
```

2.13 Symbolic Integration

class sage.symbolic.integration.integral.**DefiniteIntegral**

Bases: *BuiltinFunction*

The symbolic function representing a definite integral.

EXAMPLES:

```
sage: from sage.symbolic.integration.integral import definite_integral
sage: definite_integral(sin(x),x,0,pi)
2
```

```
>>> from sage.all import *
>>> from sage.symbolic.integration.integral import definite_integral
>>> definite_integral(sin(x),x,Integer(0),pi)
2
```

class sage.symbolic.integration.integral.**IndefiniteIntegral**

Bases: *BuiltinFunction*

Class to represent an indefinite integral.

EXAMPLES:

```
sage: from sage.symbolic.integration.integral import indefinite_integral
sage: indefinite_integral(log(x), x) #indirect doctest
x*log(x) - x
sage: indefinite_integral(x^2, x)
1/3*x^3
sage: indefinite_integral(4*x*log(x), x)
2*x^2*log(x) - x^2
sage: indefinite_integral(exp(x), 2*x)
2*e^x
```

```
>>> from sage.all import *
>>> from sage.symbolic.integration.integral import indefinite_integral
>>> indefinite_integral(log(x), x) #indirect doctest
x*log(x) - x
>>> indefinite_integral(x**Integer(2), x)
1/3*x^3
>>> indefinite_integral(Integer(4)*x*log(x), x)
2*x^2*log(x) - x^2
>>> indefinite_integral(exp(x), Integer(2)*x)
2*e^x
```

sage.symbolic.integration.integral.**integral**(*expression*, *v=None*, *a=None*, *b=None*, *algorithm=None*, *hold=False*)

Return the indefinite integral with respect to the variable v, ignoring the constant of integration. Or, if endpoints a and b are specified, returns the definite integral over the interval $[a, b]$.

If `self` has only one variable, then it returns the integral with respect to that variable.

If definite integration fails, it could be still possible to evaluate the definite integral using indefinite integration with the Newton - Leibniz theorem (however, the user has to ensure that the indefinite integral is continuous on the compact interval $[a, b]$ and this theorem can be applied).

INPUT:

- `v` – a variable or variable name; this can also be a tuple of the variable (optional) and endpoints (i.e., `(x,0,1)` or `(0,1)`)
- `a` – (optional) lower endpoint of definite integral
- `b` – (optional) upper endpoint of definite integral
- `algorithm` – (default: `'maxima'`, `'libgiac'` and `'sympy'`) one of
  - `'maxima'` – use maxima
  - `'sympy'` – use sympy (also in Sage)
  - `'mathematica_free'` – use http://integrals.wolfram.com/
  - `'fricas'` – use FriCAS (the optional fricas spkg has to be installed)
  - `'giac'` – use libgiac
  - `'libgiac'` – use libgiac (alias for `'giac'`)

To prevent automatic evaluation, use the `hold` argument.

See also

To integrate a polynomial over a polytope, use the optional `latte_int` package `sage.geometry.polyhedron.base.Polyhedron_base.integrate()`.

EXAMPLES:

```
sage: x = var('x')
sage: h = sin(x)/(cos(x))^2
sage: h.integral(x)
1/cos(x)
```

```
>>> from sage.all import *
>>> x = var('x')
>>> h = sin(x)/(cos(x))**Integer(2)
>>> h.integral(x)
1/cos(x)
```

```
sage: f = x^2/(x+1)^3
sage: f.integral(x)
1/2*(4*x + 3)/(x^2 + 2*x + 1) + log(x + 1)
```

```
>>> from sage.all import *
>>> f = x**Integer(2)/(x+Integer(1))**Integer(3)
>>> f.integral(x)
1/2*(4*x + 3)/(x^2 + 2*x + 1) + log(x + 1)
```

```
sage: f = x*cos(x^2)
sage: f.integral(x, 0, sqrt(pi))
0
sage: f.integral(x, a=-pi, b=pi)
0
```

```
>>> from sage.all import *
>>> f = x*cos(x**Integer(2))
>>> f.integral(x, Integer(0), sqrt(pi))
0
>>> f.integral(x, a=-pi, b=pi)
0
```

```
sage: f(x) = sin(x)
sage: f.integral(x, 0, pi/2)
1
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(sin(x)).function(x)
>>> f.integral(x, Integer(0), pi/Integer(2))
1
```

The variable is required, but the endpoints are optional:

```
sage: y = var('y')
sage: integral(sin(x), x)
-cos(x)
sage: integral(sin(x), y)
y*sin(x)
sage: integral(sin(x), x, pi, 2*pi)
-2
sage: integral(sin(x), y, pi, 2*pi)
pi*sin(x)
sage: integral(sin(x), (x, pi, 2*pi))
-2
sage: integral(sin(x), (y, pi, 2*pi))
pi*sin(x)
```

```
>>> from sage.all import *
>>> y = var('y')
>>> integral(sin(x), x)
-cos(x)
>>> integral(sin(x), y)
y*sin(x)
>>> integral(sin(x), x, pi, Integer(2)*pi)
-2
>>> integral(sin(x), y, pi, Integer(2)*pi)
```

(continues on next page)

(continued from previous page)

```
pi*sin(x)
>>> integral(sin(x), (x, pi, Integer(2)*pi))
-2
>>> integral(sin(x), (y, pi, Integer(2)*pi))
pi*sin(x)
```

Using the `hold` parameter it is possible to prevent automatic evaluation, which can then be evaluated via *`simplify()`*:

```
sage: integral(x^2, x, 0, 3)
9
sage: a = integral(x^2, x, 0, 3, hold=True) ; a
integrate(x^2, x, 0, 3)
sage: a.simplify()
9
```

```
>>> from sage.all import *
>>> integral(x**Integer(2), x, Integer(0), Integer(3))
9
>>> a = integral(x**Integer(2), x, Integer(0), Integer(3), hold=True) ; a
integrate(x^2, x, 0, 3)
>>> a.simplify()
9
```

Constraints are sometimes needed:

```
sage: var('x, n')
(x, n)
sage: integral(x^n,x)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(n>0)', see `assume?`
for more details)
Is n equal to -1?
sage: assume(n > 0)
sage: integral(x^n,x)
x^(n + 1)/(n + 1)
sage: forget()
```

```
>>> from sage.all import *
>>> var('x, n')
(x, n)
>>> integral(x**n,x)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(n>0)', see `assume?`
for more details)
Is n equal to -1?
```

(continues on next page)

(continued from previous page)

```
>>> assume(n > Integer(0))
>>> integral(x**n,x)
x^(n + 1)/(n + 1)
>>> forget()
```

Usually the constraints are of sign, but others are possible:

```
sage: assume(n==-1)
sage: integral(x^n,x)
log(x)
```

```
>>> from sage.all import *
>>> assume(n==-Integer(1))
>>> integral(x**n,x)
log(x)
```

Note that an exception is raised when a definite integral is divergent:

```
sage: forget() # always remember to forget assumptions you no longer need
sage: integrate(1/x^3,(x,0,1))
Traceback (most recent call last):
...
ValueError: Integral is divergent.
sage: integrate(1/x^3,x,-1,3)
Traceback (most recent call last):
...
ValueError: Integral is divergent.
```

```
>>> from sage.all import *
>>> forget() # always remember to forget assumptions you no longer need
>>> integrate(Integer(1)/x**Integer(3),(x,Integer(0),Integer(1)))
Traceback (most recent call last):
...
ValueError: Integral is divergent.
>>> integrate(Integer(1)/x**Integer(3),x,-Integer(1),Integer(3))
Traceback (most recent call last):
...
ValueError: Integral is divergent.
```

But Sage can calculate the convergent improper integral of this function:

```
sage: integrate(1/x^3,x,1,infinity)
1/2
```

```
>>> from sage.all import *
>>> integrate(Integer(1)/x**Integer(3),x,Integer(1),infinity)
1/2
```

The examples in the Maxima documentation:

```
sage: var('x, y, z, b')
(x, y, z, b)
```

(continues on next page)

(continued from previous page)

```
sage: integral(sin(x)^3, x)
1/3*cos(x)^3 - cos(x)
sage: integral(x/sqrt(b^2-x^2), b)
x*log(2*b + 2*sqrt(b^2 - x^2))
sage: integral(x/sqrt(b^2-x^2), x)
-sqrt(b^2 - x^2)
sage: integral(cos(x)^2 * exp(x), x, 0, pi)
3/5*e^pi - 3/5
sage: integral(x^2 * exp(-x^2), x, -oo, oo)
1/2*sqrt(pi)
```

```
>>> from sage.all import *
>>> var('x, y, z, b')
(x, y, z, b)
>>> integral(sin(x)**Integer(3), x)
1/3*cos(x)^3 - cos(x)
>>> integral(x/sqrt(b**Integer(2)-x**Integer(2)), b)
x*log(2*b + 2*sqrt(b^2 - x^2))
>>> integral(x/sqrt(b**Integer(2)-x**Integer(2)), x)
-sqrt(b^2 - x^2)
>>> integral(cos(x)**Integer(2) * exp(x), x, Integer(0), pi)
3/5*e^pi - 3/5
>>> integral(x**Integer(2) * exp(-x**Integer(2)), x, -oo, oo)
1/2*sqrt(pi)
```

We integrate the same function in both Mathematica and Sage (via Maxima):

```
sage: _ = var('x, y, z')
sage: f = sin(x^2) + y^z
sage: g = mathematica(f)                                  # optional - mathematica
sage: print(g)                                            # optional - mathematica
          z        2
         y  + Sin[x ]
sage: print(g.Integrate(x))                               # optional - mathematica
            z        Pi                 2
         x y  + Sqrt[--] FresnelS[Sqrt[--] x]
                     2                  Pi
sage: print(f.integral(x))
x*y^z + 1/16*sqrt(pi)*((I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sqrt(2)*x)
                        + (I - 1)*sqrt(2)*erf((1/2*I - 1/2)*sqrt(2)*x)
                        - (I - 1)*sqrt(2)*erf(sqrt(-I)*x)
                        + (I + 1)*sqrt(2)*erf((-1)^(1/4)*x))
```

```
>>> from sage.all import *
>>> _ = var('x, y, z')
>>> f = sin(x**Integer(2)) + y**z
>>> g = mathematica(f)                                    # optional - mathematica
>>> print(g)                                              # optional - mathematica
          z        2
         y  + Sin[x ]
>>> print(g.Integrate(x))                                 # optional - mathematica
            z        Pi                 2
```

(continues on next page)

(continued from previous page)

```
        x y  + Sqrt[--] FresnelS[Sqrt[--] x]
                    2                  Pi
>>> print(f.integral(x))
x*y^z + 1/16*sqrt(pi)*((I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sqrt(2)*x)
                        + (I - 1)*sqrt(2)*erf((1/2*I - 1/2)*sqrt(2)*x)
                        - (I - 1)*sqrt(2)*erf(sqrt(-I)*x)
                        + (I + 1)*sqrt(2)*erf((-1)^(1/4)*x))
```

Alternatively, just use algorithm='mathematica_free' to integrate via Mathematica over the internet (does NOT require a Mathematica license!):

```
sage: _ = var('x, y, z')  # optional - internet
sage: f = sin(x^2) + y^z   # optional - internet
sage: f.integrate(x, algorithm='mathematica_free')   # optional - internet
x*y^z + sqrt(1/2)*sqrt(pi)*fresnel_sin(sqrt(2)*x/sqrt(pi))
```

```
>>> from sage.all import *
>>> _ = var('x, y, z')  # optional - internet
>>> f = sin(x**Integer(2)) + y**z   # optional - internet
>>> f.integrate(x, algorithm='mathematica_free')   # optional - internet
x*y^z + sqrt(1/2)*sqrt(pi)*fresnel_sin(sqrt(2)*x/sqrt(pi))
```

We can also use Sympy:

```
sage: integrate(x*sin(log(x)), x)
-1/5*x^2*(cos(log(x)) - 2*sin(log(x)))
sage: integrate(x*sin(log(x)), x, algorithm='sympy')                          #␣
↪needs sympy
-1/5*x^2*cos(log(x)) + 2/5*x^2*sin(log(x))
sage: _ = var('y, z')
sage: (x^y - z).integrate(y)
-y*z + x^y/log(x)
sage: (x^y - z).integrate(y, algorithm='sympy')                               #␣
↪needs sympy
-y*z + cases(((log(x) != 0, x^y/log(x)), (1, y)))
```

```
>>> from sage.all import *
>>> integrate(x*sin(log(x)), x)
-1/5*x^2*(cos(log(x)) - 2*sin(log(x)))
>>> integrate(x*sin(log(x)), x, algorithm='sympy')                          #␣
↪needs sympy
-1/5*x^2*cos(log(x)) + 2/5*x^2*sin(log(x))
>>> _ = var('y, z')
>>> (x**y - z).integrate(y)
-y*z + x^y/log(x)
>>> (x**y - z).integrate(y, algorithm='sympy')                               #␣
↪needs sympy
-y*z + cases(((log(x) != 0, x^y/log(x)), (1, y)))
```

We integrate the above function in Maple now:

```
sage: g = maple(f); g.sort()          # optional - maple
y^z+sin(x^2)
sage: g.integrate(x).sort()           # optional - maple
x*y^z+1/2*2^(1/2)*Pi^(1/2)*FresnelS(2^(1/2)/Pi^(1/2)*x)
```

```
>>> from sage.all import *
>>> g = maple(f); g.sort()          # optional - maple
y^z+sin(x^2)
>>> g.integrate(x).sort()           # optional - maple
x*y^z+1/2*2^(1/2)*Pi^(1/2)*FresnelS(2^(1/2)/Pi^(1/2)*x)
```

We next integrate a function with no closed form integral. Notice that the answer comes back as an expression that contains an integral itself.

```
sage: A = integral(1/ ((x-4) * (x^4+x+1)), x); A
integrate(1/((x^4 + x + 1)*(x - 4)), x)
```

```
>>> from sage.all import *
>>> A = integral(Integer(1)/ ((x-Integer(4)) * (x**Integer(4)+x+Integer(1))), x);
↪A
integrate(1/((x^4 + x + 1)*(x - 4)), x)
```

Sometimes, in this situation, using the algorithm "maxima" gives instead a partially integrated answer:

```
sage: integral(1/(x**7-1),x,algorithm='maxima')
-1/7*integrate((x^5 + 2*x^4 + 3*x^3 + 4*x^2 + 5*x + 6)/(x^6 + x^5 + x^4 + x^3 + x^
↪2 + x + 1), x) + 1/7*log(x - 1)
```

```
>>> from sage.all import *
>>> integral(Integer(1)/(x**Integer(7)-Integer(1)),x,algorithm='maxima')
-1/7*integrate((x^5 + 2*x^4 + 3*x^3 + 4*x^2 + 5*x + 6)/(x^6 + x^5 + x^4 + x^3 + x^
↪2 + x + 1), x) + 1/7*log(x - 1)
```

We now show that floats are not converted to rationals automatically since we by default have keepfloat: true in maxima.

```
sage: integral(e^(-x^2),(x, 0, 0.1))
0.05623145800914245*sqrt(pi)
```

```
>>> from sage.all import *
>>> integral(e**(-x**Integer(2)),(x, Integer(0), RealNumber('0.1')))
0.05623145800914245*sqrt(pi)
```

An example of an integral that fricas can integrate:

```
sage: f(x) = sqrt(x+sqrt(1+x^2))/x
sage: integrate(f(x), x, algorithm='fricas')       # optional - fricas
2*sqrt(x + sqrt(x^2 + 1)) - 2*arctan(sqrt(x + sqrt(x^2 + 1)))
 - log(sqrt(x + sqrt(x^2 + 1)) + 1) + log(sqrt(x + sqrt(x^2 + 1)) - 1)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_
```

(continues on next page)

(continued from previous page)

```
↪expression(sqrt(x+sqrt(Integer(1)+x**Integer(2)))/x).function(x)
>>> integrate(f(x), x, algorithm='fricas')      # optional - fricas
2*sqrt(x + sqrt(x^2 + 1)) - 2*arctan(sqrt(x + sqrt(x^2 + 1)))
 - log(sqrt(x + sqrt(x^2 + 1)) + 1) + log(sqrt(x + sqrt(x^2 + 1)) - 1)
```

where the default integrator obtains another answer:

```
sage: integrate(f(x), x)  # long time
1/8*sqrt(x)*gamma(1/4)*gamma(-1/4)^2*hypergeometric((-1/4, -1/4, 1/4),
                                                    (1/2, 3/4),
                                                    -1/x^2)/(pi*gamma(3/4))
```

```
>>> from sage.all import *
>>> integrate(f(x), x)  # long time
1/8*sqrt(x)*gamma(1/4)*gamma(-1/4)^2*hypergeometric((-1/4, -1/4, 1/4),
                                                    (1/2, 3/4),
                                                    -1/x^2)/(pi*gamma(3/4))
```

The following definite integral is not found by maxima:

```
sage: f(x) = (x^4 - 3*x^2 + 6) / (x^6 - 5*x^4 + 5*x^2 + 4)
sage: integrate(f(x), x, 1, 2, algorithm='maxima')
integrate((x^4 - 3*x^2 + 6)/(x^6 - 5*x^4 + 5*x^2 + 4), x, 1, 2)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression((x**Integer(4) -␣
↪Integer(3)*x**Integer(2) + Integer(6)) / (x**Integer(6) -␣
↪Integer(5)*x**Integer(4) + Integer(5)*x**Integer(2) + Integer(4))).function(x)
>>> integrate(f(x), x, Integer(1), Integer(2), algorithm='maxima')
integrate((x^4 - 3*x^2 + 6)/(x^6 - 5*x^4 + 5*x^2 + 4), x, 1, 2)
```

but is nevertheless computed:

```
sage: integrate(f(x), x, 1, 2)
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

```
>>> from sage.all import *
>>> integrate(f(x), x, Integer(1), Integer(2))
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

Both fricas and sympy give the correct result:

```
sage: integrate(f(x), x, 1, 2, algorithm='fricas')  # optional - fricas
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
sage: integrate(f(x), x, 1, 2, algorithm='sympy')                          #␣
↪needs sympy
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

```
>>> from sage.all import *
>>> integrate(f(x), x, Integer(1), Integer(2), algorithm='fricas')  # optional -␣
↪fricas
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

(continues on next page)

(continued from previous page)

```
>>> integrate(f(x), x, Integer(1), Integer(2), algorithm='sympy')                    ␣
↪               # needs sympy
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

Using Giac to integrate the absolute value of a trigonometric expression. If Giac is installed, this will be attempted automatically in the event that Maxima is unable to integrate the expression:

```
sage: # needs sage.libs.giac
sage: result = integrate(abs(cos(x)), x, 0, 2*pi, algorithm='giac')
...
sage: result
4
sage: result = integrate(abs(cos(x)), x, 0, 2*pi)
...
sage: result
4
```

```
>>> from sage.all import *
>>> # needs sage.libs.giac
>>> result = integrate(abs(cos(x)), x, Integer(0), Integer(2)*pi, algorithm='giac
↪')
...
>>> result
4
>>> result = integrate(abs(cos(x)), x, Integer(0), Integer(2)*pi)
...
>>> result
4
```

ALIASES: integral() and integrate() are the same.

EXAMPLES:

Here is an example where we have to use assume:

```
sage: a,b = var('a,b')
sage: integrate(1/(x^3 *(a+b*x)^(1/3)), x)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)', see `assume?`
for more details)
Is a positive or negative?
```

```
>>> from sage.all import *
>>> a,b = var('a,b')
>>> integrate(Integer(1)/(x**Integer(3) *(a+b*x)**(Integer(1)/Integer(3))), x)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)', see `assume?`
```

(continues on next page)

(continued from previous page)

```
for more details)
Is a positive or negative?
```

So we just assume that $a > 0$ and the integral works:

```
sage: assume(a>0)
sage: integrate(1/(x^3 *(a+b*x)^(1/3)), x)
2/9*sqrt(3)*b^2*arctan(1/3*sqrt(3)*(2*(b*x + a)^(1/3) + a^(1/3))/a^(1/3))/a^(7/3)
 - 1/9*b^2*log((b*x + a)^(2/3) + (b*x + a)^(1/3)*a^(1/3) + a^(2/3))/a^(7/3)
 + 2/9*b^2*log((b*x + a)^(1/3) - a^(1/3))/a^(7/3) + 1/6*(4*(b*x + a)^(5/3)*b^2
 - 7*(b*x + a)^(2/3)*a*b^2)/((b*x + a)^2*a^2 - 2*(b*x + a)*a^3 + a^4)
```

```
>>> from sage.all import *
>>> assume(a>Integer(0))
>>> integrate(Integer(1)/(x**Integer(3) *(a+b*x)**(Integer(1)/Integer(3))), x)
2/9*sqrt(3)*b^2*arctan(1/3*sqrt(3)*(2*(b*x + a)^(1/3) + a^(1/3))/a^(1/3))/a^(7/3)
 - 1/9*b^2*log((b*x + a)^(2/3) + (b*x + a)^(1/3)*a^(1/3) + a^(2/3))/a^(7/3)
 + 2/9*b^2*log((b*x + a)^(1/3) - a^(1/3))/a^(7/3) + 1/6*(4*(b*x + a)^(5/3)*b^2
 - 7*(b*x + a)^(2/3)*a*b^2)/((b*x + a)^2*a^2 - 2*(b*x + a)*a^3 + a^4)
```

sage.symbolic.integration.integral.**integrate**(*expression*, *v=None*, *a=None*, *b=None*, *algorithm=None*, *hold=False*)

Return the indefinite integral with respect to the variable v, ignoring the constant of integration. Or, if endpoints a and b are specified, returns the definite integral over the interval $[a, b]$.

If `self` has only one variable, then it returns the integral with respect to that variable.

If definite integration fails, it could be still possible to evaluate the definite integral using indefinite integration with the Newton - Leibniz theorem (however, the user has to ensure that the indefinite integral is continuous on the compact interval $[a, b]$ and this theorem can be applied).

INPUT:

- `v` – a variable or variable name; this can also be a tuple of the variable (optional) and endpoints (i.e., `(x,0,1)` or `(0,1)`)
- `a` – (optional) lower endpoint of definite integral
- `b` – (optional) upper endpoint of definite integral
- `algorithm` – (default: `'maxima'`, `'libgiac'` and `'sympy'`) one of
  - `'maxima'` – use maxima
  - `'sympy'` – use sympy (also in Sage)
  - `'mathematica_free'` – use http://integrals.wolfram.com/
  - `'fricas'` – use FriCAS (the optional fricas spkg has to be installed)
  - `'giac'` – use libgiac
  - `'libgiac'` – use libgiac (alias for `'giac'`)

To prevent automatic evaluation, use the `hold` argument.

See also

To integrate a polynomial over a polytope, use the optional `latte_int` package `sage.geometry.polyhedron.base.Polyhedron_base.integrate()`.

EXAMPLES:

```
sage: x = var('x')
sage: h = sin(x)/(cos(x))^2
sage: h.integral(x)
1/cos(x)
```

```
>>> from sage.all import *
>>> x = var('x')
>>> h = sin(x)/(cos(x))**Integer(2)
>>> h.integral(x)
1/cos(x)
```

```
sage: f = x^2/(x+1)^3
sage: f.integral(x)
1/2*(4*x + 3)/(x^2 + 2*x + 1) + log(x + 1)
```

```
>>> from sage.all import *
>>> f = x**Integer(2)/(x+Integer(1))**Integer(3)
>>> f.integral(x)
1/2*(4*x + 3)/(x^2 + 2*x + 1) + log(x + 1)
```

```
sage: f = x*cos(x^2)
sage: f.integral(x, 0, sqrt(pi))
0
sage: f.integral(x, a=-pi, b=pi)
0
```

```
>>> from sage.all import *
>>> f = x*cos(x**Integer(2))
>>> f.integral(x, Integer(0), sqrt(pi))
0
>>> f.integral(x, a=-pi, b=pi)
0
```

```
sage: f(x) = sin(x)
sage: f.integral(x, 0, pi/2)
1
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(sin(x)).function(x)
>>> f.integral(x, Integer(0), pi/Integer(2))
1
```

The variable is required, but the endpoints are optional:

```
sage: y = var('y')
sage: integral(sin(x), x)
-cos(x)
sage: integral(sin(x), y)
y*sin(x)
sage: integral(sin(x), x, pi, 2*pi)
-2
sage: integral(sin(x), y, pi, 2*pi)
pi*sin(x)
sage: integral(sin(x), (x, pi, 2*pi))
-2
sage: integral(sin(x), (y, pi, 2*pi))
pi*sin(x)
```

```
>>> from sage.all import *
>>> y = var('y')
>>> integral(sin(x), x)
-cos(x)
>>> integral(sin(x), y)
y*sin(x)
>>> integral(sin(x), x, pi, Integer(2)*pi)
-2
>>> integral(sin(x), y, pi, Integer(2)*pi)
pi*sin(x)
>>> integral(sin(x), (x, pi, Integer(2)*pi))
-2
>>> integral(sin(x), (y, pi, Integer(2)*pi))
pi*sin(x)
```

Using the `hold` parameter it is possible to prevent automatic evaluation, which can then be evaluated via *`simplify()`*:

```
sage: integral(x^2, x, 0, 3)
9
sage: a = integral(x^2, x, 0, 3, hold=True) ; a
integrate(x^2, x, 0, 3)
sage: a.simplify()
9
```

```
>>> from sage.all import *
>>> integral(x**Integer(2), x, Integer(0), Integer(3))
9
>>> a = integral(x**Integer(2), x, Integer(0), Integer(3), hold=True) ; a
integrate(x^2, x, 0, 3)
>>> a.simplify()
9
```

Constraints are sometimes needed:

```
sage: var('x, n')
(x, n)
sage: integral(x^n,x)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(n>0)', see `assume?`
for more details)
Is n equal to -1?
sage: assume(n > 0)
sage: integral(x^n,x)
x^(n + 1)/(n + 1)
sage: forget()
```

```
>>> from sage.all import *
>>> var('x, n')
(x, n)
>>> integral(x**n,x)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(n>0)', see `assume?`
for more details)
Is n equal to -1?
>>> assume(n > Integer(0))
>>> integral(x**n,x)
x^(n + 1)/(n + 1)
>>> forget()
```

Usually the constraints are of sign, but others are possible:

```
sage: assume(n==-1)
sage: integral(x^n,x)
log(x)
```

```
>>> from sage.all import *
>>> assume(n==-Integer(1))
>>> integral(x**n,x)
log(x)
```

Note that an exception is raised when a definite integral is divergent:

```
sage: forget() # always remember to forget assumptions you no longer need
sage: integrate(1/x^3,(x,0,1))
Traceback (most recent call last):
...
ValueError: Integral is divergent.
sage: integrate(1/x^3,x,-1,3)
Traceback (most recent call last):
...
ValueError: Integral is divergent.
```

```
>>> from sage.all import *
>>> forget() # always remember to forget assumptions you no longer need
```

(continues on next page)

(continued from previous page)

```
>>> integrate(Integer(1)/x**Integer(3),(x,Integer(0),Integer(1)))
Traceback (most recent call last):
...
ValueError: Integral is divergent.
>>> integrate(Integer(1)/x**Integer(3),x,-Integer(1),Integer(3))
Traceback (most recent call last):
...
ValueError: Integral is divergent.
```

But Sage can calculate the convergent improper integral of this function:

```
sage: integrate(1/x^3,x,1,infinity)
1/2
```

```
>>> from sage.all import *
>>> integrate(Integer(1)/x**Integer(3),x,Integer(1),infinity)
1/2
```

The examples in the Maxima documentation:

```
sage: var('x, y, z, b')
(x, y, z, b)
sage: integral(sin(x)^3, x)
1/3*cos(x)^3 - cos(x)
sage: integral(x/sqrt(b^2-x^2), b)
x*log(2*b + 2*sqrt(b^2 - x^2))
sage: integral(x/sqrt(b^2-x^2), x)
-sqrt(b^2 - x^2)
sage: integral(cos(x)^2 * exp(x), x, 0, pi)
3/5*e^pi - 3/5
sage: integral(x^2 * exp(-x^2), x, -oo, oo)
1/2*sqrt(pi)
```

```
>>> from sage.all import *
>>> var('x, y, z, b')
(x, y, z, b)
>>> integral(sin(x)**Integer(3), x)
1/3*cos(x)^3 - cos(x)
>>> integral(x/sqrt(b**Integer(2)-x**Integer(2)), b)
x*log(2*b + 2*sqrt(b^2 - x^2))
>>> integral(x/sqrt(b**Integer(2)-x**Integer(2)), x)
-sqrt(b^2 - x^2)
>>> integral(cos(x)**Integer(2) * exp(x), x, Integer(0), pi)
3/5*e^pi - 3/5
>>> integral(x**Integer(2) * exp(-x**Integer(2)), x, -oo, oo)
1/2*sqrt(pi)
```

We integrate the same function in both Mathematica and Sage (via Maxima):

```
sage: _ = var('x, y, z')
sage: f = sin(x^2) + y^z
sage: g = mathematica(f)                           # optional - mathematica
```

(continues on next page)

(continued from previous page)

```
sage: print(g)                                          # optional - mathematica
          z        2
         y   + Sin[x ]
sage: print(g.Integrate(x))                             # optional - mathematica
            z        Pi                 2
         x y   + Sqrt[--] FresnelS[Sqrt[--] x]
                     2                  Pi
sage: print(f.integral(x))
x*y^z + 1/16*sqrt(pi)*((I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sqrt(2)*x)
                        + (I - 1)*sqrt(2)*erf((1/2*I - 1/2)*sqrt(2)*x)
                        - (I - 1)*sqrt(2)*erf(sqrt(-I)*x)
                        + (I + 1)*sqrt(2)*erf((-1)^(1/4)*x))
```

```
>>> from sage.all import *
>>> _ = var('x, y, z')
>>> f = sin(x**Integer(2)) + y**z
>>> g = mathematica(f)                                # optional - mathematica
>>> print(g)                                          # optional - mathematica
          z        2
         y   + Sin[x ]
>>> print(g.Integrate(x))                             # optional - mathematica
            z        Pi                 2
         x y   + Sqrt[--] FresnelS[Sqrt[--] x]
                     2                  Pi
>>> print(f.integral(x))
x*y^z + 1/16*sqrt(pi)*((I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sqrt(2)*x)
                        + (I - 1)*sqrt(2)*erf((1/2*I - 1/2)*sqrt(2)*x)
                        - (I - 1)*sqrt(2)*erf(sqrt(-I)*x)
                        + (I + 1)*sqrt(2)*erf((-1)^(1/4)*x))
```

Alternatively, just use algorithm='mathematica_free' to integrate via Mathematica over the internet (does NOT require a Mathematica license!):

```
sage: _ = var('x, y, z')   # optional - internet
sage: f = sin(x^2) + y^z    # optional - internet
sage: f.integrate(x, algorithm='mathematica_free')    # optional - internet
x*y^z + sqrt(1/2)*sqrt(pi)*fresnel_sin(sqrt(2)*x/sqrt(pi))
```

```
>>> from sage.all import *
>>> _ = var('x, y, z')  # optional - internet
>>> f = sin(x**Integer(2)) + y**z    # optional - internet
>>> f.integrate(x, algorithm='mathematica_free')    # optional - internet
x*y^z + sqrt(1/2)*sqrt(pi)*fresnel_sin(sqrt(2)*x/sqrt(pi))
```

We can also use Sympy:

```
sage: integrate(x*sin(log(x)), x)
-1/5*x^2*(cos(log(x)) - 2*sin(log(x)))
sage: integrate(x*sin(log(x)), x, algorithm='sympy')                         #␣
↪needs sympy
-1/5*x^2*cos(log(x)) + 2/5*x^2*sin(log(x))
sage: _ = var('y, z')
```

(continues on next page)

(continued from previous page)

```
sage: (x^y - z).integrate(y)
-y*z + x^y/log(x)
sage: (x^y - z).integrate(y, algorithm='sympy')                                #␣
↪needs sympy
-y*z + cases(((log(x) != 0, x^y/log(x)), (1, y)))
```

```
>>> from sage.all import *
>>> integrate(x*sin(log(x)), x)
-1/5*x^2*(cos(log(x)) - 2*sin(log(x)))
>>> integrate(x*sin(log(x)), x, algorithm='sympy')                            #␣
↪needs sympy
-1/5*x^2*cos(log(x)) + 2/5*x^2*sin(log(x))
>>> _ = var('y, z')
>>> (x**y - z).integrate(y)
-y*z + x^y/log(x)
>>> (x**y - z).integrate(y, algorithm='sympy')                                #␣
↪needs sympy
-y*z + cases(((log(x) != 0, x^y/log(x)), (1, y)))
```

We integrate the above function in Maple now:

```
sage: g = maple(f); g.sort()          # optional - maple
y^z+sin(x^2)
sage: g.integrate(x).sort()           # optional - maple
x*y^z+1/2*2^(1/2)*Pi^(1/2)*FresnelS(2^(1/2)/Pi^(1/2)*x)
```

```
>>> from sage.all import *
>>> g = maple(f); g.sort()          # optional - maple
y^z+sin(x^2)
>>> g.integrate(x).sort()           # optional - maple
x*y^z+1/2*2^(1/2)*Pi^(1/2)*FresnelS(2^(1/2)/Pi^(1/2)*x)
```

We next integrate a function with no closed form integral. Notice that the answer comes back as an expression that contains an integral itself.

```
sage: A = integral(1/ ((x-4) * (x^4+x+1)), x); A
integrate(1/((x^4 + x + 1)*(x - 4)), x)
```

```
>>> from sage.all import *
>>> A = integral(Integer(1)/ ((x-Integer(4)) * (x**Integer(4)+x+Integer(1))), x);␣
↪A
integrate(1/((x^4 + x + 1)*(x - 4)), x)
```

Sometimes, in this situation, using the algorithm "maxima" gives instead a partially integrated answer:

```
sage: integral(1/(x**7-1),x,algorithm='maxima')
-1/7*integrate((x^5 + 2*x^4 + 3*x^3 + 4*x^2 + 5*x + 6)/(x^6 + x^5 + x^4 + x^3 + x^
↪2 + x + 1), x) + 1/7*log(x - 1)
```

```
>>> from sage.all import *
>>> integral(Integer(1)/(x**Integer(7)-Integer(1)),x,algorithm='maxima')
```

(continues on next page)

(continued from previous page)

```
-1/7*integrate((x^5 + 2*x^4 + 3*x^3 + 4*x^2 + 5*x + 6)/(x^6 + x^5 + x^4 + x^3 + x^
↪2 + x + 1), x) + 1/7*log(x - 1)
```

We now show that floats are not converted to rationals automatically since we by default have keepfloat: true in maxima.

```
sage: integral(e^(-x^2),(x, 0, 0.1))
0.0562314580091424*sqrt(pi)
```

```
>>> from sage.all import *
>>> integral(e**(-x**Integer(2)),(x, Integer(0), RealNumber('0.1')))
0.0562314580091424*sqrt(pi)
```

An example of an integral that fricas can integrate:

```
sage: f(x) = sqrt(x+sqrt(1+x^2))/x
sage: integrate(f(x), x, algorithm='fricas')       # optional - fricas
2*sqrt(x + sqrt(x^2 + 1)) - 2*arctan(sqrt(x + sqrt(x^2 + 1)))
 - log(sqrt(x + sqrt(x^2 + 1)) + 1) + log(sqrt(x + sqrt(x^2 + 1)) - 1)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_
↪expression(sqrt(x+sqrt(Integer(1)+x**Integer(2)))/x).function(x)
>>> integrate(f(x), x, algorithm='fricas')       # optional - fricas
2*sqrt(x + sqrt(x^2 + 1)) - 2*arctan(sqrt(x + sqrt(x^2 + 1)))
 - log(sqrt(x + sqrt(x^2 + 1)) + 1) + log(sqrt(x + sqrt(x^2 + 1)) - 1)
```

where the default integrator obtains another answer:

```
sage: integrate(f(x), x)   # long time
1/8*sqrt(x)*gamma(1/4)*gamma(-1/4)^2*hypergeometric((-1/4, -1/4, 1/4),
                                                    (1/2, 3/4),
                                                    -1/x^2)/(pi*gamma(3/4))
```

```
>>> from sage.all import *
>>> integrate(f(x), x)   # long time
1/8*sqrt(x)*gamma(1/4)*gamma(-1/4)^2*hypergeometric((-1/4, -1/4, 1/4),
                                                    (1/2, 3/4),
                                                    -1/x^2)/(pi*gamma(3/4))
```

The following definite integral is not found by maxima:

```
sage: f(x) = (x^4 - 3*x^2 + 6) / (x^6 - 5*x^4 + 5*x^2 + 4)
sage: integrate(f(x), x, 1, 2, algorithm='maxima')
integrate((x^4 - 3*x^2 + 6)/(x^6 - 5*x^4 + 5*x^2 + 4), x, 1, 2)
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression((x**Integer(4) -␣
↪Integer(3)*x**Integer(2) + Integer(6)) / (x**Integer(6) -␣
↪Integer(5)*x**Integer(4) + Integer(5)*x**Integer(2) + Integer(4))).function(x)
>>> integrate(f(x), x, Integer(1), Integer(2), algorithm='maxima')
integrate((x^4 - 3*x^2 + 6)/(x^6 - 5*x^4 + 5*x^2 + 4), x, 1, 2)
```

but is nevertheless computed:

```
sage: integrate(f(x), x, 1, 2)
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

```
>>> from sage.all import *
>>> integrate(f(x), x, Integer(1), Integer(2))
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

Both fricas and sympy give the correct result:

```
sage: integrate(f(x), x, 1, 2, algorithm='fricas')  # optional - fricas
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
sage: integrate(f(x), x, 1, 2, algorithm='sympy')                          #␣
↪needs sympy
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

```
>>> from sage.all import *
>>> integrate(f(x), x, Integer(1), Integer(2), algorithm='fricas')  # optional -␣
↪fricas
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
>>> integrate(f(x), x, Integer(1), Integer(2), algorithm='sympy')              ␣
↪              # needs sympy
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

Using Giac to integrate the absolute value of a trigonometric expression. If Giac is installed, this will be attempted automatically in the event that Maxima is unable to integrate the expression:

```
sage: # needs sage.libs.giac
sage: result = integrate(abs(cos(x)), x, 0, 2*pi, algorithm='giac')
...
sage: result
4
sage: result = integrate(abs(cos(x)), x, 0, 2*pi)
...
sage: result
4
```

```
>>> from sage.all import *
>>> # needs sage.libs.giac
>>> result = integrate(abs(cos(x)), x, Integer(0), Integer(2)*pi, algorithm='giac
↪')
...
>>> result
4
>>> result = integrate(abs(cos(x)), x, Integer(0), Integer(2)*pi)
...
>>> result
4
```

ALIASES: integral() and integrate() are the same.

EXAMPLES:

Here is an example where we have to use assume:

```
sage: a,b = var('a,b')
sage: integrate(1/(x^3 *(a+b*x)^(1/3)), x)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)', see `assume?`
for more details)
Is a positive or negative?
```

```
>>> from sage.all import *
>>> a,b = var('a,b')
>>> integrate(Integer(1)/(x**Integer(3) *(a+b*x)**(Integer(1)/Integer(3))), x)
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional
constraints; using the 'assume' command before evaluation
*may* help (example of legal syntax is 'assume(a>0)', see `assume?`
for more details)
Is a positive or negative?
```

So we just assume that $a > 0$ and the integral works:

```
sage: assume(a>0)
sage: integrate(1/(x^3 *(a+b*x)^(1/3)), x)
2/9*sqrt(3)*b^2*arctan(1/3*sqrt(3)*(2*(b*x + a)^(1/3) + a^(1/3))/a^(1/3))/a^(7/3)
 - 1/9*b^2*log((b*x + a)^(2/3) + (b*x + a)^(1/3)*a^(1/3) + a^(2/3))/a^(7/3)
 + 2/9*b^2*log((b*x + a)^(1/3) - a^(1/3))/a^(7/3) + 1/6*(4*(b*x + a)^(5/3)*b^2
 - 7*(b*x + a)^(2/3)*a*b^2)/((b*x + a)^2*a^2 - 2*(b*x + a)*a^3 + a^4)
```

```
>>> from sage.all import *
>>> assume(a>Integer(0))
>>> integrate(Integer(1)/(x**Integer(3) *(a+b*x)**(Integer(1)/Integer(3))), x)
2/9*sqrt(3)*b^2*arctan(1/3*sqrt(3)*(2*(b*x + a)^(1/3) + a^(1/3))/a^(1/3))/a^(7/3)
 - 1/9*b^2*log((b*x + a)^(2/3) + (b*x + a)^(1/3)*a^(1/3) + a^(2/3))/a^(7/3)
 + 2/9*b^2*log((b*x + a)^(1/3) - a^(1/3))/a^(7/3) + 1/6*(4*(b*x + a)^(5/3)*b^2
 - 7*(b*x + a)^(2/3)*a*b^2)/((b*x + a)^2*a^2 - 2*(b*x + a)*a^3 + a^4)
```

2.14 TESTS::

sage.symbolic.integration.external.**fricas_integrator**(*expression*, *v*, *a=None*, *b=None*, *noPole=True*)

Integration using FriCAS.

EXAMPLES:

```
sage: # optional - fricas
sage: from sage.symbolic.integration.external import fricas_integrator
sage: fricas_integrator(sin(x), x)
-cos(x)
sage: fricas_integrator(cos(x), x)
```

(continues on next page)

(continued from previous page)

```
sin(x)
sage: fricas_integrator(1/(x^2-2), x, 0, 1)
-1/8*sqrt(2)*(log(2) - log(-24*sqrt(2) + 34))
sage: fricas_integrator(1/(x^2+6), x, -oo, oo)
1/6*sqrt(6)*pi
```

```
>>> from sage.all import *
>>> # optional - fricas
>>> from sage.symbolic.integration.external import fricas_integrator
>>> fricas_integrator(sin(x), x)
-cos(x)
>>> fricas_integrator(cos(x), x)
sin(x)
>>> fricas_integrator(Integer(1)/(x**Integer(2)-Integer(2)), x, Integer(0),␣
↪Integer(1))
-1/8*sqrt(2)*(log(2) - log(-24*sqrt(2) + 34))
>>> fricas_integrator(Integer(1)/(x**Integer(2)+Integer(6)), x, -oo, oo)
1/6*sqrt(6)*pi
```

sage.symbolic.integration.external.**libgiac_integrator**(*expression*, *v*, *a=None*, *b=None*)

Integration using libgiac.

EXAMPLES:

```
sage: # needs sage.libs.giac
sage: import sage.libs.giac
...
sage: from sage.symbolic.integration.external import libgiac_integrator
sage: libgiac_integrator(sin(x), x)
-cos(x)
sage: libgiac_integrator(1/(x^2+6), x, -oo, oo)
1/6*sqrt(6)*pi
```

```
>>> from sage.all import *
>>> # needs sage.libs.giac
>>> import sage.libs.giac
...
>>> from sage.symbolic.integration.external import libgiac_integrator
>>> libgiac_integrator(sin(x), x)
-cos(x)
>>> libgiac_integrator(Integer(1)/(x**Integer(2)+Integer(6)), x, -oo, oo)
1/6*sqrt(6)*pi
```

sage.symbolic.integration.external.**maxima_integrator**(*expression*, *v*, *a=None*, *b=None*)

Integration using Maxima.

EXAMPLES:

```
sage: from sage.symbolic.integration.external import maxima_integrator
sage: maxima_integrator(sin(x), x)
-cos(x)
sage: maxima_integrator(cos(x), x)
```

(continues on next page)

(continued from previous page)

```
sin(x)
sage: f(x) = function('f')(x)
sage: maxima_integrator(f(x), x)
integrate(f(x), x)
```

```
>>> from sage.all import *
>>> from sage.symbolic.integration.external import maxima_integrator
>>> maxima_integrator(sin(x), x)
-cos(x)
>>> maxima_integrator(cos(x), x)
sin(x)
>>> __tmp__=var("x"); f = symbolic_expression(function('f')(x)).function(x)
>>> maxima_integrator(f(x), x)
integrate(f(x), x)
```

sage.symbolic.integration.external.**mma_free_integrator**(*expression*, *v*, *a=None*, *b=None*)

Integration using Mathematica's online integrator.

EXAMPLES:

```
sage: from sage.symbolic.integration.external import mma_free_integrator
sage: mma_free_integrator(sin(x), x) # optional - internet
-cos(x)
```

```
>>> from sage.all import *
>>> from sage.symbolic.integration.external import mma_free_integrator
>>> mma_free_integrator(sin(x), x) # optional - internet
-cos(x)
```

A definite integral:

```
sage: mma_free_integrator(e^(-x), x, a=0, b=oo) # optional - internet
1
```

```
>>> from sage.all import *
>>> mma_free_integrator(e**(-x), x, a=Integer(0), b=oo) # optional - internet
1
```

sage.symbolic.integration.external.**sympy_integrator**(*expression*, *v*, *a=None*, *b=None*)

Integration using SymPy.

EXAMPLES:

```
sage: from sage.symbolic.integration.external import sympy_integrator
sage: sympy_integrator(sin(x), x)                                              #␣
↪needs sympy
-cos(x)
sage: sympy_integrator(cos(x), x)                                              #␣
↪needs sympy
sin(x)
```

```
>>> from sage.all import *
>>> from sage.symbolic.integration.external import sympy_integrator
>>> sympy_integrator(sin(x), x)                                    #␣
↪needs sympy
-cos(x)
>>> sympy_integrator(cos(x), x)                                    #␣
↪needs sympy
sin(x)
```

2.15 A Sample Session using SymPy

In this first part, we do all of the examples in the SymPy tutorial (https://github.com/sympy/sympy/wiki/Tutorial), but using Sage instead of SymPy.

```
sage: a = Rational((1,2))
sage: a
1/2
sage: a*2
1
sage: Rational(2)^50 / Rational(10)^50
1/88817841970012523233890533447265625
sage: 1.0/2
0.500000000000000
sage: 1/2
1/2
sage: pi^2
pi^2
sage: float(pi)
3.141592653589793
sage: RealField(200)(pi)
3.1415926535897932384626433832795028841971693993751058209749
sage: float(pi + exp(1))
5.85987448204883...
sage: oo != 2
True
```

```
>>> from sage.all import *
>>> a = Rational((Integer(1),Integer(2)))
>>> a
1/2
>>> a*Integer(2)
1
>>> Rational(Integer(2))**Integer(50) / Rational(Integer(10))**Integer(50)
1/88817841970012523233890533447265625
>>> RealNumber('1.0')/Integer(2)
0.500000000000000
>>> Integer(1)/Integer(2)
1/2
>>> pi**Integer(2)
pi^2
>>> float(pi)
```

(continues on next page)

(continued from previous page)

```
3.141592653589793
>>> RealField(Integer(200))(pi)
3.1415926535897932384626433832795028841971693993751058209749
>>> float(pi + exp(Integer(1)))
5.85987448204883...
>>> oo != Integer(2)
True
```

```
sage: var('x y')
(x, y)
sage: x + y + x - y
2*x
sage: (x+y)^2
(x + y)^2
sage: ((x+y)^2).expand()
x^2 + 2*x*y + y^2
sage: ((x+y)^2).subs(x=1)
(y + 1)^2
sage: ((x+y)^2).subs(x=y)
4*y^2
```

```
>>> from sage.all import *
>>> var('x y')
(x, y)
>>> x + y + x - y
2*x
>>> (x+y)**Integer(2)
(x + y)^2
>>> ((x+y)**Integer(2)).expand()
x^2 + 2*x*y + y^2
>>> ((x+y)**Integer(2)).subs(x=Integer(1))
(y + 1)^2
>>> ((x+y)**Integer(2)).subs(x=y)
4*y^2
```

```
sage: limit(sin(x)/x, x=0)
1
sage: limit(x, x=oo)
+Infinity
sage: limit((5^x + 3^x)^(1/x), x=oo)
5
```

```
>>> from sage.all import *
>>> limit(sin(x)/x, x=Integer(0))
1
>>> limit(x, x=oo)
+Infinity
>>> limit((Integer(5)**x + Integer(3)**x)**(Integer(1)/x), x=oo)
5
```

```
sage: diff(sin(x), x)
cos(x)
sage: diff(sin(2*x), x)
2*cos(2*x)
sage: diff(tan(x), x)
tan(x)^2 + 1
sage: limit((tan(x+y) - tan(x))/y, y=0)
cos(x)^(-2)
sage: diff(sin(2*x), x, 1)
2*cos(2*x)
sage: diff(sin(2*x), x, 2)
-4*sin(2*x)
sage: diff(sin(2*x), x, 3)
-8*cos(2*x)
```

```
>>> from sage.all import *
>>> diff(sin(x), x)
cos(x)
>>> diff(sin(Integer(2)*x), x)
2*cos(2*x)
>>> diff(tan(x), x)
tan(x)^2 + 1
>>> limit((tan(x+y) - tan(x))/y, y=Integer(0))
cos(x)^(-2)
>>> diff(sin(Integer(2)*x), x, Integer(1))
2*cos(2*x)
>>> diff(sin(Integer(2)*x), x, Integer(2))
-4*sin(2*x)
>>> diff(sin(Integer(2)*x), x, Integer(3))
-8*cos(2*x)
```

```
sage: cos(x).taylor(x,0,10)
-1/3628800*x^10 + 1/40320*x^8 - 1/720*x^6 + 1/24*x^4 - 1/2*x^2 + 1
sage: (1/cos(x)).taylor(x,0,10)
50521/3628800*x^10 + 277/8064*x^8 + 61/720*x^6 + 5/24*x^4 + 1/2*x^2 + 1
```

```
>>> from sage.all import *
>>> cos(x).taylor(x,Integer(0),Integer(10))
-1/3628800*x^10 + 1/40320*x^8 - 1/720*x^6 + 1/24*x^4 - 1/2*x^2 + 1
>>> (Integer(1)/cos(x)).taylor(x,Integer(0),Integer(10))
50521/3628800*x^10 + 277/8064*x^8 + 61/720*x^6 + 5/24*x^4 + 1/2*x^2 + 1
```

```
sage: matrix([[1,0], [0,1]])
[1 0]
[0 1]
sage: var('x y')
(x, y)
sage: A = matrix([[1,x], [y,1]])
sage: A
[1 x]
[y 1]
sage: A^2
```

(continues on next page)

(continued from previous page)

```
[x*y + 1      2*x]
[    2*y x*y + 1]
sage: R.<x,y> = QQ[]
sage: A = matrix([[1,x], [y,1]])
sage: A^10
[x^5*y^5 + 45*x^4*y^4 + 210*x^3*y^3 + 210*x^2*y^2 + 45*x*y + 1      10*x^5*y^4 + 120*x^
↪4*y^3 + 252*x^3*y^2 + 120*x^2*y + 10*x]
[    10*x^4*y^5 + 120*x^3*y^4 + 252*x^2*y^3 + 120*x*y^2 + 10*y x^5*y^5 + 45*x^4*y^4 +␣
↪210*x^3*y^3 + 210*x^2*y^2 + 45*x*y + 1]
sage: var('x y')
(x, y)
```

```
>>> from sage.all import *
>>> matrix([[Integer(1),Integer(0)], [Integer(0),Integer(1)]])
[1 0]
[0 1]
>>> var('x y')
(x, y)
>>> A = matrix([[Integer(1),x], [y,Integer(1)]])
>>> A
[1 x]
[y 1]
>>> A**Integer(2)
[x*y + 1      2*x]
[    2*y x*y + 1]
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> A = matrix([[Integer(1),x], [y,Integer(1)]])
>>> A**Integer(10)
[x^5*y^5 + 45*x^4*y^4 + 210*x^3*y^3 + 210*x^2*y^2 + 45*x*y + 1      10*x^5*y^4 + 120*x^
↪4*y^3 + 252*x^3*y^2 + 120*x^2*y + 10*x]
[    10*x^4*y^5 + 120*x^3*y^4 + 252*x^2*y^3 + 120*x*y^2 + 10*y x^5*y^5 + 45*x^4*y^4 +␣
↪210*x^3*y^3 + 210*x^2*y^2 + 45*x*y + 1]
>>> var('x y')
(x, y)
```

And here are some actual tests of sympy:

```
sage: from sympy import Symbol, cos, sympify, pprint                                   #␣
↪needs sympy
sage: from sympy.abc import x                                                          #␣
↪needs sympy
```

```
>>> from sage.all import *
>>> from sympy import Symbol, cos, sympify, pprint                                   #␣
↪needs sympy
>>> from sympy.abc import x                                                          #␣
↪needs sympy
```

```
sage: e = (1/cos(x)^3)._sympy_(); e                                                    #␣
↪needs sympy
cos(x)**(-3)
```

(continues on next page)

(continued from previous page)

```
sage: f = e.series(x, 0, int(10)); f                                    #␣
↪needs sympy
1 + 3*x**2/2 + 11*x**4/8 + 241*x**6/240 + 8651*x**8/13440 + O(x**10)
```

```
>>> from sage.all import *
>>> e = (Integer(1)/cos(x)**Integer(3))._sympy_(); e                     ␣
↪                # needs sympy
cos(x)**(-3)
>>> f = e.series(x, Integer(0), int(Integer(10))); f                     ␣
↪               # needs sympy
1 + 3*x**2/2 + 11*x**4/8 + 241*x**6/240 + 8651*x**8/13440 + O(x**10)
```

And the pretty-printer. Since unicode characters are not working on some architectures, we disable it:

```
sage: from sympy.printing import pprint_use_unicode                     #␣
↪needs sympy
sage: prev_use = pprint_use_unicode(False)                              #␣
↪needs sympy
sage: pprint(e)                                                         #␣
↪needs sympy
   1
-------
   3
cos (x)

sage: pprint(f)                                                         #␣
↪needs sympy
       2        4         6          8
    3*x    11*x    241*x    8651*x      / 10\
1 + ---- + ----- + ------ + ------- + O\x   /
     2       8      240      13440
sage: pprint_use_unicode(prev_use)                                      #␣
↪needs sympy
False
```

```
>>> from sage.all import *
>>> from sympy.printing import pprint_use_unicode                     #␣
↪needs sympy
>>> prev_use = pprint_use_unicode(False)                              #␣
↪needs sympy
>>> pprint(e)                                                         #␣
↪needs sympy
   1
-------
   3
cos (x)

>>> pprint(f)                                                         #␣
↪needs sympy
       2        4         6          8
    3*x    11*x    241*x    8651*x      / 10\
1 + ---- + ----- + ------ + ------- + O\x   /
```

(continues on next page)

(continued from previous page)

```
     2        8       240       13440
>>> pprint_use_unicode(prev_use)                                          #␣
↪needs sympy
False
```

And the functionality to convert from sympy format to Sage format:

```
sage: e._sage_()                                                            #␣
↪needs sympy
cos(x)^(-3)
sage: e._sage_().taylor(x._sage_(), 0, 8)                                   #␣
↪needs sympy
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + 1
sage: f._sage_()                                                            #␣
↪needs sympy
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + Order(x^10) + 1
```

```
>>> from sage.all import *
>>> e._sage_()                                                            #␣
↪needs sympy
cos(x)^(-3)
>>> e._sage_().taylor(x._sage_(), Integer(0), Integer(8))                   ␣
↪                # needs sympy
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + 1
>>> f._sage_()                                                            #␣
↪needs sympy
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + Order(x^10) + 1
```

Mixing SymPy with Sage:

```
sage: # needs sympy
sage: import sympy
sage: var("x")._sympy_() + var("y")._sympy_()
x + y
sage: o = var("omega")
sage: s = sympy.Symbol("x")
sage: t1 = s + o
sage: t2 = o + s
sage: type(t1)
<class 'sympy.core.add.Add'>
sage: type(t2)
<class 'sage.symbolic.expression.Expression'>
sage: t1, t2
(omega + x, omega + x)
sage: e = sympy.sin(var("y")) + sage.functions.trig.cos(sympy.Symbol("x"))
sage: type(e)
<class 'sympy.core.add.Add'>
sage: e
sin(y) + cos(x)
sage: e=e._sage_()
sage: type(e)
<class 'sage.symbolic.expression.Expression'>
```

(continues on next page)

(continued from previous page)

```
sage: e
cos(x) + sin(y)
sage: e = sage.functions.trig.cos(var("y")**3)**4+var("x")**2
sage: e = e._sympy_()
sage: e
x**2 + cos(y**3)**4
```

```
>>> from sage.all import *
>>> # needs sympy
>>> import sympy
>>> var("x")._sympy_() + var("y")._sympy_()
x + y
>>> o = var("omega")
>>> s = sympy.Symbol("x")
>>> t1 = s + o
>>> t2 = o + s
>>> type(t1)
<class 'sympy.core.add.Add'>
>>> type(t2)
<class 'sage.symbolic.expression.Expression'>
>>> t1, t2
(omega + x, omega + x)
>>> e = sympy.sin(var("y")) + sage.functions.trig.cos(sympy.Symbol("x"))
>>> type(e)
<class 'sympy.core.add.Add'>
>>> e
sin(y) + cos(x)
>>> e=e._sage_()
>>> type(e)
<class 'sage.symbolic.expression.Expression'>
>>> e
cos(x) + sin(y)
>>> e = sage.functions.trig.cos(var("y")**Integer(3))**Integer(4)+var("x")**Integer(2)
>>> e = e._sympy_()
>>> e
x**2 + cos(y**3)**4
```

```
sage: a = sympy.Matrix([1, 2, 3])                                              #␣
↪needs sympy
sage: a[1]                                                                     #␣
↪needs sympy
2
```

```
>>> from sage.all import *
>>> a = sympy.Matrix([Integer(1), Integer(2), Integer(3)])                     ␣
↪                         # needs sympy
>>> a[Integer(1)]                                                              ␣
↪     # needs sympy
2
```

```
sage: sympify(1.5)                                                          #␣
↪needs sympy
1.50000000000000
sage: sympify(2)                                                            #␣
↪needs sympy
2
sage: sympify(-2)                                                           #␣
↪needs sympy
-2
```

```
>>> from sage.all import *
>>> sympify(RealNumber('1.5'))                                              ␣
↪           # needs sympy
1.50000000000000
>>> sympify(Integer(2))                                                     ␣
↪     # needs sympy
2
>>> sympify(-Integer(2))                                                    ␣
↪     # needs sympy
-2
```

2.16 Calculus Tests and Examples

Compute the Christoffel symbol.

```
sage: var('r t theta phi')
(r, t, theta, phi)
sage: m = matrix(SR, [[(1-1/r),0,0,0],[0,-(1-1/r)^(-1),0,0],[0,0,-r^2,0],[0,0,0,-r^
↪2*(sin(theta))^2]])
sage: m
[         -1/r + 1                 0                 0                 0]
[                0       1/(1/r - 1)                 0                 0]
[                0                 0              -r^2                 0]
[                0                 0                 0 -r^2*sin(theta)^2]
```

```
>>> from sage.all import *
>>> var('r t theta phi')
(r, t, theta, phi)
>>> m = matrix(SR, [[(Integer(1)-Integer(1)/r),Integer(0),Integer(0),Integer(0)],
↪[Integer(0),-(Integer(1)-Integer(1)/r)**(-Integer(1)),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),-r**Integer(2),Integer(0)],[Integer(0),Integer(0),Integer(0),
↪-r**Integer(2)*(sin(theta))**Integer(2)]])
>>> m
[         -1/r + 1                 0                 0                 0]
[                0       1/(1/r - 1)                 0                 0]
[                0                 0              -r^2                 0]
[                0                 0                 0 -r^2*sin(theta)^2]
```

```
sage: def christoffel(i, j, k, vars, g):
....:     s = 0
....:     ginv = g^(-1)
```

(continues on next page)

(continued from previous page)

```
....:     for l in range(g.nrows()):
....:         s = s + (1/2)*ginv[k,l]*(g[j,l].diff(vars[i])+g[i,l].diff(vars[j])-g[i,
↪j].diff(vars[l]))
....:     return s
```

```
>>> from sage.all import *
>>> def christoffel(i, j, k, vars, g):
...     s = Integer(0)
...     ginv = g**(-Integer(1))
...     for l in range(g.nrows()):
...         s = s + (Integer(1)/Integer(2))*ginv[k,l]*(g[j,l].diff(vars[i])+g[i,l].
↪diff(vars[j])-g[i,j].diff(vars[l]))
...     return s
```

```
sage: christoffel(3,3,2, [t,r,theta,phi], m)
-cos(theta)*sin(theta)
sage: X = christoffel(1,1,1,[t,r,theta,phi],m)
sage: X
1/2/(r^2*(1/r - 1))
sage: X.rational_simplify()
 -1/2/(r^2 - r)
```

```
>>> from sage.all import *
>>> christoffel(Integer(3),Integer(3),Integer(2), [t,r,theta,phi], m)
-cos(theta)*sin(theta)
>>> X = christoffel(Integer(1),Integer(1),Integer(1),[t,r,theta,phi],m)
>>> X
1/2/(r^2*(1/r - 1))
>>> X.rational_simplify()
 -1/2/(r^2 - r)
```

Some basic things:

```
sage: f(x,y) = x^3 + sinh(1/y)
sage: f
(x, y) |--> x^3 + sinh(1/y)
sage: f^3
(x, y) |--> (x^3 + sinh(1/y))^3
sage: (f^3).expand()
(x, y) |--> x^9 + 3*x^6*sinh(1/y) + 3*x^3*sinh(1/y)^2 + sinh(1/y)^3
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); f = symbolic_expression(x**Integer(3) + sinh(Integer(1)/y)).
↪function(x,y)
>>> f
(x, y) |--> x^3 + sinh(1/y)
>>> f**Integer(3)
(x, y) |--> (x^3 + sinh(1/y))^3
>>> (f**Integer(3)).expand()
(x, y) |--> x^9 + 3*x^6*sinh(1/y) + 3*x^3*sinh(1/y)^2 + sinh(1/y)^3
```

A polynomial over a symbolic base ring:

```
sage: R = SR['x']
sage: f = R([1/sqrt(2), 1/(4*sqrt(2))])
sage: f
1/8*sqrt(2)*x + 1/2*sqrt(2)
sage: -f
-1/8*sqrt(2)*x - 1/2*sqrt(2)
sage: (-f).degree()
1
```

```
>>> from sage.all import *
>>> R = SR['x']
>>> f = R([Integer(1)/sqrt(Integer(2)), Integer(1)/(Integer(4)*sqrt(Integer(2)))])
>>> f
1/8*sqrt(2)*x + 1/2*sqrt(2)
>>> -f
-1/8*sqrt(2)*x - 1/2*sqrt(2)
>>> (-f).degree()
1
```

A big product. Notice that simplifying simplifies the product further:

```
sage: A = exp(I*pi/7)
sage: b = A^14
sage: b
1
```

```
>>> from sage.all import *
>>> A = exp(I*pi/Integer(7))
>>> b = A**Integer(14)
>>> b
1
```

We check a statement made at the beginning of Friedlander and Joshi's book on Distributions:

```
sage: f(x) = sin(x^2)
sage: g(x) = cos(x) + x^3
sage: u = f(x+t) + g(x-t)
sage: u
-(t - x)^3 + cos(-t + x) + sin((t + x)^2)
sage: u.diff(t,2) - u.diff(x,2)
0
```

```
>>> from sage.all import *
>>> __tmp__=var("x"); f = symbolic_expression(sin(x**Integer(2))).function(x)
>>> __tmp__=var("x"); g = symbolic_expression(cos(x) + x**Integer(3)).function(x)
>>> u = f(x+t) + g(x-t)
>>> u
-(t - x)^3 + cos(-t + x) + sin((t + x)^2)
>>> u.diff(t,Integer(2)) - u.diff(x,Integer(2))
0
```

Restoring variables after they have been turned into functions:

```
sage: x = function('x')
sage: type(x)
<class 'sage.symbolic.function_factory...NewSymbolicFunction'>
sage: x(2/3)
x(2/3)
sage: restore('x')
sage: sin(x).variables()
(x,)
```

```
>>> from sage.all import *
>>> x = function('x')
>>> type(x)
<class 'sage.symbolic.function_factory...NewSymbolicFunction'>
>>> x(Integer(2)/Integer(3))
x(2/3)
>>> restore('x')
>>> sin(x).variables()
(x,)
```

MATHEMATICA: Some examples of integration and differentiation taken from some Mathematica docs:

```
sage: var('x n a')
(x, n, a)
sage: diff(x^n, x)         # the output looks funny, but is correct
n*x^(n - 1)
sage: diff(x^2 * log(x+a), x)
2*x*log(a + x) + x^2/(a + x)
sage: derivative(arctan(x), x)
1/(x^2 + 1)
sage: derivative(x^n, x, 3)
(n - 1)*(n - 2)*n*x^(n - 3)
sage: derivative( function('f')(x), x)
diff(f(x), x)
sage: diff( 2*x*f(x^2), x)
4*x^2*D[0](f)(x^2) + 2*f(x^2)
sage: integrate( 1/(x^4 - a^4), x)
-1/2*arctan(x/a)/a^3 - 1/4*log(a + x)/a^3 + 1/4*log(-a + x)/a^3
sage: expand(integrate(log(1-x^2), x))
x*log(-x^2 + 1) - 2*x + log(x + 1) - log(x - 1)
```

```
>>> from sage.all import *
>>> var('x n a')
(x, n, a)
>>> diff(x**n, x)         # the output looks funny, but is correct
n*x^(n - 1)
>>> diff(x**Integer(2) * log(x+a), x)
2*x*log(a + x) + x^2/(a + x)
>>> derivative(arctan(x), x)
1/(x^2 + 1)
>>> derivative(x**n, x, Integer(3))
(n - 1)*(n - 2)*n*x^(n - 3)
>>> derivative( function('f')(x), x)
```

(continues on next page)

(continued from previous page)

```
diff(f(x), x)
>>> diff( Integer(2)*x*f(x**Integer(2)), x)
4*x^2*D[0](f)(x^2) + 2*f(x^2)
>>> integrate( Integer(1)/(x**Integer(4) - a**Integer(4)), x)
-1/2*arctan(x/a)/a^3 - 1/4*log(a + x)/a^3 + 1/4*log(-a + x)/a^3
>>> expand(integrate(log(Integer(1)-x**Integer(2)), x))
x*log(-x^2 + 1) - 2*x + log(x + 1) - log(x - 1)
```

This is an apparent regression in Maxima 5.39.0, although the antiderivative is correct, assuming we work with (poly)logs of complex argument. More convenient form is 1/2*log(x^2)*log(-x^2 + 1) + 1/2*dilog(-x^2 + 1). See also https://sourceforge.net/p/maxima/bugs/3275/:

```
sage: integrate(log(1-x^2)/x, x)
log(-x)*log(x + 1) + log(x)*log(-x + 1) + dilog(x + 1) + dilog(-x + 1)
```

```
>>> from sage.all import *
>>> integrate(log(Integer(1)-x**Integer(2))/x, x)
log(-x)*log(x + 1) + log(x)*log(-x + 1) + dilog(x + 1) + dilog(-x + 1)
```

No problems here:

```
sage: integrate(exp(1-x^2),x)
1/2*sqrt(pi)*erf(x)*e
sage: integrate(sin(x^2),x)
1/16*sqrt(pi)*((I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sqrt(2)*x) + (I - 1)*sqrt(2)*erf((1/
↪2*I - 1/2)*sqrt(2)*x) - (I - 1)*sqrt(2)*erf(sqrt(-I)*x) + (I + 1)*sqrt(2)*erf((-1)^
↪(1/4)*x))

sage: integrate((1-x^2)^n,x)  # long time
x*hypergeometric((1/2, -n), (3/2,), x^2*exp_polar(2*I*pi))
sage: integrate(x^x,x)
integrate(x^x, x)
sage: integrate(1/(x^3+1),x)
1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x - 1)) - 1/6*log(x^2 - x + 1) + 1/3*log(x + 1)
sage: integrate(1/(x^3+1), x, 0, 1)
1/9*sqrt(3)*pi + 1/3*log(2)
```

```
>>> from sage.all import *
>>> integrate(exp(Integer(1)-x**Integer(2)),x)
1/2*sqrt(pi)*erf(x)*e
>>> integrate(sin(x**Integer(2)),x)
1/16*sqrt(pi)*((I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sqrt(2)*x) + (I - 1)*sqrt(2)*erf((1/
↪2*I - 1/2)*sqrt(2)*x) - (I - 1)*sqrt(2)*erf(sqrt(-I)*x) + (I + 1)*sqrt(2)*erf((-1)^
↪(1/4)*x))

>>> integrate((Integer(1)-x**Integer(2))**n,x)  # long time
x*hypergeometric((1/2, -n), (3/2,), x^2*exp_polar(2*I*pi))
>>> integrate(x**x,x)
integrate(x^x, x)
>>> integrate(Integer(1)/(x**Integer(3)+Integer(1)),x)
1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x - 1)) - 1/6*log(x^2 - x + 1) + 1/3*log(x + 1)
>>> integrate(Integer(1)/(x**Integer(3)+Integer(1)), x, Integer(0), Integer(1))
```

(continues on next page)

(continued from previous page)

```
1/9*sqrt(3)*pi + 1/3*log(2)
```

```
sage: forget()
sage: c = var('c')
sage: assume(c > 0)
sage: integrate(exp(-c*x^2), x, -oo, oo)
sqrt(pi)/sqrt(c)
sage: forget()
```

```
>>> from sage.all import *
>>> forget()
>>> c = var('c')
>>> assume(c > Integer(0))
>>> integrate(exp(-c*x**Integer(2)), x, -oo, oo)
sqrt(pi)/sqrt(c)
>>> forget()
```

Other examples that now (Issue #27958) work:

```
sage: integrate(log(x)*exp(-x^2), x)  # long time
1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)

sage: integrate(log(1+sqrt(1+4*x)/2)/x, x, 0, 1)
Traceback (most recent call last):
...
ValueError: Integral is divergent.
```

```
>>> from sage.all import *
>>> integrate(log(x)*exp(-x**Integer(2)), x)  # long time
1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)

>>> integrate(log(Integer(1)+sqrt(Integer(1)+Integer(4)*x)/Integer(2))/x, x,␣
↪Integer(0), Integer(1))
Traceback (most recent call last):
...
ValueError: Integral is divergent.
```

The following is an example of integral that Mathematica can do, but Sage currently cannot do:

```
sage: integrate(ceil(x^2 + floor(x)), x, 0, 5, algorithm='maxima')
integrate(ceil(x^2) + floor(x), x, 0, 5)
```

```
>>> from sage.all import *
>>> integrate(ceil(x**Integer(2) + floor(x)), x, Integer(0), Integer(5), algorithm=
↪'maxima')
integrate(ceil(x^2) + floor(x), x, 0, 5)
```

MAPLE: The basic differentiation and integration examples in the Maple documentation:

```
sage: diff(sin(x), x)
cos(x)
```

(continues on next page)

(continued from previous page)

```
sage: diff(sin(x), y)
0
sage: diff(sin(x), x, 3)
-cos(x)
sage: diff(x*sin(cos(x)), x)
-x*cos(cos(x))*sin(x) + sin(cos(x))
sage: diff(tan(x), x)
tan(x)^2 + 1
sage: f = function('f'); f
f
sage: diff(f(x), x)
diff(f(x), x)
sage: diff(f(x,y), x, y)
diff(f(x, y), x, y)
sage: diff(f(x,y), x, y) - diff(f(x,y), y, x)
0
sage: g = function('g')
sage: var('x y z')
(x, y, z)
sage: diff(g(x,y,z), x,z,z)
diff(g(x, y, z), x, z, z)
sage: integrate(sin(x), x)
-cos(x)
sage: integrate(sin(x), x, 0, pi)
2
```

```
>>> from sage.all import *
>>> diff(sin(x), x)
cos(x)
>>> diff(sin(x), y)
0
>>> diff(sin(x), x, Integer(3))
-cos(x)
>>> diff(x*sin(cos(x)), x)
-x*cos(cos(x))*sin(x) + sin(cos(x))
>>> diff(tan(x), x)
tan(x)^2 + 1
>>> f = function('f'); f
f
>>> diff(f(x), x)
diff(f(x), x)
>>> diff(f(x,y), x, y)
diff(f(x, y), x, y)
>>> diff(f(x,y), x, y) - diff(f(x,y), y, x)
0
>>> g = function('g')
>>> var('x y z')
(x, y, z)
>>> diff(g(x,y,z), x,z,z)
diff(g(x, y, z), x, z, z)
>>> integrate(sin(x), x)
-cos(x)
```

(continues on next page)

(continued from previous page)

```
>>> integrate(sin(x), x, Integer(0), pi)
2
```

```
sage: var('a b')
(a, b)
sage: integrate(sin(x), x, a, b)
cos(a) - cos(b)
```

```
>>> from sage.all import *
>>> var('a b')
(a, b)
>>> integrate(sin(x), x, a, b)
cos(a) - cos(b)
```

```
sage: integrate( x/(x^3-1), x)
1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x + 1)) - 1/6*log(x^2 + x + 1) + 1/3*log(x - 1)
sage: integrate(exp(-x^2), x)
1/2*sqrt(pi)*erf(x)
sage: integrate(exp(-x^2)*log(x), x)  # long time
1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)
sage: f = exp(-x^2)*log(x)
sage: f.nintegral(x, 0, 999)
(-0.87005772672831..., 7.5584...e-10, 567, 0)
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)     # long time  # todo: maple can␣
↪do this
integrate(1/(sqrt(2*t^2 + 1)*sqrt(t^2 - 2)), t, 2, 3)
sage: integral(integral(x*y^2, x, 0, y), y, -2, 2)
32/5
```

```
>>> from sage.all import *
>>> integrate( x/(x**Integer(3)-Integer(1)), x)
1/3*sqrt(3)*arctan(1/3*sqrt(3)*(2*x + 1)) - 1/6*log(x^2 + x + 1) + 1/3*log(x - 1)
>>> integrate(exp(-x**Integer(2)), x)
1/2*sqrt(pi)*erf(x)
>>> integrate(exp(-x**Integer(2))*log(x), x)  # long time
1/2*sqrt(pi)*erf(x)*log(x) - x*hypergeometric((1/2, 1/2), (3/2, 3/2), -x^2)
>>> f = exp(-x**Integer(2))*log(x)
>>> f.nintegral(x, Integer(0), Integer(999))
(-0.87005772672831..., 7.5584...e-10, 567, 0)
>>> integral(Integer(1)/sqrt(Integer(2)*t**Integer(4) - Integer(3)*t**Integer(2) -␣
↪Integer(2)), t, Integer(2), Integer(3))     # long time  # todo: maple can do this
integrate(1/(sqrt(2*t^2 + 1)*sqrt(t^2 - 2)), t, 2, 3)
>>> integral(integral(x*y**Integer(2), x, Integer(0), y), y, -Integer(2), Integer(2))
32/5
```

We verify several standard differentiation rules:

```
sage: function('f, g')
(f, g)
sage: diff(f(t)*g(t),t)
g(t)*diff(f(t), t) + f(t)*diff(g(t), t)
```

(continues on next page)

(continued from previous page)

```
sage: diff(f(t)/g(t), t)
diff(f(t), t)/g(t) - f(t)*diff(g(t), t)/g(t)^2
sage: diff(f(t) + g(t), t)
diff(f(t), t) + diff(g(t), t)
sage: diff(c*f(t), t)
c*diff(f(t), t)
```

```
>>> from sage.all import *
>>> function('f, g')
(f, g)
>>> diff(f(t)*g(t),t)
g(t)*diff(f(t), t) + f(t)*diff(g(t), t)
>>> diff(f(t)/g(t), t)
diff(f(t), t)/g(t) - f(t)*diff(g(t), t)/g(t)^2
>>> diff(f(t) + g(t), t)
diff(f(t), t) + diff(g(t), t)
>>> diff(c*f(t), t)
c*diff(f(t), t)
```

2.17 Conversion of symbolic expressions to other types

This module provides routines for converting new symbolic expressions to other types. Primarily, it provides a class `Converter` which will walk the expression tree and make calls to methods overridden by subclasses.

class `sage.symbolic.expression_conversions.`**`Converter`**`(`*use_fake_div=False*`)`

Bases: `object`

If use_fake_div is set to True, then the converter will try to replace expressions whose operator is operator.mul with the corresponding expression whose operator is operator.truediv.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: c = Converter(use_fake_div=True)
sage: c.use_fake_div
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import Converter
>>> c = Converter(use_fake_div=True)
>>> c.use_fake_div
True
```

`arithmetic``(`*ex*, *operator*`)`

The input to this method is a symbolic expression and the infix operator corresponding to that expression. Typically, one will convert all of the arguments and then perform the operation afterward.

`composition``(`*ex*, *operator*`)`

The input to this method is a symbolic expression and its operator. This method will get called when you have a symbolic function application.

`derivative``(`*ex*, *operator*`)`

The input to this method is a symbolic expression which corresponds to a relation.

get_fake_div(*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: c = Converter(use_fake_div=True)
sage: c.get_fake_div(sin(x)/x)
FakeExpression([sin(x), x], <built-in function truediv>)
sage: c.get_fake_div(-1*sin(x))
FakeExpression([sin(x)], <built-in function neg>)
sage: c.get_fake_div(-x)
FakeExpression([x], <built-in function neg>)
sage: c.get_fake_div((2*x^3+2*x-1)/((x-2)*(x+1)))
FakeExpression([2*x^3 + 2*x - 1, FakeExpression([x + 1, x - 2], <built-in␣
↪function mul>)], <built-in function truediv>)
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import Converter
>>> c = Converter(use_fake_div=True)
>>> c.get_fake_div(sin(x)/x)
FakeExpression([sin(x), x], <built-in function truediv>)
>>> c.get_fake_div(-Integer(1)*sin(x))
FakeExpression([sin(x)], <built-in function neg>)
>>> c.get_fake_div(-x)
FakeExpression([x], <built-in function neg>)
>>> c.get_fake_div((Integer(2)*x**Integer(3)+Integer(2)*x-Integer(1))/((x-
↪Integer(2))*(x+Integer(1))))
FakeExpression([2*x^3 + 2*x - 1, FakeExpression([x + 1, x - 2], <built-in␣
↪function mul>)], <built-in function truediv>)
```

Check if Issue #8056 is fixed, i.e., if numerator is 1.:

```
sage: c.get_fake_div(1/pi/x)
FakeExpression([1, FakeExpression([pi, x], <built-in function mul>)], <built-
↪in function truediv>)
```

```
>>> from sage.all import *
>>> c.get_fake_div(Integer(1)/pi/x)
FakeExpression([1, FakeExpression([pi, x], <built-in function mul>)], <built-
↪in function truediv>)
```

pyobject(*ex*, *obj*)

The input to this method is the result of calling `pyobject()` on a symbolic expression.

> **Note**
>
> Note that if a constant such as `pi` is encountered in the expression tree, its corresponding pyobject which is an instance of `sage.symbolic.constants.Pi` will be passed into this method. One cannot do arithmetic using such an object.

relation(*ex*, *operator*)

The input to this method is a symbolic expression which corresponds to a relation.

symbol(*ex*)

The input to this method is a symbolic expression which corresponds to a single variable. For example, this method could be used to return a generator for a polynomial ring.

class sage.symbolic.expression_conversions.**DeMoivre**(*ex*, *force=False*)

Bases: *ExpressionTreeWalker*

A class that walks a symbolic expression tree and replaces occurences of complex exponentials (optionally, all exponentials) by their respective trigonometric expressions.

INPUT:

- force – boolean (default: False); replace $\exp(x)$ with $\cosh(x) + \sinh(x)$

EXAMPLES:

```
sage: a, b = SR.var("a, b")
sage: from sage.symbolic.expression_conversions import DeMoivre
sage: d = DeMoivre(e^a)
sage: d(e^(a+I*b))
(cos(b) + I*sin(b))*e^a
```

```
>>> from sage.all import *
>>> a, b = SR.var("a, b")
>>> from sage.symbolic.expression_conversions import DeMoivre
>>> d = DeMoivre(e**a)
>>> d(e**(a+I*b))
(cos(b) + I*sin(b))*e^a
```

composition(*ex*, *op*)

Return the composition of self with ex by op.

EXAMPLES:

```
sage: x, a, b = SR.var('x, a, b')
sage: from sage.symbolic.expression_conversions import DeMoivre
sage: p = exp(x)
sage: s = DeMoivre(p)
sage: q = exp(a+I*b)
sage: s.composition(q, q.operator())
(cos(b) + I*sin(b))*e^a
```

```
>>> from sage.all import *
>>> x, a, b = SR.var('x, a, b')
>>> from sage.symbolic.expression_conversions import DeMoivre
>>> p = exp(x)
>>> s = DeMoivre(p)
>>> q = exp(a+I*b)
>>> s.composition(q, q.operator())
(cos(b) + I*sin(b))*e^a
```

class sage.symbolic.expression_conversions.**Exponentialize**(*ex*)

Bases: *ExpressionTreeWalker*

A class that walks a symbolic expression tree and replace circular and hyperbolic functions by their respective exponential expressions.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import Exponentialize
sage: d = Exponentialize(sin(x))
sage: d(sin(x))
-1/2*I*e^(I*x) + 1/2*I*e^(-I*x)
sage: d(cosh(x))
1/2*e^(-x) + 1/2*e^x
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import Exponentialize
>>> d = Exponentialize(sin(x))
>>> d(sin(x))
-1/2*I*e^(I*x) + 1/2*I*e^(-I*x)
>>> d(cosh(x))
1/2*e^(-x) + 1/2*e^x
```

CircDict = {sinh: x |--> -1/2*e^(-x) + 1/2*e^x, cosh: x |--> 1/2*e^(-x) + 1/2*e^x, tanh: x |--> -(e^(-x) - e^x)/(e^(-x) + e^x), coth: x |--> -(e^(-x) + e^x)/(e^(-x) - e^x), sech: x |--> 2/(e^(-x) + e^x), csch: x |--> -2/(e^(-x) - e^x), sin: x |--> -1/2*I*e^(I*x) + 1/2*I*e^(-I*x), cos: x |--> 1/2*e^(I*x) + 1/2*e^(-I*x), tan: x |--> (-I*e^(I*x) + I*e^(-I*x))/(e^(I*x) + e^(-I*x)), cot: x |--> (I*e^(I*x) + I*e^(-I*x))/(e^(I*x) - e^(-I*x)), sec: x |--> 2/(e^(I*x) + e^(-I*x)), csc: x |--> 2*I/(e^(I*x) - e^(-I*x))}

Circs = [sin, cos, sec, csc, tan, cot, sinh, cosh, sech, csch, tanh, coth]

I = I

Integer

alias of `Integer`

SR = Symbolic Ring

composition(*ex*, *op*)

Return the composition of `self` with `ex` by `op`.

EXAMPLES:

```
sage: x = SR.var("x")
sage: from sage.symbolic.expression_conversions import Exponentialize
sage: p = x
sage: s = Exponentialize(p)
sage: q = sin(x)
sage: s.composition(q, q.operator())
-1/2*I*e^(I*x) + 1/2*I*e^(-I*x)
```

```
>>> from sage.all import *
>>> x = SR.var("x")
>>> from sage.symbolic.expression_conversions import Exponentialize
>>> p = x
>>> s = Exponentialize(p)
>>> q = sin(x)
>>> s.composition(q, q.operator())
-1/2*I*e^(I*x) + 1/2*I*e^(-I*x)
```

cos = cos

cosh = cosh

cot = cot

coth = coth

csc = csc

csch = csch

e = e

function(*s*, ***kwds*)

Create a formal symbolic function with the name *s*.

INPUT:

- `nargs=0` – number of arguments the function accepts, defaults to variable number of arguments, or 0
- `latex_name` – name used when printing in latex mode
- `conversions` – dictionary specifying names of this function in other systems, this is used by the interfaces internally during conversion
- `eval_func` – method used for automatic evaluation
- `evalf_func` – method used for numeric evaluation
- `evalf_params_first` – boolean to indicate if parameters should be evaluated numerically before calling the custom evalf function
- `conjugate_func` – method used for complex conjugation
- `real_part_func` – method used when taking real parts
- `imag_part_func` – method used when taking imaginary parts
- `derivative_func` – method to be used for (partial) derivation This method should take a keyword argument deriv_param specifying the index of the argument to differentiate w.r.t
- `tderivative_func` – method to be used for derivatives
- `power_func` – method used when taking powers This method should take a keyword argument power_param specifying the exponent
- `series_func` – method used for series expansion This method should expect keyword arguments - `order` – order for the expansion to be computed - `var` – variable to expand w.r.t. - `at` – expand at this value
- `print_func` – method for custom printing
- `print_latex_func` – method for custom printing in latex mode

Note that custom methods must be instance methods, i.e., expect the instance of the symbolic function as the first argument.

Note

The new function is both returned and automatically injected into the global namespace. If you use this function in library code, it is better to use sage.symbolic.function_factory.function, since it will not touch the global namespace.

EXAMPLES:

We create a formal function called supersin

```
sage: function('supersin')
supersin
```

```
>>> from sage.all import *
>>> function('supersin')
supersin
```

We can immediately use supersin in symbolic expressions:

```
sage: y, z, A = var('y z A')
sage: supersin(y+z) + A^3
A^3 + supersin(y + z)
```

```
>>> from sage.all import *
>>> y, z, A = var('y z A')
>>> supersin(y+z) + A**Integer(3)
A^3 + supersin(y + z)
```

We can define other functions in terms of supersin:

```
sage: g(x,y) = supersin(x)^2 + sin(y/2)
sage: g
(x, y) |--> supersin(x)^2 + sin(1/2*y)
sage: g.diff(y)
(x, y) |--> 1/2*cos(1/2*y)
sage: k = g.diff(x); k
(x, y) |--> 2*supersin(x)*diff(supersin(x), x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); g = symbolic_expression(supersin(x)**Integer(2) +␣
↪sin(y/Integer(2))).function(x,y)
>>> g
(x, y) |--> supersin(x)^2 + sin(1/2*y)
>>> g.diff(y)
(x, y) |--> 1/2*cos(1/2*y)
>>> k = g.diff(x); k
(x, y) |--> 2*supersin(x)*diff(supersin(x), x)
```

We create a formal function of one variable, write down an expression that involves first and second derivatives, and extract off coefficients:

```
sage: r, kappa = var('r,kappa')
sage: psi = function('psi', nargs=1)(r); psi
psi(r)
sage: g = 1/r^2*(2*r*psi.derivative(r,1) + r^2*psi.derivative(r,2)); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
sage: g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
sage: g.coefficient(psi.derivative(r,2))
1
```

(continues on next page)

(continued from previous page)

```
sage: g.coefficient(psi.derivative(r,1))
2/r
```

```
>>> from sage.all import *
>>> r, kappa = var('r,kappa')
>>> psi = function('psi', nargs=Integer(1))(r); psi
psi(r)
>>> g = Integer(1)/r**Integer(2)*(Integer(2)*r*psi.derivative(r,Integer(1)) +
→r**Integer(2)*psi.derivative(r,Integer(2))); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
>>> g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
>>> g.coefficient(psi.derivative(r,Integer(2)))
1
>>> g.coefficient(psi.derivative(r,Integer(1)))
2/r
```

Custom typesetting of symbolic functions in LaTeX, either using latex_name keyword:

```
sage: function('riemann', latex_name='\\mathcal{R}')
riemann
sage: latex(riemann(x))
\mathcal{R}\left(x\right)
```

```
>>> from sage.all import *
>>> function('riemann', latex_name='\\mathcal{R}')
riemann
>>> latex(riemann(x))
\mathcal{R}\left(x\right)
```

or passing a custom callable function that returns a latex expression:

```
sage: mu,nu = var('mu,nu')
sage: def my_latex_print(self, *args): return "\\psi_{%s}"%(', '.
→join(map(latex, args)))
sage: function('psi', print_latex_func=my_latex_print)
psi
sage: latex(psi(mu,nu))
\psi_{\mu, \nu}
```

```
>>> from sage.all import *
>>> mu,nu = var('mu,nu')
>>> def my_latex_print(self, *args): return "\\psi_{%s}"%(', '.join(map(latex,
→ args)))
>>> function('psi', print_latex_func=my_latex_print)
psi
>>> latex(psi(mu,nu))
\psi_{\mu, \nu}
```

Defining custom methods for automatic or numeric evaluation, derivation, conjugation, etc. is supported:

```
sage: def ev(self, x): return 2*x
sage: foo = function("foo", nargs=1, eval_func=ev)
sage: foo(x)
2*x
sage: foo = function("foo", nargs=1, eval_func=lambda self, x: 5)
sage: foo(x)
5
sage: def ef(self, x): pass
sage: bar = function("bar", nargs=1, eval_func=ef)
sage: bar(x)
bar(x)

sage: def evalf_f(self, x, parent=None, algorithm=None): return 6
sage: foo = function("foo", nargs=1, evalf_func=evalf_f)
sage: foo(x)
foo(x)
sage: foo(x).n()
6

sage: foo = function("foo", nargs=1, conjugate_func=ev)
sage: foo(x).conjugate()
2*x

sage: def deriv(self, *args, **kwds): print("{} {}".format(args, kwds));␣
↪return args[kwds['diff_param']]^2
sage: foo = function("foo", nargs=2, derivative_func=deriv)
sage: foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2

sage: def pow(self, x, power_param=None): print("{} {}".format(x, power_
↪param)); return x*power_param
sage: foo = function("foo", nargs=1, power_func=pow)
sage: foo(y)^(x+y)
y x + y
(x + y)*y

sage: from pprint import pformat
sage: def expand(self, *args, **kwds):
....:     print("{} {}".format(args, pformat(kwds)))
....:     return sum(args[0]^i for i in range(kwds['order']))
sage: foo = function("foo", nargs=1, series_func=expand)
sage: foo(y).series(y, 5)
(y,) {'at': 0, 'options': 0, 'order': 5, 'var': y}
y^4 + y^3 + y^2 + y + 1

sage: def my_print(self, *args):
....:     return "my args are: " + ', '.join(map(repr, args))
sage: foo = function('t', nargs=2, print_func=my_print)
sage: foo(x,y^z)
my args are: x, y^z

sage: latex(foo(x,y^z))
```

(continues on next page)

(continued from previous page)

```
t\left(x, y^{z}\right)
sage: foo = function('t', nargs=2, print_latex_func=my_print)
sage: foo(x,y^z)
t(x, y^z)
sage: latex(foo(x,y^z))
my args are: x, y^z
sage: foo = function('t', nargs=2, latex_name='foo')
sage: latex(foo(x,y^z))
foo\left(x, y^{z}\right)
```

```
>>> from sage.all import *
>>> def ev(self, x): return Integer(2)*x
>>> foo = function("foo", nargs=Integer(1), eval_func=ev)
>>> foo(x)
2*x
>>> foo = function("foo", nargs=Integer(1), eval_func=lambda self, x:␣
↪Integer(5))
>>> foo(x)
5
>>> def ef(self, x): pass
>>> bar = function("bar", nargs=Integer(1), eval_func=ef)
>>> bar(x)
bar(x)

>>> def evalf_f(self, x, parent=None, algorithm=None): return Integer(6)
>>> foo = function("foo", nargs=Integer(1), evalf_func=evalf_f)
>>> foo(x)
foo(x)
>>> foo(x).n()
6

>>> foo = function("foo", nargs=Integer(1), conjugate_func=ev)
>>> foo(x).conjugate()
2*x

>>> def deriv(self, *args, **kwds): print("{} {}".format(args, kwds)); return␣
↪args[kwds['diff_param']]**Integer(2)
>>> foo = function("foo", nargs=Integer(2), derivative_func=deriv)
>>> foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2

>>> def pow(self, x, power_param=None): print("{} {}".format(x, power_param));
↪ return x*power_param
>>> foo = function("foo", nargs=Integer(1), power_func=pow)
>>> foo(y)**(x+y)
y x + y
(x + y)*y

>>> from pprint import pformat
>>> def expand(self, *args, **kwds):
...     print("{} {}".format(args, pformat(kwds)))
```

(continues on next page)

(continued from previous page)

```
...     return sum(args[Integer(0)]**i for i in range(kwds['order']))
>>> foo = function("foo", nargs=Integer(1), series_func=expand)
>>> foo(y).series(y, Integer(5))
(y,) {'at': 0, 'options': 0, 'order': 5, 'var': y}
y^4 + y^3 + y^2 + y + 1

>>> def my_print(self, *args):
...     return "my args are: " + ', '.join(map(repr, args))
>>> foo = function('t', nargs=Integer(2), print_func=my_print)
>>> foo(x,y**z)
my args are: x, y^z

>>> latex(foo(x,y**z))
t\left(x, y^{z}\right)
>>> foo = function('t', nargs=Integer(2), print_latex_func=my_print)
>>> foo(x,y**z)
t(x, y^z)
>>> latex(foo(x,y**z))
my args are: x, y^z
>>> foo = function('t', nargs=Integer(2), latex_name='foo')
>>> latex(foo(x,y**z))
foo\left(x, y^{z}\right)
```

Chain rule:

```
sage: def print_args(self, *args, **kwds): print("args: {}".format(args));␣
↪print("kwds: {}".format(kwds)); return args[0]
sage: foo = function('t', nargs=2, tderivative_func=print_args)
sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': x}
x
sage: foo = function('t', nargs=2, derivative_func=print_args)
sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
args: (x, x)
kwds: {'diff_param': 1}
2*x
```

```
>>> from sage.all import *
>>> def print_args(self, *args, **kwds): print("args: {}".format(args));␣
↪print("kwds: {}".format(kwds)); return args[Integer(0)]
>>> foo = function('t', nargs=Integer(2), tderivative_func=print_args)
>>> foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': x}
x
>>> foo = function('t', nargs=Integer(2), derivative_func=print_args)
>>> foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
```

(continues on next page)

(continued from previous page)

```
args: (x, x)
kwds: {'diff_param': 1}
2*x
```

Since Sage 4.0, basic arithmetic with unevaluated functions is no longer supported:

```
sage: x = var('x')
sage: f = function('f')
sage: 2*f
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring' and '<class
↪'sage.symbolic.function_factory...NewSymbolicFunction'>'
```

```
>>> from sage.all import *
>>> x = var('x')
>>> f = function('f')
>>> Integer(2)*f
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring' and '<class
↪'sage.symbolic.function_factory...NewSymbolicFunction'>'
```

You now need to evaluate the function in order to do the arithmetic:

```
sage: 2*f(x)
2*f(x)
```

```
>>> from sage.all import *
>>> Integer(2)*f(x)
2*f(x)
```

Since Sage 4.0, you need to use `substitute_function()` to replace all occurrences of a function with another:

```
sage: var('a, b')
(a, b)
sage: cr = function('cr')
sage: f = cr(a)
sage: g = f.diff(a).integral(b)
sage: g
b*diff(cr(a), a)
sage: g.substitute_function(cr, cos)
-b*sin(a)

sage: g.substitute_function(cr, (sin(x) + cos(x)).function(x))
b*(cos(a) - sin(a))
```

```
>>> from sage.all import *
>>> var('a, b')
(a, b)
>>> cr = function('cr')
```

(continues on next page)

(continued from previous page)

```
>>> f = cr(a)
>>> g = f.diff(a).integral(b)
>>> g
b*diff(cr(a), a)
>>> g.substitute_function(cr, cos)
-b*sin(a)

>>> g.substitute_function(cr, (sin(x) + cos(x)).function(x))
b*(cos(a) - sin(a))
```

half = 1/2

sec = sec

sech = sech

sin = sin

sinh = sinh

tan = tan

tanh = tanh

two = 2

x = x

class sage.symbolic.expression_conversions.**ExpressionTreeWalker**(*ex*)

Bases: *Converter*

A class that walks the tree. Mainly for subclassing.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
sage: from sage.symbolic.random_tests import random_expr
sage: ex = sin(atan(0,hold=True)+hypergeometric((1,),(1,),x))
sage: s = ExpressionTreeWalker(ex)
sage: bool(s() == ex)
True
sage: set_random_seed(0)  # random_expr is unstable
sage: foo = random_expr(20, nvars=2)
sage: s = ExpressionTreeWalker(foo)
sage: bool(s() == foo)
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import ExpressionTreeWalker
>>> from sage.symbolic.random_tests import random_expr
>>> ex = sin(atan(Integer(0),hold=True)+hypergeometric((Integer(1),),(Integer(1),
↪),x))
>>> s = ExpressionTreeWalker(ex)
>>> bool(s() == ex)
True
```

(continues on next page)

(continued from previous page)

```
>>> set_random_seed(Integer(0))  # random_expr is unstable
>>> foo = random_expr(Integer(20), nvars=Integer(2))
>>> s = ExpressionTreeWalker(foo)
>>> bool(s() == foo)
True
```

arithmetic(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
sage: foo = function('foo')
sage: f = x*foo(x) + pi/foo(x)
sage: s = ExpressionTreeWalker(f)
sage: bool(s.arithmetic(f, f.operator()) == f)
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import ExpressionTreeWalker
>>> foo = function('foo')
>>> f = x*foo(x) + pi/foo(x)
>>> s = ExpressionTreeWalker(f)
>>> bool(s.arithmetic(f, f.operator()) == f)
True
```

composition(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
sage: foo = function('foo')
sage: f = foo(atan2(0, 0, hold=True))
sage: s = ExpressionTreeWalker(f)
sage: bool(s.composition(f, f.operator()) == f)
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import ExpressionTreeWalker
>>> foo = function('foo')
>>> f = foo(atan2(Integer(0), Integer(0), hold=True))
>>> s = ExpressionTreeWalker(f)
>>> bool(s.composition(f, f.operator()) == f)
True
```

derivative(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
sage: foo = function('foo')
sage: f = foo(x).diff(x)
sage: s = ExpressionTreeWalker(f)
sage: bool(s.derivative(f, f.operator()) == f)
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import ExpressionTreeWalker
>>> foo = function('foo')
>>> f = foo(x).diff(x)
>>> s = ExpressionTreeWalker(f)
>>> bool(s.derivative(f, f.operator()) == f)
True
```

pyobject (*ex*, *obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
sage: f = SR(2)
sage: s = ExpressionTreeWalker(f)
sage: bool(s.pyobject(f, f.pyobject()) == f.pyobject())
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import ExpressionTreeWalker
>>> f = SR(Integer(2))
>>> s = ExpressionTreeWalker(f)
>>> bool(s.pyobject(f, f.pyobject()) == f.pyobject())
True
```

relation (*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
sage: foo = function('foo')
sage: eq = foo(x) == x
sage: s = ExpressionTreeWalker(eq)
sage: s.relation(eq, eq.operator()) == eq
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import ExpressionTreeWalker
>>> foo = function('foo')
>>> eq = foo(x) == x
>>> s = ExpressionTreeWalker(eq)
>>> s.relation(eq, eq.operator()) == eq
True
```

symbol (*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
sage: s = ExpressionTreeWalker(x)
sage: bool(s.symbol(x) == x)
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import ExpressionTreeWalker
```

(continues on next page)

(continued from previous page)

```
>>> s = ExpressionTreeWalker(x)
>>> bool(s.symbol(x) == x)
True
```

tuple(*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import ExpressionTreeWalker
sage: foo = function('foo')
sage: f = hypergeometric((1,2,3,),(x,),x)
sage: s = ExpressionTreeWalker(f)
sage: bool(s() == f)
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import ExpressionTreeWalker
>>> foo = function('foo')
>>> f = hypergeometric((Integer(1),Integer(2),Integer(3),),(x,),x)
>>> s = ExpressionTreeWalker(f)
>>> bool(s() == f)
True
```

class sage.symbolic.expression_conversions.**FakeExpression**(*operands*, *operator*)

Bases: object

Pynac represents x/y as xy^{-1}. Often, tree-walkers would prefer to see divisions instead of multiplications and negative exponents. To allow for this (since Pynac internally doesn't have division at all), there is a possibility to pass use_fake_div=True; this will rewrite an Expression into a mixture of Expression and FakeExpression nodes, where the FakeExpression nodes are used to represent divisions. These nodes are intended to act sufficiently like Expression nodes that tree-walkers won't care about the difference.

operands()

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import FakeExpression
sage: import operator; x,y = var('x,y')
sage: f = FakeExpression([x, y], operator.truediv)
sage: f.operands()
[x, y]
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import FakeExpression
>>> import operator; x,y = var('x,y')
>>> f = FakeExpression([x, y], operator.truediv)
>>> f.operands()
[x, y]
```

operator()

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import FakeExpression
sage: import operator; x,y = var('x,y')
```

(continues on next page)

(continued from previous page)

```
sage: f = FakeExpression([x, y], operator.truediv)
sage: f.operator()
<built-in function truediv>
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import FakeExpression
>>> import operator; x,y = var('x,y')
>>> f = FakeExpression([x, y], operator.truediv)
>>> f.operator()
<built-in function truediv>
```

pyobject()

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import FakeExpression
sage: import operator; x,y = var('x,y')
sage: f = FakeExpression([x, y], operator.truediv)
sage: f.pyobject()
Traceback (most recent call last):
...
TypeError: self must be a numeric expression
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import FakeExpression
>>> import operator; x,y = var('x,y')
>>> f = FakeExpression([x, y], operator.truediv)
>>> f.pyobject()
Traceback (most recent call last):
...
TypeError: self must be a numeric expression
```

class sage.symbolic.expression_conversions.**FastCallableConverter**(*ex*, *etb*)

Bases: *Converter*

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import FastCallableConverter
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x'])
sage: f = FastCallableConverter(x+2, etb)
sage: f.ex
x + 2
sage: f.etb
<sage.ext.fast_callable.ExpressionTreeBuilder object at 0x...>
sage: f.use_fake_div
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import FastCallableConverter
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=['x'])
>>> f = FastCallableConverter(x+Integer(2), etb)
```

(continues on next page)

(continued from previous page)

```
>>> f.ex
x + 2
>>> f.etb
<sage.ext.fast_callable.ExpressionTreeBuilder object at 0x...>
>>> f.use_fake_div
True
```

arithmetic(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x','y'])
sage: var('x,y')
(x, y)
sage: (x+y)._fast_callable_(etb)
add(v_0, v_1)
sage: (-x)._fast_callable_(etb)
neg(v_0)
sage: (x+y+x^2)._fast_callable_(etb)
add(add(ipow(v_0, 2), v_0), v_1)
```

```
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=['x','y'])
>>> var('x,y')
(x, y)
>>> (x+y)._fast_callable_(etb)
add(v_0, v_1)
>>> (-x)._fast_callable_(etb)
neg(v_0)
>>> (x+y+x**Integer(2))._fast_callable_(etb)
add(add(ipow(v_0, 2), v_0), v_1)
```

composition(*ex*, *function*)

Given an ExpressionTreeBuilder, return an Expression representing this value.

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x','y'])
sage: x,y = var('x,y')
sage: sin(sqrt(x+y))._fast_callable_(etb)
sin(sqrt(add(v_0, v_1)))
sage: arctan2(x,y)._fast_callable_(etb)
{arctan2}(v_0, v_1)
```

```
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=['x','y'])
>>> x,y = var('x,y')
>>> sin(sqrt(x+y))._fast_callable_(etb)
sin(sqrt(add(v_0, v_1)))
```

(continues on next page)

(continued from previous page)

```
>>> arctan2(x,y)._fast_callable_(etb)
{arctan2}(v_0, v_1)
```

pyobject (*ex*, *obj*)

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x'])
sage: pi._fast_callable_(etb)
pi
sage: etb = ExpressionTreeBuilder(vars=['x'], domain=RDF)
sage: pi._fast_callable_(etb)
3.141592653589793
```

```
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=['x'])
>>> pi._fast_callable_(etb)
pi
>>> etb = ExpressionTreeBuilder(vars=['x'], domain=RDF)
>>> pi._fast_callable_(etb)
3.141592653589793
```

relation (*ex*, *operator*)

EXAMPLES:

```
sage: ff = fast_callable(x == 2, vars=['x'])
sage: ff(2)
0
sage: ff(4)
2
sage: ff = fast_callable(x < 2, vars=['x'])
Traceback (most recent call last):
...
NotImplementedError
```

```
>>> from sage.all import *
>>> ff = fast_callable(x == Integer(2), vars=['x'])
>>> ff(Integer(2))
0
>>> ff(Integer(4))
2
>>> ff = fast_callable(x < Integer(2), vars=['x'])
Traceback (most recent call last):
...
NotImplementedError
```

symbol (*ex*)

Given an ExpressionTreeBuilder, return an Expression representing this value.

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x','y'])
sage: x, y, z = var('x,y,z')
sage: x._fast_callable_(etb)
v_0
sage: y._fast_callable_(etb)
v_1
sage: z._fast_callable_(etb)
Traceback (most recent call last):
...
ValueError: Variable 'z' not found...
```

```
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=['x','y'])
>>> x, y, z = var('x,y,z')
>>> x._fast_callable_(etb)
v_0
>>> y._fast_callable_(etb)
v_1
>>> z._fast_callable_(etb)
Traceback (most recent call last):
...
ValueError: Variable 'z' not found...
```

tuple(*ex*)

Given a symbolic tuple, return its elements as a Python list.

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x'])
sage: SR._force_pyobject((2, 3, x^2))._fast_callable_(etb)
[2, 3, x^2]
```

```
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=['x'])
>>> SR._force_pyobject((Integer(2), Integer(3), x**Integer(2)))._fast_
↪callable_(etb)
[2, 3, x^2]
```

class sage.symbolic.expression_conversions.**FriCASConverter**

Bases: *InterfaceInit*

Convert any expression to FriCAS.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: f = exp(x^2) - arcsin(pi+x)/y
sage: f._fricas_()                                                    #␣
↪optional - fricas
```

(continues on next page)

(continued from previous page)

```
     2
    x
y %e    - asin(x + %pi)
----------------------
           y
```

```
>>> from sage.all import *
>>> var('x,y')
(x, y)
>>> f = exp(x**Integer(2)) - arcsin(pi+x)/y
>>> f._fricas_()                                                    # optional -
↪ fricas
     2
    x
y %e    - asin(x + %pi)
----------------------
           y
```

derivative(*ex*, *operator*)

Convert the derivative of `self` in FriCAS.

INPUT:

- `ex` – a symbolic expression
- `operator` – operator

Note that `ex.operator() == operator`.

EXAMPLES:

```
sage: var('x,y,z')
(x, y, z)
sage: f = function("F")
sage: f(x)._fricas_()                                                    #␣
↪optional - fricas
F(x)
sage: diff(f(x,y,z), x, z, x)._fricas_()                                 #␣
↪optional - fricas
F       (x,y,z)
 ,1,1,3
```

```
>>> from sage.all import *
>>> var('x,y,z')
(x, y, z)
>>> f = function("F")
>>> f(x)._fricas_()                                                 # optional -
↪ fricas
F(x)
>>> diff(f(x,y,z), x, z, x)._fricas_()                              # optional -
↪ fricas
F       (x,y,z)
 ,1,1,3
```

Check that Issue #25838 is fixed:

```
sage: var('x')
x
sage: F = function('F')
sage: integrate(F(x), x, algorithm='fricas')                          #␣
↪optional - fricas
integral(F(x), x)

sage: integrate(diff(F(x), x)*sin(F(x)), x, algorithm='fricas')      #␣
↪optional - fricas
-cos(F(x))
```

```
>>> from sage.all import *
>>> var('x')
x
>>> F = function('F')
>>> integrate(F(x), x, algorithm='fricas')                          # optional -
↪ fricas
integral(F(x), x)

>>> integrate(diff(F(x), x)*sin(F(x)), x, algorithm='fricas')      # optional -
↪ fricas
-cos(F(x))
```

Check that Issue #27310 is fixed:

```
sage: f = function("F")
sage: var("y")
y
sage: ex = (diff(f(x,y), x, x, y)).subs(y=x+y); ex
D[0, 0, 1](F)(x, x + y)
sage: fricas(ex)                                                      #␣
↪optional - fricas
F       (x,y + x)
 ,1,1,2
```

```
>>> from sage.all import *
>>> f = function("F")
>>> var("y")
y
>>> ex = (diff(f(x,y), x, x, y)).subs(y=x+y); ex
D[0, 0, 1](F)(x, x + y)
>>> fricas(ex)                                                      # optional -
↪ fricas
F       (x,y + x)
 ,1,1,2
```

pyobject(*ex*, *obj*)

Return a string which, when evaluated by FriCAS, returns the object as an expression.

We explicitly add the coercion to the FriCAS domains $ExpressionInteger$ and $ExpressionComplexInteger$ to make sure that elements of the symbolic ring are translated to these. In particular, this is needed for integration, see Issue #28641 and Issue #28647.

EXAMPLES:

```
sage: 2._fricas_().domainOf()                                               #␣
↪optional - fricas
PositiveInteger...

sage: (-1/2)._fricas_().domainOf()                                          #␣
↪optional - fricas
Fraction(Integer...)

sage: SR(2)._fricas_().domainOf()                                           #␣
↪optional - fricas
Expression(Integer...)

sage: (sqrt(2))._fricas_().domainOf()                                       #␣
↪optional - fricas
Expression(Integer...)

sage: pi._fricas_().domainOf()                                              #␣
↪optional - fricas
Pi...

sage: asin(pi)._fricas_()                                                   #␣
↪optional - fricas
asin(%pi)

sage: I._fricas_().domainOf()                                   # optional -␣
↪fricas
Complex(Integer...)

sage: SR(I)._fricas_().domainOf()                                           #␣
↪optional - fricas
Expression(Complex(Integer...))

sage: ex = (I+sqrt(2)+2)
sage: ex._fricas_().domainOf()                                              #␣
↪optional - fricas
Expression(Complex(Integer...))

sage: ex._fricas_()^2                                                       #␣
↪optional - fricas
           +-+
(4 + 2 %i)\|2  + 5 + 4 %i

sage: (ex^2)._fricas_()                                                     #␣
↪optional - fricas
           +-+
(4 + 2 %i)\|2  + 5 + 4 %i
```

```
>>> from sage.all import *
>>> Integer(2)._fricas_().domainOf()                                        #␣
↪optional - fricas
PositiveInteger...
```

(continues on next page)

(continued from previous page)

```
>>> (-Integer(1)/Integer(2))._fricas_().domainOf()                                    ␣
↪       # optional - fricas
Fraction(Integer...)

>>> SR(Integer(2))._fricas_().domainOf()                                            #␣
↪optional - fricas
Expression(Integer...)

>>> (sqrt(Integer(2)))._fricas_().domainOf()                                        #␣
↪optional - fricas
Expression(Integer...)

>>> pi._fricas_().domainOf()                                             # optional -
↪ fricas
Pi...

>>> asin(pi)._fricas_()                                                  # optional -
↪ fricas
asin(%pi)

>>> I._fricas_().domainOf()                                          # optional -␣
↪fricas
Complex(Integer...)

>>> SR(I)._fricas_().domainOf()                                          # optional -
↪ fricas
Expression(Complex(Integer...))

>>> ex = (I+sqrt(Integer(2))+Integer(2))
>>> ex._fricas_().domainOf()                                             # optional -
↪ fricas
Expression(Complex(Integer...))

>>> ex._fricas_()**Integer(2)                                                        #␣
↪optional - fricas
           +-+
(4 + 2 %i)\|2  + 5 + 4 %i

>>> (ex**Integer(2))._fricas_()                                                      #␣
↪optional - fricas
           +-+
(4 + 2 %i)\|2  + 5 + 4 %i
```

symbol(*ex*)

Convert the argument, which is a symbol, to FriCAS.

In this case, we do not return an $ExpressionInteger$, because FriCAS frequently requires elements of domain $Symbol$ or $Variable$ as arguments, for example to $integrate$. Moreover, FriCAS is able to do the conversion itself, whenever the argument should be interpreted as a symbolic expression.

EXAMPLES:

```
sage: x._fricas_().domainOf()                                            #␣
```

(continues on next page)

(continued from previous page)

```
↪optional - fricas
Variable(x)

sage: (x^2)._fricas_().domainOf()                                        #␣
↪optional - fricas
Expression(Integer...)

sage: (2*x)._fricas_().integrate(x)                                      #␣
↪optional - fricas
 2
x
```

```
>>> from sage.all import *
>>> x._fricas_().domainOf()                                     # optional -
↪ fricas
Variable(x)

>>> (x**Integer(2))._fricas_().domainOf()                                #␣
↪optional - fricas
Expression(Integer...)

>>> (Integer(2)*x)._fricas_().integrate(x)                              #␣
↪optional - fricas
 2
x
```

class sage.symbolic.expression_conversions.**HalfAngle**(*ex*)

Bases: `ExpressionTreeWalker`

A class that walks a symbolic expression tree, replacing each occurrence of a trigonometric or hyperbolic function by its expression as a rational fraction in the (hyperbolic) tangent of half the original argument.

Halves = [sin, cos, tan, csc, sec, cot, sinh, cosh, tanh, csch, sech, coth]

HalvesDict = {sinh: x |--> -2*tanh(1/2*x)/(tanh(1/2*x)^2 - 1), cosh: x |--> -(tanh(1/2*x)^2 + 1)/(tanh(1/2*x)^2 - 1), tanh: x |--> 2*tanh(1/2*x)/(tanh(1/2*x)^2 + 1), coth: x |--> 1/2*(tanh(1/2*x)^2 + 1)/tanh(1/2*x), sech: x |--> -(tanh(1/2*x)^2 - 1)/(tanh(1/2*x)^2 + 1), csch: x |--> -1/2*(tanh(1/2*x)^2 - 1)/tanh(1/2*x), sin: x |--> 2*tan(1/2*x)/(tan(1/2*x)^2 + 1), cos: x |--> -(tan(1/2*x)^2 - 1)/(tan(1/2*x)^2 + 1), tan: x |--> -2*tan(1/2*x)/(tan(1/2*x)^2 - 1), cot: x |--> -1/2*(tan(1/2*x)^2 - 1)/tan(1/2*x), sec: x |--> -(tan(1/2*x)^2 + 1)/(tan(1/2*x)^2 - 1), csc: x |--> 1/2*(tan(1/2*x)^2 + 1)/tan(1/2*x)}

Integer

alias of `Integer`

SR = Symbolic Ring

composition(*ex*, *op*)

Compose.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import HalfAngle
sage: x, t = SR.var("x, t")
sage: a = HalfAngle(cos(3*x)/(4-cos(x)).trig_expand())()
sage: a.subs(tan(x/2) == t).simplify_full()
(2*(t^2 + 1)*cos(3/2*x)^2 - t^2 - 1)/(5*t^2 + 3)
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import HalfAngle
>>> x, t = SR.var("x, t")
>>> a = HalfAngle(cos(Integer(3)*x)/(Integer(4)-cos(x)).trig_expand())()
>>> a.subs(tan(x/Integer(2)) == t).simplify_full()
(2*(t^2 + 1)*cos(3/2*x)^2 - t^2 - 1)/(5*t^2 + 3)
```

cos = cos

cosh = cosh

cot = cot

coth = coth

csc = csc

csch = csch

function(*s*, ***kwds*)

Create a formal symbolic function with the name *s*.

INPUT:

- `nargs=0` – number of arguments the function accepts, defaults to variable number of arguments, or 0
- `latex_name` – name used when printing in latex mode
- `conversions` – dictionary specifying names of this function in other systems, this is used by the interfaces internally during conversion
- `eval_func` – method used for automatic evaluation
- `evalf_func` – method used for numeric evaluation
- `evalf_params_first` – boolean to indicate if parameters should be evaluated numerically before calling the custom evalf function
- `conjugate_func` – method used for complex conjugation
- `real_part_func` – method used when taking real parts
- `imag_part_func` – method used when taking imaginary parts
- `derivative_func` – method to be used for (partial) derivation This method should take a keyword argument deriv_param specifying the index of the argument to differentiate w.r.t
- `tderivative_func` – method to be used for derivatives
- `power_func` – method used when taking powers This method should take a keyword argument power_param specifying the exponent
- `series_func` – method used for series expansion This method should expect keyword arguments - `order` – order for the expansion to be computed - `var` – variable to expand w.r.t. - `at` – expand at this value

- `print_func` – method for custom printing
- `print_latex_func` – method for custom printing in latex mode

Note that custom methods must be instance methods, i.e., expect the instance of the symbolic function as the first argument.

> **Note**
>
> The new function is both returned and automatically injected into the global namespace. If you use this function in library code, it is better to use sage.symbolic.function_factory.function, since it will not touch the global namespace.

EXAMPLES:

We create a formal function called supersin

```
sage: function('supersin')
supersin
```

```
>>> from sage.all import *
>>> function('supersin')
supersin
```

We can immediately use supersin in symbolic expressions:

```
sage: y, z, A = var('y z A')
sage: supersin(y+z) + A^3
A^3 + supersin(y + z)
```

```
>>> from sage.all import *
>>> y, z, A = var('y z A')
>>> supersin(y+z) + A**Integer(3)
A^3 + supersin(y + z)
```

We can define other functions in terms of supersin:

```
sage: g(x,y) = supersin(x)^2 + sin(y/2)
sage: g
(x, y) |--> supersin(x)^2 + sin(1/2*y)
sage: g.diff(y)
(x, y) |--> 1/2*cos(1/2*y)
sage: k = g.diff(x); k
(x, y) |--> 2*supersin(x)*diff(supersin(x), x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); g = symbolic_expression(supersin(x)**Integer(2) +␣
↪sin(y/Integer(2))).function(x,y)
>>> g
(x, y) |--> supersin(x)^2 + sin(1/2*y)
>>> g.diff(y)
(x, y) |--> 1/2*cos(1/2*y)
>>> k = g.diff(x); k
(x, y) |--> 2*supersin(x)*diff(supersin(x), x)
```

We create a formal function of one variable, write down an expression that involves first and second derivatives, and extract off coefficients:

```
sage: r, kappa = var('r,kappa')
sage: psi = function('psi', nargs=1)(r); psi
psi(r)
sage: g = 1/r^2*(2*r*psi.derivative(r,1) + r^2*psi.derivative(r,2)); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
sage: g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
sage: g.coefficient(psi.derivative(r,2))
1
sage: g.coefficient(psi.derivative(r,1))
2/r
```

```
>>> from sage.all import *
>>> r, kappa = var('r,kappa')
>>> psi = function('psi', nargs=Integer(1))(r); psi
psi(r)
>>> g = Integer(1)/r**Integer(2)*(Integer(2)*r*psi.derivative(r,Integer(1)) +␣
↪r**Integer(2)*psi.derivative(r,Integer(2))); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
>>> g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
>>> g.coefficient(psi.derivative(r,Integer(2)))
1
>>> g.coefficient(psi.derivative(r,Integer(1)))
2/r
```

Custom typesetting of symbolic functions in LaTeX, either using latex_name keyword:

```
sage: function('riemann', latex_name='\\mathcal{R}')
riemann
sage: latex(riemann(x))
\mathcal{R}\left(x\right)
```

```
>>> from sage.all import *
>>> function('riemann', latex_name='\\mathcal{R}')
riemann
>>> latex(riemann(x))
\mathcal{R}\left(x\right)
```

or passing a custom callable function that returns a latex expression:

```
sage: mu,nu = var('mu,nu')
sage: def my_latex_print(self, *args): return "\\psi_{%s}"%(', '.
↪join(map(latex, args)))
sage: function('psi', print_latex_func=my_latex_print)
psi
sage: latex(psi(mu,nu))
\psi_{\mu, \nu}
```

```
>>> from sage.all import *
>>> mu,nu = var('mu,nu')
>>> def my_latex_print(self, *args): return "\\psi_{%s}"%(', '.join(map(latex,
↪ args)))
>>> function('psi', print_latex_func=my_latex_print)
psi
>>> latex(psi(mu,nu))
\psi_{\mu, \nu}
```

Defining custom methods for automatic or numeric evaluation, derivation, conjugation, etc. is supported:

```
sage: def ev(self, x): return 2*x
sage: foo = function("foo", nargs=1, eval_func=ev)
sage: foo(x)
2*x
sage: foo = function("foo", nargs=1, eval_func=lambda self, x: 5)
sage: foo(x)
5
sage: def ef(self, x): pass
sage: bar = function("bar", nargs=1, eval_func=ef)
sage: bar(x)
bar(x)

sage: def evalf_f(self, x, parent=None, algorithm=None): return 6
sage: foo = function("foo", nargs=1, evalf_func=evalf_f)
sage: foo(x)
foo(x)
sage: foo(x).n()
6

sage: foo = function("foo", nargs=1, conjugate_func=ev)
sage: foo(x).conjugate()
2*x

sage: def deriv(self, *args, **kwds): print("{} {}".format(args, kwds));␣
↪return args[kwds['diff_param']]^2
sage: foo = function("foo", nargs=2, derivative_func=deriv)
sage: foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2

sage: def pow(self, x, power_param=None): print("{} {}".format(x, power_
↪param)); return x*power_param
sage: foo = function("foo", nargs=1, power_func=pow)
sage: foo(y)^(x+y)
y x + y
(x + y)*y

sage: from pprint import pformat
sage: def expand(self, *args, **kwds):
....:     print("{} {}".format(args, pformat(kwds)))
....:     return sum(args[0]^i for i in range(kwds['order']))
sage: foo = function("foo", nargs=1, series_func=expand)
```

(continues on next page)

(continued from previous page)

```
sage: foo(y).series(y, 5)
(y,) {'at': 0, 'options': 0, 'order': 5, 'var': y}
y^4 + y^3 + y^2 + y + 1

sage: def my_print(self, *args):
....:     return "my args are: " + ', '.join(map(repr, args))
sage: foo = function('t', nargs=2, print_func=my_print)
sage: foo(x,y^z)
my args are: x, y^z

sage: latex(foo(x,y^z))
t\left(x, y^{z}\right)
sage: foo = function('t', nargs=2, print_latex_func=my_print)
sage: foo(x,y^z)
t(x, y^z)
sage: latex(foo(x,y^z))
my args are: x, y^z
sage: foo = function('t', nargs=2, latex_name='foo')
sage: latex(foo(x,y^z))
foo\left(x, y^{z}\right)
```

```
>>> from sage.all import *
>>> def ev(self, x): return Integer(2)*x
>>> foo = function("foo", nargs=Integer(1), eval_func=ev)
>>> foo(x)
2*x
>>> foo = function("foo", nargs=Integer(1), eval_func=lambda self, x:␣
↪Integer(5))
>>> foo(x)
5
>>> def ef(self, x): pass
>>> bar = function("bar", nargs=Integer(1), eval_func=ef)
>>> bar(x)
bar(x)

>>> def evalf_f(self, x, parent=None, algorithm=None): return Integer(6)
>>> foo = function("foo", nargs=Integer(1), evalf_func=evalf_f)
>>> foo(x)
foo(x)
>>> foo(x).n()
6

>>> foo = function("foo", nargs=Integer(1), conjugate_func=ev)
>>> foo(x).conjugate()
2*x

>>> def deriv(self, *args, **kwds): print("{} {}".format(args, kwds)); return␣
↪args[kwds['diff_param']]**Integer(2)
>>> foo = function("foo", nargs=Integer(2), derivative_func=deriv)
>>> foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2
```

(continues on next page)

(continued from previous page)

```
>>> def pow(self, x, power_param=None): print("{} {}".format(x, power_param));
↪ return x*power_param
>>> foo = function("foo", nargs=Integer(1), power_func=pow)
>>> foo(y)**(x+y)
y x + y
(x + y)*y

>>> from pprint import pformat
>>> def expand(self, *args, **kwds):
...     print("{} {}".format(args, pformat(kwds)))
...     return sum(args[Integer(0)]**i for i in range(kwds['order']))
>>> foo = function("foo", nargs=Integer(1), series_func=expand)
>>> foo(y).series(y, Integer(5))
(y,) {'at': 0, 'options': 0, 'order': 5, 'var': y}
y^4 + y^3 + y^2 + y + 1

>>> def my_print(self, *args):
...     return "my args are: " + ', '.join(map(repr, args))
>>> foo = function('t', nargs=Integer(2), print_func=my_print)
>>> foo(x,y**z)
my args are: x, y^z

>>> latex(foo(x,y**z))
t\left(x, y^{z}\right)
>>> foo = function('t', nargs=Integer(2), print_latex_func=my_print)
>>> foo(x,y**z)
t(x, y^z)
>>> latex(foo(x,y**z))
my args are: x, y^z
>>> foo = function('t', nargs=Integer(2), latex_name='foo')
>>> latex(foo(x,y**z))
foo\left(x, y^{z}\right)
```

Chain rule:

```
sage: def print_args(self, *args, **kwds): print("args: {}".format(args));␣
↪print("kwds: {}".format(kwds)); return args[0]
sage: foo = function('t', nargs=2, tderivative_func=print_args)
sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': x}
x
sage: foo = function('t', nargs=2, derivative_func=print_args)
sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
args: (x, x)
kwds: {'diff_param': 1}
2*x
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> def print_args(self, *args, **kwds): print("args: {}".format(args));␣
↪print("kwds: {}".format(kwds)); return args[Integer(0)]
>>> foo = function('t', nargs=Integer(2), tderivative_func=print_args)
>>> foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': x}
x
>>> foo = function('t', nargs=Integer(2), derivative_func=print_args)
>>> foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
args: (x, x)
kwds: {'diff_param': 1}
2*x
```

Since Sage 4.0, basic arithmetic with unevaluated functions is no longer supported:

```
sage: x = var('x')
sage: f = function('f')
sage: 2*f
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring' and '<class
↪'sage.symbolic.function_factory...NewSymbolicFunction'>'
```

```
>>> from sage.all import *
>>> x = var('x')
>>> f = function('f')
>>> Integer(2)*f
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring' and '<class
↪'sage.symbolic.function_factory...NewSymbolicFunction'>'
```

You now need to evaluate the function in order to do the arithmetic:

```
sage: 2*f(x)
2*f(x)
```

```
>>> from sage.all import *
>>> Integer(2)*f(x)
2*f(x)
```

Since Sage 4.0, you need to use `substitute_function()` to replace all occurrences of a function with another:

```
sage: var('a, b')
(a, b)
sage: cr = function('cr')
sage: f = cr(a)
sage: g = f.diff(a).integral(b)
sage: g
```

(continues on next page)

(continued from previous page)

```
b*diff(cr(a), a)
sage: g.substitute_function(cr, cos)
-b*sin(a)

sage: g.substitute_function(cr, (sin(x) + cos(x)).function(x))
b*(cos(a) - sin(a))
```

```
>>> from sage.all import *
>>> var('a, b')
(a, b)
>>> cr = function('cr')
>>> f = cr(a)
>>> g = f.diff(a).integral(b)
>>> g
b*diff(cr(a), a)
>>> g.substitute_function(cr, cos)
-b*sin(a)

>>> g.substitute_function(cr, (sin(x) + cos(x)).function(x))
b*(cos(a) - sin(a))
```

half = 1/2

halfx = 1/2*x

one = 1

sec = sec

sech = sech

sin = sin

sinh = sinh

tan = tan

tanh = tanh

two = 2

x = x

class sage.symbolic.expression_conversions.**HoldRemover**(*ex*, *exclude=None*)

Bases: *ExpressionTreeWalker*

A class that walks the tree and evaluates every operator that is not in a given list of exceptions.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import HoldRemover
sage: ex = sin(pi*cos(0, hold=True), hold=True); ex
sin(pi*cos(0))
sage: h = HoldRemover(ex)
sage: h()
```

(continues on next page)

(continued from previous page)

```
0
sage: h = HoldRemover(ex, [sin])
sage: h()
sin(pi)
sage: h = HoldRemover(ex, [cos])
sage: h()
sin(pi*cos(0))
sage: ex = atan2(0, 0, hold=True) + hypergeometric([1,2], [3,4], 0, hold=True)
sage: h = HoldRemover(ex, [atan2])
sage: h()
arctan2(0, 0) + 1
sage: h = HoldRemover(ex, [hypergeometric])
sage: h()
NaN + hypergeometric((1, 2), (3, 4), 0)
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import HoldRemover
>>> ex = sin(pi*cos(Integer(0), hold=True), hold=True); ex
sin(pi*cos(0))
>>> h = HoldRemover(ex)
>>> h()
0
>>> h = HoldRemover(ex, [sin])
>>> h()
sin(pi)
>>> h = HoldRemover(ex, [cos])
>>> h()
sin(pi*cos(0))
>>> ex = atan2(Integer(0), Integer(0), hold=True) + hypergeometric([Integer(1),
↪Integer(2)], [Integer(3),Integer(4)], Integer(0), hold=True)
>>> h = HoldRemover(ex, [atan2])
>>> h()
arctan2(0, 0) + 1
>>> h = HoldRemover(ex, [hypergeometric])
>>> h()
NaN + hypergeometric((1, 2), (3, 4), 0)
```

composition(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import HoldRemover
sage: ex = sin(pi*cos(0, hold=True), hold=True); ex
sin(pi*cos(0))
sage: h = HoldRemover(ex)
sage: h()
0
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import HoldRemover
>>> ex = sin(pi*cos(Integer(0), hold=True), hold=True); ex
sin(pi*cos(0))
```

(continues on next page)

(continued from previous page)

```
>>> h = HoldRemover(ex)
>>> h()
0
```

class sage.symbolic.expression_conversions.**InterfaceInit**(*interface*)

Bases: *Converter*

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: a = pi + 2
sage: m(a)
'(%pi)+(2)'
sage: m(sin(a))
'sin((%pi)+(2))'
sage: m(exp(x^2) + pi + 2)
'(%pi)+(exp((_SAGE_VAR_x)^(2)))+(2)'
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import InterfaceInit
>>> m = InterfaceInit(maxima)
>>> a = pi + Integer(2)
>>> m(a)
'(%pi)+(2)'
>>> m(sin(a))
'sin((%pi)+(2))'
>>> m(exp(x**Integer(2)) + pi + Integer(2))
'(%pi)+(exp((_SAGE_VAR_x)^(2)))+(2)'
```

arithmetic(*ex*, *operator*)

EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.arithmetic(x+2, sage.symbolic.operators.add_vararg)
'(_SAGE_VAR_x)+(2)'
```

```
>>> from sage.all import *
>>> import operator
>>> from sage.symbolic.expression_conversions import InterfaceInit
>>> m = InterfaceInit(maxima)
>>> m.arithmetic(x+Integer(2), sage.symbolic.operators.add_vararg)
'(_SAGE_VAR_x)+(2)'
```

composition(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.composition(sin(x), sin)
```

(continues on next page)

(continued from previous page)

```
'sin(_SAGE_VAR_x)'
sage: m.composition(ceil(x), ceil)
'ceiling(_SAGE_VAR_x)'

sage: m = InterfaceInit(mathematica)
sage: m.composition(sin(x), sin)
'Sin[x]'
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import InterfaceInit
>>> m = InterfaceInit(maxima)
>>> m.composition(sin(x), sin)
'sin(_SAGE_VAR_x)'
>>> m.composition(ceil(x), ceil)
'ceiling(_SAGE_VAR_x)'

>>> m = InterfaceInit(mathematica)
>>> m.composition(sin(x), sin)
'Sin[x]'
```

derivative(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: f = function('f')
sage: a = f(x).diff(x); a
diff(f(x), x)
sage: print(m.derivative(a, a.operator()))
diff('f(_SAGE_VAR_x), _SAGE_VAR_x, 1)
sage: b = f(x).diff(x, x)
sage: print(m.derivative(b, b.operator()))
diff('f(_SAGE_VAR_x), _SAGE_VAR_x, 2)
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import InterfaceInit
>>> m = InterfaceInit(maxima)
>>> f = function('f')
>>> a = f(x).diff(x); a
diff(f(x), x)
>>> print(m.derivative(a, a.operator()))
diff('f(_SAGE_VAR_x), _SAGE_VAR_x, 1)
>>> b = f(x).diff(x, x)
>>> print(m.derivative(b, b.operator()))
diff('f(_SAGE_VAR_x), _SAGE_VAR_x, 2)
```

We can also convert expressions where the argument is not just a variable, but the result is an “at” expression using temporary variables:

```
sage: y = var('y')
sage: t = (f(x*y).diff(x))/y
sage: t
```

(continues on next page)

(continued from previous page)

```
D[0](f)(x*y)
sage: m.derivative(t, t.operator())
"at(diff('f(_SAGE_VAR__symbol0), _SAGE_VAR__symbol0, 1), [_SAGE_VAR__symbol0␣
↪= (_SAGE_VAR_x)*(_SAGE_VAR_y)])"
```

```
>>> from sage.all import *
>>> y = var('y')
>>> t = (f(x*y).diff(x))/y
>>> t
D[0](f)(x*y)
>>> m.derivative(t, t.operator())
"at(diff('f(_SAGE_VAR__symbol0), _SAGE_VAR__symbol0, 1), [_SAGE_VAR__symbol0␣
↪= (_SAGE_VAR_x)*(_SAGE_VAR_y)])"
```

pyobject (*ex*, *obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: ii = InterfaceInit(gp)
sage: f = 2+SR(I)
sage: ii.pyobject(f, f.pyobject())
'I + 2'

sage: ii.pyobject(SR(2), 2)
'2'

sage: ii.pyobject(pi, pi.pyobject())
'Pi'
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import InterfaceInit
>>> ii = InterfaceInit(gp)
>>> f = Integer(2)+SR(I)
>>> ii.pyobject(f, f.pyobject())
'I + 2'

>>> ii.pyobject(SR(Integer(2)), Integer(2))
'2'

>>> ii.pyobject(pi, pi.pyobject())
'Pi'
```

relation (*ex*, *operator*)

EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.relation(x==3, operator.eq)
'_SAGE_VAR_x = 3'
sage: m.relation(x==3, operator.lt)
'_SAGE_VAR_x < 3'
```

```
>>> from sage.all import *
>>> import operator
>>> from sage.symbolic.expression_conversions import InterfaceInit
>>> m = InterfaceInit(maxima)
>>> m.relation(x==Integer(3), operator.eq)
'_SAGE_VAR_x = 3'
>>> m.relation(x==Integer(3), operator.lt)
'_SAGE_VAR_x < 3'
```

symbol(*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.symbol(x)
'_SAGE_VAR_x'
sage: f(x) = x
sage: m.symbol(f)
'_SAGE_VAR_x'
sage: ii = InterfaceInit(gp)
sage: ii.symbol(x)
'x'
sage: g = InterfaceInit(giac)
sage: g.symbol(x)
'sageVARx'
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import InterfaceInit
>>> m = InterfaceInit(maxima)
>>> m.symbol(x)
'_SAGE_VAR_x'
>>> __tmp__=var("x"); f = symbolic_expression(x).function(x)
>>> m.symbol(f)
'_SAGE_VAR_x'
>>> ii = InterfaceInit(gp)
>>> ii.symbol(x)
'x'
>>> g = InterfaceInit(giac)
>>> g.symbol(x)
'sageVARx'
```

tuple(*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: t = SR._force_pyobject((3, 4, e^x))
sage: m.tuple(t)
'[3,4,exp(_SAGE_VAR_x)]'
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import InterfaceInit
```

(continues on next page)

(continued from previous page)

```
>>> m = InterfaceInit(maxima)
>>> t = SR._force_pyobject((Integer(3), Integer(4), e**x))
>>> m.tuple(t)
'[3,4,exp(_SAGE_VAR_x)]'
```

class sage.symbolic.expression_conversions.**LaurentPolynomialConverter**(*ex*, *base_ring=None*, *ring=None*)

Bases: *PolynomialConverter*

A converter from symbolic expressions to Laurent polynomials.

See *laurent_polynomial()* for details.

class sage.symbolic.expression_conversions.**PolynomialConverter**(*ex*, *base_ring=None*, *ring=None*)

Bases: *Converter*

A converter from symbolic expressions to polynomials.

See *polynomial()* for details.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import PolynomialConverter
sage: x, y = var('x,y')
sage: p = PolynomialConverter(x+y, base_ring=QQ)
sage: p.base_ring
Rational Field
sage: p.ring
Multivariate Polynomial Ring in x, y over Rational Field

sage: p = PolynomialConverter(x, base_ring=QQ)
sage: p.base_ring
Rational Field
sage: p.ring
Univariate Polynomial Ring in x over Rational Field

sage: p = PolynomialConverter(x, ring=QQ['x,y'])
sage: p.base_ring
Rational Field
sage: p.ring
Multivariate Polynomial Ring in x, y over Rational Field

sage: p = PolynomialConverter(x+y, ring=QQ['x'])
Traceback (most recent call last):
...
TypeError: y is not a variable of Univariate Polynomial Ring in x over Rational␣
↪Field
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import PolynomialConverter
>>> x, y = var('x,y')
>>> p = PolynomialConverter(x+y, base_ring=QQ)
>>> p.base_ring
```

(continues on next page)

(continued from previous page)

```
Rational Field
>>> p.ring
Multivariate Polynomial Ring in x, y over Rational Field

>>> p = PolynomialConverter(x, base_ring=QQ)
>>> p.base_ring
Rational Field
>>> p.ring
Univariate Polynomial Ring in x over Rational Field

>>> p = PolynomialConverter(x, ring=QQ['x,y'])
>>> p.base_ring
Rational Field
>>> p.ring
Multivariate Polynomial Ring in x, y over Rational Field

>>> p = PolynomialConverter(x+y, ring=QQ['x'])
Traceback (most recent call last):
...
TypeError: y is not a variable of Univariate Polynomial Ring in x over Rational␣
↪Field
```

arithmetic(*ex*, *operator*)

EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import PolynomialConverter

sage: x, y = var('x, y')
sage: p = PolynomialConverter(x, base_ring=RR)
sage: p.arithmetic(pi+e, operator.add)
5.85987448204884
sage: p.arithmetic(x^2, operator.pow)
x^2

sage: p = PolynomialConverter(x+y, base_ring=RR)
sage: p.arithmetic(x*y+y^2, operator.add)
x*y + y^2

sage: p = PolynomialConverter(y^(3/2), ring=SR['x'])
sage: p.arithmetic(y^(3/2), operator.pow)
y^(3/2)
sage: _.parent()
Symbolic Ring
```

```
>>> from sage.all import *
>>> import operator
>>> from sage.symbolic.expression_conversions import PolynomialConverter

>>> x, y = var('x, y')
>>> p = PolynomialConverter(x, base_ring=RR)
>>> p.arithmetic(pi+e, operator.add)
```

(continues on next page)

(continued from previous page)

```
5.85987448204884
>>> p.arithmetic(x**Integer(2), operator.pow)
x^2

>>> p = PolynomialConverter(x+y, base_ring=RR)
>>> p.arithmetic(x*y+y**Integer(2), operator.add)
x*y + y^2

>>> p = PolynomialConverter(y**(Integer(3)/Integer(2)), ring=SR['x'])
>>> p.arithmetic(y**(Integer(3)/Integer(2)), operator.pow)
y^(3/2)
>>> _.parent()
Symbolic Ring
```

composition(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import PolynomialConverter
sage: a = sin(2)
sage: p = PolynomialConverter(a*x, base_ring=RR)
sage: p.composition(a, a.operator())
0.909297426825682
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import PolynomialConverter
>>> a = sin(Integer(2))
>>> p = PolynomialConverter(a*x, base_ring=RR)
>>> p.composition(a, a.operator())
0.909297426825682
```

pyobject(*ex*, *obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import PolynomialConverter
sage: p = PolynomialConverter(x, base_ring=QQ)
sage: f = SR(2)
sage: p.pyobject(f, f.pyobject())
2
sage: _.parent()
Rational Field
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import PolynomialConverter
>>> p = PolynomialConverter(x, base_ring=QQ)
>>> f = SR(Integer(2))
>>> p.pyobject(f, f.pyobject())
2
>>> _.parent()
Rational Field
```

relation(*ex*, *op*)

EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import PolynomialConverter

sage: x, y = var('x, y')
sage: p = PolynomialConverter(x, base_ring=RR)

sage: p.relation(x==3, operator.eq)
x - 3.00000000000000
sage: p.relation(x==3, operator.lt)
Traceback (most recent call last):
...
ValueError: Unable to represent as a polynomial

sage: p = PolynomialConverter(x - y, base_ring=QQ)
sage: p.relation(x^2 - y^3 + 1 == x^3, operator.eq)
-x^3 - y^3 + x^2 + 1
```

```
>>> from sage.all import *
>>> import operator
>>> from sage.symbolic.expression_conversions import PolynomialConverter

>>> x, y = var('x, y')
>>> p = PolynomialConverter(x, base_ring=RR)

>>> p.relation(x==Integer(3), operator.eq)
x - 3.00000000000000
>>> p.relation(x==Integer(3), operator.lt)
Traceback (most recent call last):
...
ValueError: Unable to represent as a polynomial

>>> p = PolynomialConverter(x - y, base_ring=QQ)
>>> p.relation(x**Integer(2) - y**Integer(3) + Integer(1) == x**Integer(3),␣
↪operator.eq)
-x^3 - y^3 + x^2 + 1
```

symbol(*ex*)

Return a variable in the polynomial ring.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import PolynomialConverter
sage: p = PolynomialConverter(x, base_ring=QQ)
sage: p.symbol(x)
x
sage: _.parent()
Univariate Polynomial Ring in x over Rational Field
sage: y = var('y')
sage: p = PolynomialConverter(x*y, ring=SR['x'])
sage: p.symbol(y)
y
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import PolynomialConverter
>>> p = PolynomialConverter(x, base_ring=QQ)
>>> p.symbol(x)
x
>>> _.parent()
Univariate Polynomial Ring in x over Rational Field
>>> y = var('y')
>>> p = PolynomialConverter(x*y, ring=SR['x'])
>>> p.symbol(y)
y
```

class sage.symbolic.expression_conversions.**RingConverter**(*R*, *subs_dict=None*)

Bases: *Converter*

A class to convert expressions to other rings.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF, subs_dict={x:2})
sage: R.ring
Real Interval Field with 53 bits of precision
sage: R.subs_dict
{x: 2}
sage: R(pi+e)
5.85987448204884?
sage: loads(dumps(R))
<sage.symbolic.expression_conversions.RingConverter object at 0x...>
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import RingConverter
>>> R = RingConverter(RIF, subs_dict={x:Integer(2)})
>>> R.ring
Real Interval Field with 53 bits of precision
>>> R.subs_dict
{x: 2}
>>> R(pi+e)
5.85987448204884?
>>> loads(dumps(R))
<sage.symbolic.expression_conversions.RingConverter object at 0x...>
```

arithmetic(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: P.<z> = ZZ[]
sage: R = RingConverter(P, subs_dict={x:z})
sage: a = 2*x^2 + x + 3
sage: R(a)
2*z^2 + z + 3
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import RingConverter
>>> P = ZZ['z']; (z,) = P._first_ngens(1)
>>> R = RingConverter(P, subs_dict={x:z})
>>> a = Integer(2)*x**Integer(2) + x + Integer(3)
>>> R(a)
2*z^2 + z + 3
```

composition (*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF)
sage: R(cos(2))
-0.4161468365471424?
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import RingConverter
>>> R = RingConverter(RIF)
>>> R(cos(Integer(2)))
-0.4161468365471424?
```

pyobject (*ex*, *obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF)
sage: R(SR(5/2))
2.5000000000000000?
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import RingConverter
>>> R = RingConverter(RIF)
>>> R(SR(Integer(5)/Integer(2)))
2.5000000000000000?
```

symbol (*ex*)

All symbols appearing in the expression must either appear in *subs_dict* or be convertible by the ring's element constructor in order for the conversion to be successful.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF, subs_dict={x:2})
sage: R(x+pi)
5.141592653589794?

sage: R = RingConverter(RIF)
sage: R(x+pi)
Traceback (most recent call last):
...
TypeError: unable to simplify to a real interval approximation
```

(continues on next page)

(continued from previous page)

```
sage: R = RingConverter(QQ['x'])
sage: R(x^2+x)
x^2 + x
sage: R(x^2+x).parent()
Univariate Polynomial Ring in x over Rational Field
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import RingConverter
>>> R = RingConverter(RIF, subs_dict={x:Integer(2)})
>>> R(x+pi)
5.141592653589794?

>>> R = RingConverter(RIF)
>>> R(x+pi)
Traceback (most recent call last):
...
TypeError: unable to simplify to a real interval approximation

>>> R = RingConverter(QQ['x'])
>>> R(x**Integer(2)+x)
x^2 + x
>>> R(x**Integer(2)+x).parent()
Univariate Polynomial Ring in x over Rational Field
```

class sage.symbolic.expression_conversions.**SubstituteFunction**(*ex*, **args*)

Bases: *ExpressionTreeWalker*

A class that walks the tree and replaces occurrences of a function with another.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SubstituteFunction
sage: foo = function('foo'); bar = function('bar')
sage: s = SubstituteFunction(foo(x), {foo: bar})
sage: s(1/foo(foo(x)) + foo(2))
1/bar(bar(x)) + bar(2)
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import SubstituteFunction
>>> foo = function('foo'); bar = function('bar')
>>> s = SubstituteFunction(foo(x), {foo: bar})
>>> s(Integer(1)/foo(foo(x)) + foo(Integer(2)))
1/bar(bar(x)) + bar(2)
```

composition(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SubstituteFunction
sage: foo = function('foo'); bar = function('bar')
sage: s = SubstituteFunction(foo(x), {foo: bar})
sage: f = foo(x)
```

(continues on next page)

(continued from previous page)

```
sage: s.composition(f, f.operator())
bar(x)
sage: f = foo(foo(x))
sage: s.composition(f, f.operator())
bar(bar(x))
sage: f = sin(foo(x))
sage: s.composition(f, f.operator())
sin(bar(x))
sage: f = foo(sin(x))
sage: s.composition(f, f.operator())
bar(sin(x))
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import SubstituteFunction
>>> foo = function('foo'); bar = function('bar')
>>> s = SubstituteFunction(foo(x), {foo: bar})
>>> f = foo(x)
>>> s.composition(f, f.operator())
bar(x)
>>> f = foo(foo(x))
>>> s.composition(f, f.operator())
bar(bar(x))
>>> f = sin(foo(x))
>>> s.composition(f, f.operator())
sin(bar(x))
>>> f = foo(sin(x))
>>> s.composition(f, f.operator())
bar(sin(x))
```

derivative(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SubstituteFunction
sage: foo = function('foo'); bar = function('bar')
sage: s = SubstituteFunction(foo(x), {foo: bar})
sage: f = foo(x).diff(x)
sage: s.derivative(f, f.operator())
diff(bar(x), x)
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import SubstituteFunction
>>> foo = function('foo'); bar = function('bar')
>>> s = SubstituteFunction(foo(x), {foo: bar})
>>> f = foo(x).diff(x)
>>> s.derivative(f, f.operator())
diff(bar(x), x)
```

sage.symbolic.expression_conversions.**fast_callable**(*ex*, *etb*)

Given an ExpressionTreeBuilder *etb*, return an Expression representing the symbolic expression *ex*.

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x','y'])
sage: x,y = var('x,y')
sage: f = y+2*x^2
sage: f._fast_callable_(etb)
add(mul(ipow(v_0, 2), 2), v_1)

sage: f = (2*x^3+2*x-1)/((x-2)*(x+1))
sage: f._fast_callable_(etb)
div(add(add(mul(ipow(v_0, 3), 2), mul(v_0, 2)), -1), mul(add(v_0, 1), add(v_0, -
↪2)))
```

```
>>> from sage.all import *
>>> from sage.ext.fast_callable import ExpressionTreeBuilder
>>> etb = ExpressionTreeBuilder(vars=['x','y'])
>>> x,y = var('x,y')
>>> f = y+Integer(2)*x**Integer(2)
>>> f._fast_callable_(etb)
add(mul(ipow(v_0, 2), 2), v_1)

>>> f = (Integer(2)*x**Integer(3)+Integer(2)*x-Integer(1))/((x-
↪Integer(2))*(x+Integer(1)))
>>> f._fast_callable_(etb)
div(add(add(mul(ipow(v_0, 3), 2), mul(v_0, 2)), -1), mul(add(v_0, 1), add(v_0, -
↪2)))
```

sage.symbolic.expression_conversions.**laurent_polynomial**(*ex*, *base_ring=None*, *ring=None*)

Return a Laurent polynomial from the symbolic expression `ex`.

INPUT:

- `ex` – a symbolic expression
- `base_ring`, `ring` – either a `base_ring` or a Laurent polynomial `ring` can be specified for the parent of result. If just a `base_ring` is given, then the variables of the `base_ring` will be the variables of the expression `ex`.

OUTPUT: a Laurent polynomial

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import laurent_polynomial
sage: f = x^2 + 2/x
sage: laurent_polynomial(f, base_ring=QQ)
2*x^-1 + x^2
sage: _.parent()
Univariate Laurent Polynomial Ring in x over Rational Field

sage: laurent_polynomial(f, ring=LaurentPolynomialRing(QQ, 'x, y'))
x^2 + 2*x^-1
sage: _.parent()
Multivariate Laurent Polynomial Ring in x, y over Rational Field

sage: x, y = var('x, y')
sage: laurent_polynomial(x + 1/y^2, ring=LaurentPolynomialRing(QQ, 'x, y'))
```

(continues on next page)

(continued from previous page)

```
x + y^-2
sage: _.parent()
Multivariate Laurent Polynomial Ring in x, y over Rational Field
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import laurent_polynomial
>>> f = x**Integer(2) + Integer(2)/x
>>> laurent_polynomial(f, base_ring=QQ)
2*x^-1 + x^2
>>> _.parent()
Univariate Laurent Polynomial Ring in x over Rational Field

>>> laurent_polynomial(f, ring=LaurentPolynomialRing(QQ, 'x, y'))
x^2 + 2*x^-1
>>> _.parent()
Multivariate Laurent Polynomial Ring in x, y over Rational Field

>>> x, y = var('x, y')
>>> laurent_polynomial(x + Integer(1)/y**Integer(2),␣
↪ring=LaurentPolynomialRing(QQ, 'x, y'))
x + y^-2
>>> _.parent()
Multivariate Laurent Polynomial Ring in x, y over Rational Field
```

sage.symbolic.expression_conversions.**polynomial**(*ex*, *base_ring=None*, *ring=None*)

Return a polynomial from the symbolic expression `ex`.

INPUT:

- `ex` – a symbolic expression
- `base_ring`, `ring` – either a `base_ring` or a polynomial `ring` can be specified for the parent of result. If just a `base_ring` is given, then the variables of the `base_ring` will be the variables of the expression `ex`.

OUTPUT: a polynomial

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import polynomial
sage: f = x^2 + 2
sage: polynomial(f, base_ring=QQ)
x^2 + 2
sage: _.parent()
Univariate Polynomial Ring in x over Rational Field

sage: polynomial(f, ring=QQ['x,y'])
x^2 + 2
sage: _.parent()
Multivariate Polynomial Ring in x, y over Rational Field

sage: x, y = var('x, y')
sage: polynomial(x + y^2, ring=QQ['x,y'])
y^2 + x
sage: _.parent()
```

(continues on next page)

(continued from previous page)

```
Multivariate Polynomial Ring in x, y over Rational Field

sage: s,t = var('s,t')
sage: expr = t^2-2*s*t+1
sage: expr.polynomial(None,ring=SR['t'])
t^2 - 2*s*t + 1
sage: _.parent()
Univariate Polynomial Ring in t over Symbolic Ring

sage: polynomial(x*y, ring=SR['x'])
y*x

sage: polynomial(y - sqrt(x), ring=SR['y'])
y - sqrt(x)
sage: _.list()
[-sqrt(x), 1]
```

```
>>> from sage.all import *
>>> from sage.symbolic.expression_conversions import polynomial
>>> f = x**Integer(2) + Integer(2)
>>> polynomial(f, base_ring=QQ)
x^2 + 2
>>> _.parent()
Univariate Polynomial Ring in x over Rational Field

>>> polynomial(f, ring=QQ['x,y'])
x^2 + 2
>>> _.parent()
Multivariate Polynomial Ring in x, y over Rational Field

>>> x, y = var('x, y')
>>> polynomial(x + y**Integer(2), ring=QQ['x,y'])
y^2 + x
>>> _.parent()
Multivariate Polynomial Ring in x, y over Rational Field

>>> s,t = var('s,t')
>>> expr = t**Integer(2)-Integer(2)*s*t+Integer(1)
>>> expr.polynomial(None,ring=SR['t'])
t^2 - 2*s*t + 1
>>> _.parent()
Univariate Polynomial Ring in t over Symbolic Ring

>>> polynomial(x*y, ring=SR['x'])
y*x

>>> polynomial(y - sqrt(x), ring=SR['y'])
y - sqrt(x)
>>> _.list()
[-sqrt(x), 1]
```

The polynomials can have arbitrary (constant) coefficients so long as they coerce into the base ring:

```
sage: polynomial(2^sin(2)*x^2 + exp(3), base_ring=RR)
1.87813065119873*x^2 + 20.0855369231877
```

```
>>> from sage.all import *
>>> polynomial(Integer(2)**sin(Integer(2))*x**Integer(2) + exp(Integer(3)), base_
↪ring=RR)
1.87813065119873*x^2 + 20.0855369231877
```

2.18 Complexity Measures

Some measures of symbolic expression complexity. Each complexity measure is expected to take a symbolic expression as an argument, and return a number.

`sage.symbolic.complexity_measures.`**`string_length`**`(`*expr*`)`

Return the length of `expr` after converting it to a string.

INPUT:

- `expr` – the expression whose complexity we want to measure

OUTPUT: a real number representing the complexity of `expr`

RATIONALE:

If the expression is longer on-screen, then a human would probably consider it more complex.

EXAMPLES:

This expression has three characters, `x`, `^`, and `2`:

```
sage: from sage.symbolic.complexity_measures import string_length
sage: f = x^2
sage: string_length(f)
3
```

```
>>> from sage.all import *
>>> from sage.symbolic.complexity_measures import string_length
>>> f = x**Integer(2)
>>> string_length(f)
3
```

2.19 Further examples from Wester's paper

These are all the problems at http://yacas.sourceforge.net/essaysmanual.html

They come from the 1994 paper "Review of CAS mathematical capabilities", by Michael Wester, who put forward 123 problems that a reasonable computer algebra system should be able to solve and tested the then current versions of various commercial CAS on this list. Sage can do most of the problems natively now, i.e., with no explicit calls to Maxima or other systems.

```
sage: # (YES) factorial of 50, and factor it
sage: factorial(50)
30414093201713378043612608166064768844377641568960512000000000000
sage: factor(factorial(50))
```

(continues on next page)

(continued from previous page)

```
2^47 * 3^22 * 5^12 * 7^8 * 11^4 * 13^3 * 17^2 * 19^2 * 23^2 * 29 * 31 * 37 * 41 * 43␣
↪* 47
```

```
>>> from sage.all import *
>>> # (YES) factorial of 50, and factor it
>>> factorial(Integer(50))
30414093201713378043612608166064768844377641568960512000000000000
>>> factor(factorial(Integer(50)))
2^47 * 3^22 * 5^12 * 7^8 * 11^4 * 13^3 * 17^2 * 19^2 * 23^2 * 29 * 31 * 37 * 41 * 43␣
↪* 47
```

```
sage: # (YES) 1/2+...+1/10 = 4861/2520
sage: sum(1/n for n in range(2,10+1)) == 4861/2520
True
```

```
>>> from sage.all import *
>>> # (YES) 1/2+...+1/10 = 4861/2520
>>> sum(Integer(1)/n for n in range(Integer(2),Integer(10)+Integer(1))) ==␣
↪Integer(4861)/Integer(2520)
True
```

```
sage: # (YES) Evaluate  e^(Pi*Sqrt(163)) to 50 decimal digits
sage: a = e^(pi*sqrt(163)); a
e^(sqrt(163)*pi)
sage: RealField(150)(a)
2.6253741264076874399999999999925007259719820e17
```

```
>>> from sage.all import *
>>> # (YES) Evaluate  e^(Pi*Sqrt(163)) to 50 decimal digits
>>> a = e**(pi*sqrt(Integer(163))); a
e^(sqrt(163)*pi)
>>> RealField(Integer(150))(a)
2.6253741264076874399999999999925007259719820e17
```

```
sage: # (YES) Evaluate the Bessel function J[2] numerically at z=1+I.
sage: bessel_J(2, 1+I).n()
0.0415798869439621 + 0.247397641513306*I
```

```
>>> from sage.all import *
>>> # (YES) Evaluate the Bessel function J[2] numerically at z=1+I.
>>> bessel_J(Integer(2), Integer(1)+I).n()
0.0415798869439621 + 0.247397641513306*I
```

```
sage: # (YES) Obtain period of decimal fraction 1/7=0.(142857).
sage: a = 1/7
sage: a
1/7
sage: a.period()
6
```

```
>>> from sage.all import *
>>> # (YES) Obtain period of decimal fraction 1/7=0.(142857).
>>> a = Integer(1)/Integer(7)
>>> a
1/7
>>> a.period()
6
```

```
sage: # (YES) Continued fraction of 3.1415926535
sage: a = 3.1415926535
sage: continued_fraction(a)
[3; 7, 15, 1, 292, 1, 1, 6, 2, 13, 4]
```

```
>>> from sage.all import *
>>> # (YES) Continued fraction of 3.1415926535
>>> a = RealNumber('3.1415926535')
>>> continued_fraction(a)
[3; 7, 15, 1, 292, 1, 1, 6, 2, 13, 4]
```

```
sage: # (YES) Sqrt(2*Sqrt(3)+4)=1+Sqrt(3).
sage: # The Maxima backend equality checker does this;
sage: # note the equality only holds for one choice of sign,
sage: # but Maxima always chooses the "positive" one
sage: a = sqrt(2*sqrt(3) + 4); b = 1 + sqrt(3)
sage: float(a-b)
0.0
sage: bool(a == b)
True
sage: # We can, of course, do this in a quadratic field
sage: k.<sqrt3> = QuadraticField(3)
sage: asqr = 2*sqrt3 + 4
sage: b = 1+sqrt3
sage: asqr == b^2
True
```

```
>>> from sage.all import *
>>> # (YES) Sqrt(2*Sqrt(3)+4)=1+Sqrt(3).
>>> # The Maxima backend equality checker does this;
>>> # note the equality only holds for one choice of sign,
>>> # but Maxima always chooses the "positive" one
>>> a = sqrt(Integer(2)*sqrt(Integer(3)) + Integer(4)); b = Integer(1) +␣
↪sqrt(Integer(3))
>>> float(a-b)
0.0
>>> bool(a == b)
True
>>> # We can, of course, do this in a quadratic field
>>> k = QuadraticField(Integer(3), names=('sqrt3',)); (sqrt3,) = k._first_ngens(1)
>>> asqr = Integer(2)*sqrt3 + Integer(4)
>>> b = Integer(1)+sqrt3
>>> asqr == b**Integer(2)
True
```

```
sage: # (YES) Sqrt(14+3*Sqrt(3+2*Sqrt(5-12*Sqrt(3-2*Sqrt(2)))))=3+Sqrt(2).
sage: a = sqrt(14+3*sqrt(3+2*sqrt(5-12*sqrt(3-2*sqrt(2)))))
sage: b = 3+sqrt(2)
sage: a, b
(sqrt(3*sqrt(2*sqrt(-12*sqrt(-2*sqrt(2) + 3) + 5) + 3) + 14), sqrt(2) + 3)
sage: bool(a==b)
True
sage: abs(float(a-b)) < 1e-10
True
sage: # 2*Infinity-3=Infinity.
sage: 2*infinity-3 == infinity
True
```

```
>>> from sage.all import *
>>> # (YES) Sqrt(14+3*Sqrt(3+2*Sqrt(5-12*Sqrt(3-2*Sqrt(2)))))=3+Sqrt(2).
>>> a = sqrt(Integer(14)+Integer(3)*sqrt(Integer(3)+Integer(2)*sqrt(Integer(5)-
↪Integer(12)*sqrt(Integer(3)-Integer(2)*sqrt(Integer(2))))))
>>> b = Integer(3)+sqrt(Integer(2))
>>> a, b
(sqrt(3*sqrt(2*sqrt(-12*sqrt(-2*sqrt(2) + 3) + 5) + 3) + 14), sqrt(2) + 3)
>>> bool(a==b)
True
>>> abs(float(a-b)) < RealNumber('1e-10')
True
>>> # 2*Infinity-3=Infinity.
>>> Integer(2)*infinity-Integer(3) == infinity
True
```

```
sage: # (YES) Standard deviation of the sample (1, 2, 3, 4, 5).
sage: v = vector(RDF, 5, [1,2,3,4,5])
sage: v.standard_deviation()
1.5811388300841898
```

```
>>> from sage.all import *
>>> # (YES) Standard deviation of the sample (1, 2, 3, 4, 5).
>>> v = vector(RDF, Integer(5), [Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)])
>>> v.standard_deviation()
1.5811388300841898
```

```
sage: # (NO) Hypothesis testing with t-distribution.
sage: # (NO) Hypothesis testing with chi^2 distribution
sage: # (But both are included in Scipy and R)
```

```
>>> from sage.all import *
>>> # (NO) Hypothesis testing with t-distribution.
>>> # (NO) Hypothesis testing with chi^2 distribution
>>> # (But both are included in Scipy and R)
```

```
sage: # (YES) (x^2-4)/(x^2+4*x+4)=(x-2)/(x+2).
sage: R.<x> = QQ[]
```

(continues on next page)

(continued from previous page)

```
sage: (x^2-4)/(x^2+4*x+4) == (x-2)/(x+2)
True
sage: restore('x')
```

```
>>> from sage.all import *
>>> # (YES) (x^2-4)/(x^2+4*x+4)=(x-2)/(x+2).
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> (x**Integer(2)-Integer(4))/(x**Integer(2)+Integer(4)*x+Integer(4)) == (x-
↪Integer(2))/(x+Integer(2))
True
>>> restore('x')
```

```
sage: # (YES -- Maxima doesn't immediately consider them
sage: # equal, but simplification shows that they are)
sage: # (Exp(x)-1)/(Exp(x/2)+1)=Exp(x/2)-1.
sage: f = (exp(x)-1)/(exp(x/2)+1)
sage: g = exp(x/2)-1
sage: f
(e^x - 1)/(e^(1/2*x) + 1)
sage: g
e^(1/2*x) - 1
sage: f.canonicalize_radical()
e^(1/2*x) - 1
sage: g
e^(1/2*x) - 1
sage: f(x=10.0).n(53), g(x=10.0).n(53)
(147.413159102577, 147.413159102577)
sage: bool(f == g)
True
```

```
>>> from sage.all import *
>>> # (YES -- Maxima doesn't immediately consider them
>>> # equal, but simplification shows that they are)
>>> # (Exp(x)-1)/(Exp(x/2)+1)=Exp(x/2)-1.
>>> f = (exp(x)-Integer(1))/(exp(x/Integer(2))+Integer(1))
>>> g = exp(x/Integer(2))-Integer(1)
>>> f
(e^x - 1)/(e^(1/2*x) + 1)
>>> g
e^(1/2*x) - 1
>>> f.canonicalize_radical()
e^(1/2*x) - 1
>>> g
e^(1/2*x) - 1
>>> f(x=RealNumber('10.0')).n(Integer(53)), g(x=RealNumber('10.0')).n(Integer(53))
(147.413159102577, 147.413159102577)
>>> bool(f == g)
True
```

```
sage: # (YES) Expand (1+x)^20, take derivative and factorize.
sage: # first do it using algebraic polys
```

(continues on next page)

(continued from previous page)

```
sage: R.<x> = QQ[]
sage: f = (1+x)^20; f
x^20 + 20*x^19 + 190*x^18 + 1140*x^17 + 4845*x^16 + 15504*x^15 + 38760*x^14 + 77520*x^
↪13 + 125970*x^12 + 167960*x^11 + 184756*x^10 + 167960*x^9 + 125970*x^8 + 77520*x^7␣
↪+ 38760*x^6 + 15504*x^5 + 4845*x^4 + 1140*x^3 + 190*x^2 + 20*x + 1
sage: deriv = f.derivative()
sage: deriv
20*x^19 + 380*x^18 + 3420*x^17 + 19380*x^16 + 77520*x^15 + 232560*x^14 + 542640*x^13␣
↪+ 1007760*x^12 + 1511640*x^11 + 1847560*x^10 + 1847560*x^9 + 1511640*x^8 +␣
↪1007760*x^7 + 542640*x^6 + 232560*x^5 + 77520*x^4 + 19380*x^3 + 3420*x^2 + 380*x +␣
↪20
sage: deriv.factor()
(20) * (x + 1)^19
sage: restore('x')
sage: # next do it symbolically
sage: var('y')
y
sage: f = (1+y)^20; f
(y + 1)^20
sage: g = f.expand(); g
y^20 + 20*y^19 + 190*y^18 + 1140*y^17 + 4845*y^16 + 15504*y^15 + 38760*y^14 + 77520*y^
↪13 + 125970*y^12 + 167960*y^11 + 184756*y^10 + 167960*y^9 + 125970*y^8 + 77520*y^7␣
↪+ 38760*y^6 + 15504*y^5 + 4845*y^4 + 1140*y^3 + 190*y^2 + 20*y + 1
sage: deriv = g.derivative(); deriv
20*y^19 + 380*y^18 + 3420*y^17 + 19380*y^16 + 77520*y^15 + 232560*y^14 + 542640*y^13␣
↪+ 1007760*y^12 + 1511640*y^11 + 1847560*y^10 + 1847560*y^9 + 1511640*y^8 +␣
↪1007760*y^7 + 542640*y^6 + 232560*y^5 + 77520*y^4 + 19380*y^3 + 3420*y^2 + 380*y +␣
↪20
sage: deriv.factor()
20*(y + 1)^19
```

```
>>> from sage.all import *
>>> # (YES) Expand (1+x)^20, take derivative and factorize.
>>> # first do it using algebraic polys
>>> R = QQ['x']; (x,) = R._first_ngens(1)
>>> f = (Integer(1)+x)**Integer(20); f
x^20 + 20*x^19 + 190*x^18 + 1140*x^17 + 4845*x^16 + 15504*x^15 + 38760*x^14 + 77520*x^
↪13 + 125970*x^12 + 167960*x^11 + 184756*x^10 + 167960*x^9 + 125970*x^8 + 77520*x^7␣
↪+ 38760*x^6 + 15504*x^5 + 4845*x^4 + 1140*x^3 + 190*x^2 + 20*x + 1
>>> deriv = f.derivative()
>>> deriv
20*x^19 + 380*x^18 + 3420*x^17 + 19380*x^16 + 77520*x^15 + 232560*x^14 + 542640*x^13␣
↪+ 1007760*x^12 + 1511640*x^11 + 1847560*x^10 + 1847560*x^9 + 1511640*x^8 +␣
↪1007760*x^7 + 542640*x^6 + 232560*x^5 + 77520*x^4 + 19380*x^3 + 3420*x^2 + 380*x +␣
↪20
>>> deriv.factor()
(20) * (x + 1)^19
>>> restore('x')
>>> # next do it symbolically
>>> var('y')
y
>>> f = (Integer(1)+y)**Integer(20); f
```

(continues on next page)

(continued from previous page)

```
(y + 1)^20
>>> g = f.expand(); g
y^20 + 20*y^19 + 190*y^18 + 1140*y^17 + 4845*y^16 + 15504*y^15 + 38760*y^14 + 77520*y^
↪13 + 125970*y^12 + 167960*y^11 + 184756*y^10 + 167960*y^9 + 125970*y^8 + 77520*y^7␣
↪+ 38760*y^6 + 15504*y^5 + 4845*y^4 + 1140*y^3 + 190*y^2 + 20*y + 1
>>> deriv = g.derivative(); deriv
20*y^19 + 380*y^18 + 3420*y^17 + 19380*y^16 + 77520*y^15 + 232560*y^14 + 542640*y^13␣
↪+ 1007760*y^12 + 1511640*y^11 + 1847560*y^10 + 1847560*y^9 + 1511640*y^8 +␣
↪1007760*y^7 + 542640*y^6 + 232560*y^5 + 77520*y^4 + 19380*y^3 + 3420*y^2 + 380*y +␣
↪20
>>> deriv.factor()
20*(y + 1)^19
```

```
sage: # (YES) Factorize x^100-1.
sage: factor(x^100-1)
(x^40 - x^30 + x^20 - x^10 + 1)*(x^20 + x^15 + x^10 + x^5 + 1)*(x^20 - x^15 + x^10 -␣
↪x^5 + 1)*(x^8 - x^6 + x^4 - x^2 + 1)*(x^4 + x^3 + x^2 + x + 1)*(x^4 - x^3 + x^2 - x␣
↪+ 1)*(x^2 + 1)*(x + 1)*(x - 1)
sage: # Also, algebraically
sage: x = polygen(QQ)
sage: factor(x^100 - 1)
(x - 1) * (x + 1) * (x^2 + 1) * (x^4 - x^3 + x^2 - x + 1) * (x^4 + x^3 + x^2 + x + 1)␣
↪* (x^8 - x^6 + x^4 - x^2 + 1) * (x^20 - x^15 + x^10 - x^5 + 1) * (x^20 + x^15 + x^
↪10 + x^5 + 1) * (x^40 - x^30 + x^20 - x^10 + 1)
sage: restore('x')
```

```
>>> from sage.all import *
>>> # (YES) Factorize x^100-1.
>>> factor(x**Integer(100)-Integer(1))
(x^40 - x^30 + x^20 - x^10 + 1)*(x^20 + x^15 + x^10 + x^5 + 1)*(x^20 - x^15 + x^10 -␣
↪x^5 + 1)*(x^8 - x^6 + x^4 - x^2 + 1)*(x^4 + x^3 + x^2 + x + 1)*(x^4 - x^3 + x^2 - x␣
↪+ 1)*(x^2 + 1)*(x + 1)*(x - 1)
>>> # Also, algebraically
>>> x = polygen(QQ)
>>> factor(x**Integer(100) - Integer(1))
(x - 1) * (x + 1) * (x^2 + 1) * (x^4 - x^3 + x^2 - x + 1) * (x^4 + x^3 + x^2 + x + 1)␣
↪* (x^8 - x^6 + x^4 - x^2 + 1) * (x^20 - x^15 + x^10 - x^5 + 1) * (x^20 + x^15 + x^
↪10 + x^5 + 1) * (x^40 - x^30 + x^20 - x^10 + 1)
>>> restore('x')
```

```
sage: # (YES) Factorize  x^4-3*x^2+1 in the field of rational numbers extended by␣
↪roots of  x^2-x-1.
sage: x = polygen(ZZ, 'x')
sage: k.< a> = NumberField(x^2 - x -1)
sage: R.< y> = k[]
sage: f = y^4 - 3*y^2 + 1
sage: f
y^4 - 3*y^2 + 1
sage: factor(f)
(y - a) * (y - a + 1) * (y + a - 1) * (y + a)
```

```
>>> from sage.all import *
>>> # (YES) Factorize  x^4-3*x^2+1 in the field of rational numbers extended by roots␣
↪of  x^2-x-1.
>>> x = polygen(ZZ, 'x')
>>> k = NumberField(x**Integer(2) - x -Integer(1), names=('a',)); (a,) = k._first_
↪ngens(1)
>>> R = k['y']; (y,) = R._first_ngens(1)
>>> f = y**Integer(4) - Integer(3)*y**Integer(2) + Integer(1)
>>> f
y^4 - 3*y^2 + 1
>>> factor(f)
(y - a) * (y - a + 1) * (y + a - 1) * (y + a)
```

```
sage: # (YES) Factorize  x^4-3*x^2+1 mod 5.
sage: k.< x > = GF(5) [ ]
sage: f = x^4 - 3*x^2 + 1
sage: f.factor()
(x + 2)^2 * (x + 3)^2
sage: # Alternatively, from symbol x as follows:
sage: reset('x')
sage: f = x^4 - 3*x^2 + 1
sage: f.polynomial(GF(5)).factor()
(x + 2)^2 * (x + 3)^2
```

```
>>> from sage.all import *
>>> # (YES) Factorize  x^4-3*x^2+1 mod 5.
>>> k = GF(Integer(5)) ['x']; (x,) = k._first_ngens(1)
>>> f = x**Integer(4) - Integer(3)*x**Integer(2) + Integer(1)
>>> f.factor()
(x + 2)^2 * (x + 3)^2
>>> # Alternatively, from symbol x as follows:
>>> reset('x')
>>> f = x**Integer(4) - Integer(3)*x**Integer(2) + Integer(1)
>>> f.polynomial(GF(Integer(5))).factor()
(x + 2)^2 * (x + 3)^2
```

```
sage: # (YES) Partial fraction decomposition of (x^2+2*x+3)/(x^3+4*x^2+5*x+2)
sage: f = (x^2+2*x+3)/(x^3+4*x^2+5*x+2); f
(x^2 + 2*x + 3)/(x^3 + 4*x^2 + 5*x + 2)
sage: f.partial_fraction()
3/(x + 2) - 2/(x + 1) + 2/(x + 1)^2
```

```
>>> from sage.all import *
>>> # (YES) Partial fraction decomposition of (x^2+2*x+3)/(x^3+4*x^2+5*x+2)
>>> f = (x**Integer(2)+Integer(2)*x+Integer(3))/
↪(x**Integer(3)+Integer(4)*x**Integer(2)+Integer(5)*x+Integer(2)); f
(x^2 + 2*x + 3)/(x^3 + 4*x^2 + 5*x + 2)
>>> f.partial_fraction()
3/(x + 2) - 2/(x + 1) + 2/(x + 1)^2
```

```
sage: # (YES) Assuming  x>=y,  y>=z,  z>=x, deduce  x=z.
```

(continues on next page)

(continued from previous page)

```
sage: forget()
sage: var('x,y,z')
(x, y, z)
sage: assume(x>=y, y>=z,z>=x)
sage: bool(x==z)
True
```

```
>>> from sage.all import *
>>> # (YES) Assuming  x>=y,  y>=z,  z>=x, deduce  x=z.
>>> forget()
>>> var('x,y,z')
(x, y, z)
>>> assume(x>=y, y>=z,z>=x)
>>> bool(x==z)
True
```

```
sage: # (YES) Assuming x>y, y>0, deduce 2*x^2>2*y^2.
sage: forget()
sage: assume(x>y, y>0)
sage: sorted(assumptions())
[x > y, y > 0]
sage: bool(2*x^2 > 2*y^2)
True
sage: forget()
sage: assumptions()
[]
```

```
>>> from sage.all import *
>>> # (YES) Assuming x>y, y>0, deduce 2*x^2>2*y^2.
>>> forget()
>>> assume(x>y, y>Integer(0))
>>> sorted(assumptions())
[x > y, y > 0]
>>> bool(Integer(2)*x**Integer(2) > Integer(2)*y**Integer(2))
True
>>> forget()
>>> assumptions()
[]
```

```
sage: # (NO) Solve the inequality Abs(x-1)>2.
sage: # Maxima doesn't solve inequalities
sage: # (but some Maxima packages do):
sage: eqn = abs(x-1) > 2
sage: eqn.solve(x)
[[x < -1], [3 < x]]
```

```
>>> from sage.all import *
>>> # (NO) Solve the inequality Abs(x-1)>2.
>>> # Maxima doesn't solve inequalities
>>> # (but some Maxima packages do):
>>> eqn = abs(x-Integer(1)) > Integer(2)
```

(continues on next page)

(continued from previous page)

```
>>> eqn.solve(x)
[[x < -1], [3 < x]]
```

```
sage: # (NO) Solve the inequality (x-1)*...*(x-5)<0.
sage: eqn = prod(x-i for i in range(1,5 +1)) < 0
sage: # but don't know how to solve
sage: eqn.solve(x)
[[x < 1], [x > 2, x < 3], [x > 4, x < 5]]
```

```
>>> from sage.all import *
>>> # (NO) Solve the inequality (x-1)*...*(x-5)<0.
>>> eqn = prod(x-i for i in range(Integer(1),Integer(5) +Integer(1))) < Integer(0)
>>> # but don't know how to solve
>>> eqn.solve(x)
[[x < 1], [x > 2, x < 3], [x > 4, x < 5]]
```

```
sage: # (YES) Cos(3*x)/Cos(x)=Cos(x)^2-3*Sin(x)^2 or similar equivalent combination.
sage: f = cos(3*x)/cos(x)
sage: g = cos(x)^2 - 3*sin(x)^2
sage: h = f-g
sage: h.trig_simplify()
                                          0
```

```
>>> from sage.all import *
>>> # (YES) Cos(3*x)/Cos(x)=Cos(x)^2-3*Sin(x)^2 or similar equivalent combination.
>>> f = cos(Integer(3)*x)/cos(x)
>>> g = cos(x)**Integer(2) - Integer(3)*sin(x)**Integer(2)
>>> h = f-g
>>> h.trig_simplify()
                                          0
```

```
sage: # (YES) Cos(3*x)/Cos(x)=2*Cos(2*x)-1.
sage: f = cos(3*x)/cos(x)
sage: g = 2*cos(2*x) - 1
sage: h = f-g
sage: h.trig_simplify()
                                          0
```

```
>>> from sage.all import *
>>> # (YES) Cos(3*x)/Cos(x)=2*Cos(2*x)-1.
>>> f = cos(Integer(3)*x)/cos(x)
>>> g = Integer(2)*cos(Integer(2)*x) - Integer(1)
>>> h = f-g
>>> h.trig_simplify()
                                          0
```

```
sage: # (GOOD ENOUGH) Define rewrite rules to match  Cos(3*x)/Cos(x)=Cos(x)^2-
↪3*Sin(x)^2.
sage: # Sage has no notion of "rewrite rules", but
sage: # it can simplify both to the same thing.
```

(continues on next page)

(continued from previous page)

```
sage: (cos(3*x)/cos(x)).simplify_full()
4*cos(x)^2 - 3
sage: (cos(x)^2-3*sin(x)^2).simplify_full()
4*cos(x)^2 - 3
```

```
>>> from sage.all import *
>>> # (GOOD ENOUGH) Define rewrite rules to match  Cos(3*x)/Cos(x)=Cos(x)^2-3*Sin(x)^
↪2.
>>> # Sage has no notion of "rewrite rules", but
>>> # it can simplify both to the same thing.
>>> (cos(Integer(3)*x)/cos(x)).simplify_full()
4*cos(x)^2 - 3
>>> (cos(x)**Integer(2)-Integer(3)*sin(x)**Integer(2)).simplify_full()
4*cos(x)^2 - 3
```

```
sage: # (YES) Sqrt(997)-(997^3)^(1/6)=0
sage: a = sqrt(997) - (997^3)^(1/6)
sage: a.simplify()
0
sage: bool(a == 0)
True
```

```
>>> from sage.all import *
>>> # (YES) Sqrt(997)-(997^3)^(1/6)=0
>>> a = sqrt(Integer(997)) - (Integer(997)**Integer(3))**(Integer(1)/Integer(6))
>>> a.simplify()
0
>>> bool(a == Integer(0))
True
```

```
sage: # (YES) Sqrt(99983)-99983^3^(1/6)=0
sage: a = sqrt(99983) - (99983^3)^(1/6)
sage: bool(a==0)
True
sage: float(a)
1.1368683772...e-13
sage: 13*7691
99983
```

```
>>> from sage.all import *
>>> # (YES) Sqrt(99983)-99983^3^(1/6)=0
>>> a = sqrt(Integer(99983)) - (Integer(99983)**Integer(3))**(Integer(1)/Integer(6))
>>> bool(a==Integer(0))
True
>>> float(a)
1.1368683772...e-13
>>> Integer(13)*Integer(7691)
99983
```

```
sage: # (YES) (2^(1/3) + 4^(1/3))^3 - 6*(2^(1/3) + 4^(1/3))-6 = 0
```

(continues on next page)

(continued from previous page)

```
sage: a = (2^(1/3) + 4^(1/3))^3 - 6*(2^(1/3) + 4^(1/3)) - 6; a
(4^(1/3) + 2^(1/3))^3 - 6*4^(1/3) - 6*2^(1/3) - 6
sage: bool(a==0)
True
sage: abs(float(a)) < 1e-10
True
```

```
>>> from sage.all import *
>>> # (YES) (2^(1/3) + 4^(1/3))^3 - 6*(2^(1/3) + 4^(1/3))-6 = 0
>>> a = (Integer(2)**(Integer(1)/Integer(3)) + Integer(4)**(Integer(1)/
↪Integer(3)))**Integer(3) - Integer(6)*(Integer(2)**(Integer(1)/Integer(3)) +␣
↪Integer(4)**(Integer(1)/Integer(3))) - Integer(6); a
(4^(1/3) + 2^(1/3))^3 - 6*4^(1/3) - 6*2^(1/3) - 6
>>> bool(a==Integer(0))
True
>>> abs(float(a)) < RealNumber('1e-10')
True
```

Or we can do it using number fields.

```
sage: reset('x')
sage: k.<b> = NumberField(x^3-2)
sage: a = (b  + b^2)^3 - 6*(b  + b^2) - 6
sage: a
0
```

```
>>> from sage.all import *
>>> reset('x')
>>> k = NumberField(x**Integer(3)-Integer(2), names=('b',)); (b,) = k._first_ngens(1)
>>> a = (b  + b**Integer(2))**Integer(3) - Integer(6)*(b  + b**Integer(2)) -␣
↪Integer(6)
>>> a
0
```

```
sage: # (NO, except numerically) Ln(Tan(x/2+Pi/4))-ArcSinh(Tan(x))=0
# Sage uses the Maxima convention when comparing symbolic expressions and
# returns True only when it can prove equality. Thus, in this case, we get
# False even though the equality holds.
sage: f = log(tan(x/2 + pi/4)) - arcsinh(tan(x))
sage: bool(f == 0)
False
sage: [abs(float(f(x=i/10))) < 1e-15 for i in range(1,5)]
[True, True, True, True]
sage: # Numerically, the expression Ln(Tan(x/2+Pi/4))-ArcSinh(Tan(x))=0 and its␣
↪derivative at x=0 are zero.
sage: g = f.derivative()
sage: abs(float(f(x=0))) < 1e-10
True
sage: abs(float(g(x=0))) < 1e-10
True
sage: g
-sqrt(tan(x)^2 + 1) + 1/2*(tan(1/4*pi + 1/2*x)^2 + 1)/tan(1/4*pi + 1/2*x)
```

```
>>> from sage.all import *
>>> # (NO, except numerically) Ln(Tan(x/2+Pi/4))-ArcSinh(Tan(x))=0
# Sage uses the Maxima convention when comparing symbolic expressions and
# returns True only when it can prove equality. Thus, in this case, we get
# False even though the equality holds.
>>> f = log(tan(x/Integer(2) + pi/Integer(4))) - arcsinh(tan(x))
>>> bool(f == Integer(0))
False
>>> [abs(float(f(x=i/Integer(10)))) < RealNumber('1e-15') for i in range(Integer(1),
↪Integer(5))]
[True, True, True, True]
>>> # Numerically, the expression Ln(Tan(x/2+Pi/4))-ArcSinh(Tan(x))=0 and its␣
↪derivative at x=0 are zero.
>>> g = f.derivative()
>>> abs(float(f(x=Integer(0)))) < RealNumber('1e-10')
True
>>> abs(float(g(x=Integer(0)))) < RealNumber('1e-10')
True
>>> g
-sqrt(tan(x)^2 + 1) + 1/2*(tan(1/4*pi + 1/2*x)^2 + 1)/tan(1/4*pi + 1/2*x)
```

```
sage: # (NO) Ln((2*Sqrt(r) + 1)/Sqrt(4*r 4*Sqrt(r) 1))=0.
sage: var('r')
r
sage: f = log( (2*sqrt(r) + 1) / sqrt(4*r  + 4*sqrt(r) +  1)); f
log((2*sqrt(r) + 1)/sqrt(4*r + 4*sqrt(r) + 1))
sage: bool(f == 0)
False
sage: [abs(float(f(r=i))) < 1e-10 for i in [0.1,0.3,0.5]]
[True, True, True]
```

```
>>> from sage.all import *
>>> # (NO) Ln((2*Sqrt(r) + 1)/Sqrt(4*r 4*Sqrt(r) 1))=0.
>>> var('r')
r
>>> f = log( (Integer(2)*sqrt(r) + Integer(1)) / sqrt(Integer(4)*r  +␣
↪Integer(4)*sqrt(r) +  Integer(1))); f
log((2*sqrt(r) + 1)/sqrt(4*r + 4*sqrt(r) + 1))
>>> bool(f == Integer(0))
False
>>> [abs(float(f(r=i))) < RealNumber('1e-10') for i in [RealNumber('0.1'),RealNumber(
↪'0.3'),RealNumber('0.5')]]
[True, True, True]
```

```
sage: # (NO)
sage: # (4*r+4*Sqrt(r)+1)^(Sqrt(r)/(2*Sqrt(r)+1))*(2*Sqrt(r)+1)^(2*Sqrt(r)+1)^(-1)-
↪2*Sqrt(r)-1=0, assuming r>0.
sage: assume(r>0)
sage: f = (4*r+4*sqrt(r)+1)^(sqrt(r)/(2*sqrt(r)+1))*(2*sqrt(r)+1)^(2*sqrt(r)+1)^(-1)-
↪2*sqrt(r)-1
sage: f
(4*r + 4*sqrt(r) + 1)^(sqrt(r)/(2*sqrt(r) + 1))*(2*sqrt(r) + 1)^(1/(2*sqrt(r) + 1)) -␣
```

(continues on next page)

(continued from previous page)

```
↪2*sqrt(r) - 1
sage: bool(f == 0)
False
sage: [abs(float(f(r=i))) < 1e-10 for i in [0.1,0.3,0.5]]
[True, True, True]
```

```
>>> from sage.all import *
>>> # (NO)
>>> # (4*r+4*Sqrt(r)+1)^(Sqrt(r)/(2*Sqrt(r)+1))*(2*Sqrt(r)+1)^(2*Sqrt(r)+1)^(-1)-
↪2*Sqrt(r)-1=0, assuming r>0.
>>> assume(r>Integer(0))
>>> f = (Integer(4)*r+Integer(4)*sqrt(r)+Integer(1))**(sqrt(r)/(Integer(2)*sqrt(r)+Int
↪eger(1)))*(Integer(2)*sqrt(r)+Integer(1))**(Integer(2)*sqrt(r)+Integer(1))**(-Intege
↪r(1))-Integer(2)*sqrt(r)-Integer(1)
>>> f
(4*r + 4*sqrt(r) + 1)^(sqrt(r)/(2*sqrt(r) + 1))*(2*sqrt(r) + 1)^(1/(2*sqrt(r) + 1)) -␣
↪2*sqrt(r) - 1
>>> bool(f == Integer(0))
False
>>> [abs(float(f(r=i))) < RealNumber('1e-10') for i in [RealNumber('0.1'),RealNumber(
↪'0.3'),RealNumber('0.5')]]
[True, True, True]
```

```
sage: # (YES) Obtain real and imaginary parts of Ln(3+4*I).
sage: a = log(3+4*I); a
log(4*I + 3)
sage: a.real()
log(5)
sage: a.imag()
arctan(4/3)
```

```
>>> from sage.all import *
>>> # (YES) Obtain real and imaginary parts of Ln(3+4*I).
>>> a = log(Integer(3)+Integer(4)*I); a
log(4*I + 3)
>>> a.real()
log(5)
>>> a.imag()
arctan(4/3)
```

```
sage: # (YES) Obtain real and imaginary parts of Tan(x+I*y)
sage: z = var('z')
sage: a = tan(z); a
tan(z)
sage: a.real()
sin(2*real_part(z))/(cos(2*real_part(z)) + cosh(2*imag_part(z)))
sage: a.imag()
sinh(2*imag_part(z))/(cos(2*real_part(z)) + cosh(2*imag_part(z)))
```

```
>>> from sage.all import *
>>> # (YES) Obtain real and imaginary parts of Tan(x+I*y)
```

(continues on next page)

(continued from previous page)

```
>>> z = var('z')
>>> a = tan(z); a
tan(z)
>>> a.real()
sin(2*real_part(z))/(cos(2*real_part(z)) + cosh(2*imag_part(z)))
>>> a.imag()
sinh(2*imag_part(z))/(cos(2*real_part(z)) + cosh(2*imag_part(z)))
```

```
sage: # (YES) Simplify Ln(Exp(z)) to z for -Pi<Im(z)<=Pi.
sage: # Unfortunately (?), Maxima does this even without
sage: # any assumptions.
sage: # We *would* use assume(-pi < imag(z))
sage: # and assume(imag(z) <= pi)
sage: f = log(exp(z)); f
log(e^z)
sage: f.simplify()
z
sage: forget()
```

```
>>> from sage.all import *
>>> # (YES) Simplify Ln(Exp(z)) to z for -Pi<Im(z)<=Pi.
>>> # Unfortunately (?), Maxima does this even without
>>> # any assumptions.
>>> # We *would* use assume(-pi < imag(z))
>>> # and assume(imag(z) <= pi)
>>> f = log(exp(z)); f
log(e^z)
>>> f.simplify()
z
>>> forget()
```

```
sage: # (YES) Assuming Re(x)>0, Re(y)>0, deduce x^(1/n)*y^(1/n)-(x*y)^(1/n)=0.
sage: # Maxima 5.26 has different behaviours depending on the current
sage: # domain.
sage: # To stick with the behaviour of previous versions, the domain is set
sage: # to 'real' in the following.
sage: # See Issue #10682 for further details.
sage: n = var('n')
sage: f = x^(1/n)*y^(1/n)-(x*y)^(1/n)
sage: assume(real(x) > 0, real(y) > 0)
sage: f.simplify()
x^(1/n)*y^(1/n) - (x*y)^(1/n)
sage: maxima = sage.calculus.calculus.maxima
sage: maxima.set('domain', 'real') # set domain to real
sage: f.simplify()
0
sage: maxima.set('domain', 'complex') # set domain back to its default value
sage: forget()
```

```
>>> from sage.all import *
>>> # (YES) Assuming Re(x)>0, Re(y)>0, deduce x^(1/n)*y^(1/n)-(x*y)^(1/n)=0.
```

(continues on next page)

(continued from previous page)

```
>>> # Maxima 5.26 has different behaviours depending on the current
>>> # domain.
>>> # To stick with the behaviour of previous versions, the domain is set
>>> # to 'real' in the following.
>>> # See Issue #10682 for further details.
>>> n = var('n')
>>> f = x**(Integer(1)/n)*y**(Integer(1)/n)-(x*y)**(Integer(1)/n)
>>> assume(real(x) > Integer(0), real(y) > Integer(0))
>>> f.simplify()
x^(1/n)*y^(1/n) - (x*y)^(1/n)
>>> maxima = sage.calculus.calculus.maxima
>>> maxima.set('domain', 'real') # set domain to real
>>> f.simplify()
0
>>> maxima.set('domain', 'complex') # set domain back to its default value
>>> forget()
```

```
sage: # (YES) Transform equations, (x==2)/2+(1==1)=>x/2+1==2.
sage: eq1 = x == 2
sage: eq2 = SR(1) == SR(1)
sage: eq1/2 + eq2
1/2*x + 1 == 2
```

```
>>> from sage.all import *
>>> # (YES) Transform equations, (x==2)/2+(1==1)=>x/2+1==2.
>>> eq1 = x == Integer(2)
>>> eq2 = SR(Integer(1)) == SR(Integer(1))
>>> eq1/Integer(2) + eq2
1/2*x + 1 == 2
```

```
sage: # (SOMEWHAT) Solve Exp(x)=1 and get all solutions.
sage: # to_poly_solve in Maxima can do this.
sage: solve(exp(x) == 1, x)
[x == 0]
```

```
>>> from sage.all import *
>>> # (SOMEWHAT) Solve Exp(x)=1 and get all solutions.
>>> # to_poly_solve in Maxima can do this.
>>> solve(exp(x) == Integer(1), x)
[x == 0]
```

```
sage: # (SOMEWHAT) Solve Tan(x)=1 and get all solutions.
sage: # to_poly_solve in Maxima can do this.
sage: solve(tan(x) == 1, x)
[x == 1/4*pi]
```

```
>>> from sage.all import *
>>> # (SOMEWHAT) Solve Tan(x)=1 and get all solutions.
>>> # to_poly_solve in Maxima can do this.
>>> solve(tan(x) == Integer(1), x)
[x == 1/4*pi]
```

```
sage: # (YES) Solve a degenerate 3x3 linear system.
sage: # x+y+z==6,2*x+y+2*z==10,x+3*y+z==10
sage: # First symbolically:
sage: solve([x+y+z==6, 2*x+y+2*z==10, x+3*y+z==10], x,y,z)
[[x == -r1 + 4, y == 2, z == r1]]
```

```
>>> from sage.all import *
>>> # (YES) Solve a degenerate 3x3 linear system.
>>> # x+y+z==6,2*x+y+2*z==10,x+3*y+z==10
>>> # First symbolically:
>>> solve([x+y+z==Integer(6), Integer(2)*x+y+Integer(2)*z==Integer(10),␣
↪x+Integer(3)*y+z==Integer(10)], x,y,z)
[[x == -r1 + 4, y == 2, z == r1]]
```

```
sage: # (YES) Invert a 2x2 symbolic matrix.
sage: # [[a,b],[1,a*b]]
sage: # Using multivariate poly ring -- much nicer
sage: R.<a,b> = QQ[]
sage: m = matrix(2,2,[a,b,  1, a*b])
sage: zz = m^(-1)
sage: zz
[     a/(a^2 - 1)    (-1)/(a^2 - 1)]
[(-1)/(a^2*b - b)     a/(a^2*b - b)]
```

```
>>> from sage.all import *
>>> # (YES) Invert a 2x2 symbolic matrix.
>>> # [[a,b],[1,a*b]]
>>> # Using multivariate poly ring -- much nicer
>>> R = QQ['a, b']; (a, b,) = R._first_ngens(2)
>>> m = matrix(Integer(2),Integer(2),[a,b,  Integer(1), a*b])
>>> zz = m**(-Integer(1))
>>> zz
[     a/(a^2 - 1)    (-1)/(a^2 - 1)]
[(-1)/(a^2*b - b)     a/(a^2*b - b)]
```

```
sage: # (YES) Compute and factor the determinant of the 4x4 Vandermonde matrix in a,␣
↪b, c, d.
sage: var('a,b,c,d')
(a, b, c, d)
sage: m = matrix(SR, 4, 4, [[z^i for i in range(4)] for z in [a,b,c,d]])
sage: m
[  1   a a^2 a^3]
[  1   b b^2 b^3]
[  1   c c^2 c^3]
[  1   d d^2 d^3]
sage: d = m.determinant()
sage: d.factor()
(a - b)*(a - c)*(a - d)*(b - c)*(b - d)*(c - d)
```

```
>>> from sage.all import *
>>> # (YES) Compute and factor the determinant of the 4x4 Vandermonde matrix in a, b,␣
```

(continues on next page)

(continued from previous page)

```
↪c, d.
>>> var('a,b,c,d')
(a, b, c, d)
>>> m = matrix(SR, Integer(4), Integer(4), [[z**i for i in range(Integer(4))] for z␣
↪in [a,b,c,d]])
>>> m
[  1   a a^2 a^3]
[  1   b b^2 b^3]
[  1   c c^2 c^3]
[  1   d d^2 d^3]
>>> d = m.determinant()
>>> d.factor()
(a - b)*(a - c)*(a - d)*(b - c)*(b - d)*(c - d)
```

```
sage: # (YES) Compute and factor the determinant of the 4x4 Vandermonde matrix in a,␣
↪b, c, d.
sage: # Do it instead in a multivariate ring
sage: R.<a,b,c,d> = QQ[]
sage: m = matrix(R, 4, 4, [[z^i for i in range(4)] for z in [a,b,c,d]])
sage: m
[  1   a a^2 a^3]
[  1   b b^2 b^3]
[  1   c c^2 c^3]
[  1   d d^2 d^3]
sage: d = m.determinant()
sage: d
a^3*b^2*c - a^2*b^3*c - a^3*b*c^2 + a*b^3*c^2 + a^2*b*c^3 - a*b^2*c^3 - a^3*b^2*d + a^
↪2*b^3*d + a^3*c^2*d - b^3*c^2*d - a^2*c^3*d + b^2*c^3*d + a^3*b*d^2 - a*b^3*d^2 - a^
↪3*c*d^2 + b^3*c*d^2 + a*c^3*d^2 - b*c^3*d^2 - a^2*b*d^3 + a*b^2*d^3 + a^2*c*d^3 - b^
↪2*c*d^3 - a*c^2*d^3 + b*c^2*d^3
sage: d.factor()
(-1) * (-a + c) * (-a + b) * (-b + c) * (c - d) * (b - d) * (a - d)
```

```
>>> from sage.all import *
>>> # (YES) Compute and factor the determinant of the 4x4 Vandermonde matrix in a, b,␣
↪c, d.
>>> # Do it instead in a multivariate ring
>>> R = QQ['a, b, c, d']; (a, b, c, d,) = R._first_ngens(4)
>>> m = matrix(R, Integer(4), Integer(4), [[z**i for i in range(Integer(4))] for z in␣
↪[a,b,c,d]])
>>> m
[  1   a a^2 a^3]
[  1   b b^2 b^3]
[  1   c c^2 c^3]
[  1   d d^2 d^3]
>>> d = m.determinant()
>>> d
a^3*b^2*c - a^2*b^3*c - a^3*b*c^2 + a*b^3*c^2 + a^2*b*c^3 - a*b^2*c^3 - a^3*b^2*d + a^
↪2*b^3*d + a^3*c^2*d - b^3*c^2*d - a^2*c^3*d + b^2*c^3*d + a^3*b*d^2 - a*b^3*d^2 - a^
↪3*c*d^2 + b^3*c*d^2 + a*c^3*d^2 - b*c^3*d^2 - a^2*b*d^3 + a*b^2*d^3 + a^2*c*d^3 - b^
↪2*c*d^3 - a*c^2*d^3 + b*c^2*d^3
```

(continues on next page)

(continued from previous page)

```
>>> d.factor()
(-1) * (-a + c) * (-a + b) * (-b + c) * (c - d) * (b - d) * (a - d)
```

```
sage: # (YES) Find the eigenvalues of a 3x3 integer matrix.
sage: m = matrix(QQ, 3, [5,-3,-7, -2,1,2, 2,-3,-4])
sage: m.eigenspaces_left()
[(3,
  Vector space of degree 3 and dimension 1 over Rational Field
  User basis matrix:
  [ 1  0 -1]),
 (1,
  Vector space of degree 3 and dimension 1 over Rational Field
  User basis matrix:
  [ 1  1 -1]),
 (-2,
  Vector space of degree 3 and dimension 1 over Rational Field
  User basis matrix:
  [0 1 1])]
```

```
>>> from sage.all import *
>>> # (YES) Find the eigenvalues of a 3x3 integer matrix.
>>> m = matrix(QQ, Integer(3), [Integer(5),-Integer(3),-Integer(7), -Integer(2),
↪Integer(1),Integer(2), Integer(2),-Integer(3),-Integer(4)])
>>> m.eigenspaces_left()
[(3,
  Vector space of degree 3 and dimension 1 over Rational Field
  User basis matrix:
  [ 1  0 -1]),
 (1,
  Vector space of degree 3 and dimension 1 over Rational Field
  User basis matrix:
  [ 1  1 -1]),
 (-2,
  Vector space of degree 3 and dimension 1 over Rational Field
  User basis matrix:
  [0 1 1])]
```

```
sage: # (YES) Verify some standard limits found by L'Hopital's rule:
sage: #   Verify(Limit(x,Infinity) (1+1/x)^x, Exp(1));
sage: #   Verify(Limit(x,0) (1-Cos(x))/x^2, 1/2);
sage: limit( (1+1/x)^x, x = oo)
e
sage: limit( (1-cos(x))/(x^2), x = 1/2)
-4*cos(1/2) + 4
```

```
>>> from sage.all import *
>>> # (YES) Verify some standard limits found by L'Hopital's rule:
>>> #   Verify(Limit(x,Infinity) (1+1/x)^x, Exp(1));
>>> #   Verify(Limit(x,0) (1-Cos(x))/x^2, 1/2);
>>> limit( (Integer(1)+Integer(1)/x)**x, x = oo)
e
```

(continues on next page)

(continued from previous page)

```
>>> limit( (Integer(1)-cos(x))/(x**Integer(2)), x = Integer(1)/Integer(2))
-4*cos(1/2) + 4
```

```
sage: # (OK-ish) D(x)Abs(x)
sage: #    Verify(D(x) Abs(x), Sign(x));
sage: diff(abs(x))
1/2*(x + conjugate(x))/abs(x)
sage: _.simplify_full()
x/abs(x)
sage: _ = var('x', domain='real')
sage: diff(abs(x))
x/abs(x)
sage: forget()
```

```
>>> from sage.all import *
>>> # (OK-ish) D(x)Abs(x)
>>> #    Verify(D(x) Abs(x), Sign(x));
>>> diff(abs(x))
1/2*(x + conjugate(x))/abs(x)
>>> _.simplify_full()
x/abs(x)
>>> _ = var('x', domain='real')
>>> diff(abs(x))
x/abs(x)
>>> forget()
```

```
sage: # (YES) (Integrate(x)Abs(x))=Abs(x)*x/2
sage: integral(abs(x), x)
1/2*x*abs(x)
```

```
>>> from sage.all import *
>>> # (YES) (Integrate(x)Abs(x))=Abs(x)*x/2
>>> integral(abs(x), x)
1/2*x*abs(x)
```

```
sage: #  (YES) Compute derivative of Abs(x), piecewise defined.
sage: #     Verify(D(x)if(x<0) (-x) else x,
sage: #        Simplify(if(x<0) -1 else 1))
Piecewise defined function with 2 parts, [[(-10, 0), -1], [(0, 10), 1]]
sage: #  (NOT really) Integrate Abs(x), piecewise defined.
sage: #      Verify(Simplify(Integrate(x)
sage: #        if(x<0) (-x) else x),
sage: #        Simplify(if(x<0) (-x^2/2) else x^2/2));
sage: f = piecewise([ ((-10,0), -x), ((0,10), x)])
sage: f.integral(definite=True)
100
```

```
>>> from sage.all import *
>>> #  (YES) Compute derivative of Abs(x), piecewise defined.
>>> #     Verify(D(x)if(x<0) (-x) else x,
```

(continues on next page)

(continued from previous page)

```
>>> #        Simplify(if(x<0) -1 else 1))
Piecewise defined function with 2 parts, [[(-10, 0), -1], [(0, 10), 1]]
>>> #  (NOT really) Integrate Abs(x), piecewise defined.
>>> #      Verify(Simplify(Integrate(x)
>>> #        if(x<0) (-x) else x),
>>> #        Simplify(if(x<0) (-x^2/2) else x^2/2));
>>> f = piecewise([ ((-Integer(10),Integer(0)), -x), ((Integer(0),Integer(10)), x)])
>>> f.integral(definite=True)
100
```

```
sage: # (YES) Taylor series of 1/Sqrt(1-v^2/c^2) at v=0.
sage: var('v,c')
(v, c)
sage: taylor(1/sqrt(1-v^2/c^2), v, 0, 7)
1/2*v^2/c^2 + 3/8*v^4/c^4 + 5/16*v^6/c^6 + 1
```

```
>>> from sage.all import *
>>> # (YES) Taylor series of 1/Sqrt(1-v^2/c^2) at v=0.
>>> var('v,c')
(v, c)
>>> taylor(Integer(1)/sqrt(Integer(1)-v**Integer(2)/c**Integer(2)), v, Integer(0),␣
↪Integer(7))
1/2*v^2/c^2 + 3/8*v^4/c^4 + 5/16*v^6/c^6 + 1
```

```
sage: # (OK-ish) (Taylor expansion of Sin(x))/(Taylor expansion of Cos(x)) = (Taylor␣
↪expansion of Tan(x)).
sage: #      TestYacas(Taylor(x,0,5)(Taylor(x,0,5)Sin(x))/
sage: #        (Taylor(x,0,5)Cos(x)), Taylor(x,0,5)Tan(x));
sage: f = taylor(sin(x), x, 0, 8)
sage: g = taylor(cos(x), x, 0, 8)
sage: h = taylor(tan(x), x, 0, 8)
sage: f = f.power_series(QQ)
sage: g = g.power_series(QQ)
sage: h = h.power_series(QQ)
sage: f - g*h
O(x^8)
```

```
>>> from sage.all import *
>>> # (OK-ish) (Taylor expansion of Sin(x))/(Taylor expansion of Cos(x)) = (Taylor␣
↪expansion of Tan(x)).
>>> #      TestYacas(Taylor(x,0,5)(Taylor(x,0,5)Sin(x))/
>>> #        (Taylor(x,0,5)Cos(x)), Taylor(x,0,5)Tan(x));
>>> f = taylor(sin(x), x, Integer(0), Integer(8))
>>> g = taylor(cos(x), x, Integer(0), Integer(8))
>>> h = taylor(tan(x), x, Integer(0), Integer(8))
>>> f = f.power_series(QQ)
>>> g = g.power_series(QQ)
>>> h = h.power_series(QQ)
>>> f - g*h
O(x^8)
```

```
sage: # (YES) Taylor expansion of Ln(x)^a*Exp(-b*x) at x=1.
sage: a,b = var('a,b')
sage: taylor(log(x)^a*exp(-b*x), x, 1, 3)
-1/48*(a^3*(x - 1)^a + a^2*(6*b + 5)*(x - 1)^a + 8*b^3*(x - 1)^a + 2*(6*b^2 + 5*b +
↪3)*a*(x - 1)^a)*(x - 1)^3*e^(-b) + 1/24*(3*a^2*(x - 1)^a + a*(12*b + 5)*(x - 1)^a +
↪12*b^2*(x - 1)^a)*(x - 1)^2*e^(-b) - 1/2*(a*(x - 1)^a + 2*b*(x - 1)^a)*(x - 1)*e^(-
↪b) + (x - 1)^a*e^(-b)
```

```
>>> from sage.all import *
>>> # (YES) Taylor expansion of Ln(x)^a*Exp(-b*x) at x=1.
>>> a,b = var('a,b')
>>> taylor(log(x)**a*exp(-b*x), x, Integer(1), Integer(3))
-1/48*(a^3*(x - 1)^a + a^2*(6*b + 5)*(x - 1)^a + 8*b^3*(x - 1)^a + 2*(6*b^2 + 5*b +
↪3)*a*(x - 1)^a)*(x - 1)^3*e^(-b) + 1/24*(3*a^2*(x - 1)^a + a*(12*b + 5)*(x - 1)^a +
↪12*b^2*(x - 1)^a)*(x - 1)^2*e^(-b) - 1/2*(a*(x - 1)^a + 2*b*(x - 1)^a)*(x - 1)*e^(-
↪b) + (x - 1)^a*e^(-b)
```

```
sage: # (YES) Taylor expansion of Ln(Sin(x)/x) at x=0.
sage: taylor(log(sin(x)/x), x, 0, 10)
-1/467775*x^10 - 1/37800*x^8 - 1/2835*x^6 - 1/180*x^4 - 1/6*x^2
```

```
>>> from sage.all import *
>>> # (YES) Taylor expansion of Ln(Sin(x)/x) at x=0.
>>> taylor(log(sin(x)/x), x, Integer(0), Integer(10))
-1/467775*x^10 - 1/37800*x^8 - 1/2835*x^6 - 1/180*x^4 - 1/6*x^2
```

```
sage: # (NO) Compute n-th term of the Taylor series of Ln(Sin(x)/x) at x=0.
sage: # need formal functions
```

```
>>> from sage.all import *
>>> # (NO) Compute n-th term of the Taylor series of Ln(Sin(x)/x) at x=0.
>>> # need formal functions
```

```
sage: # (NO) Compute n-th term of the Taylor series of Exp(-x)*Sin(x) at x=0.
sage: # (Sort of, with some work)
sage: # Solve x=Sin(y)+Cos(y) for y as Taylor series in x at x=1.
sage: #      TestYacas(InverseTaylor(y,0,4) Sin(y)+Cos(y),
sage: #        (y-1)+(y-1)^2/2+2*(y-1)^3/3+(y-1)^4);
sage: #       Note that InverseTaylor does not give the series in terms of x but in
↪terms of y which is semantically
sage: # wrong. But other CAS do the same.
sage: f = sin(y) + cos(y)
sage: g = f.taylor(y, 0, 10)
sage: h = g.power_series(QQ)
sage: k = (h - 1).reverse()
sage: k
y + 1/2*y^2 + 2/3*y^3 + y^4 + 17/10*y^5 + 37/12*y^6 + 41/7*y^7 + 23/2*y^8 + 1667/72*y^
↪9 + 3803/80*y^10 + O(y^11)
```

```
>>> from sage.all import *
>>> # (NO) Compute n-th term of the Taylor series of Exp(-x)*Sin(x) at x=0.
```

(continues on next page)

(continued from previous page)

```
>>> # (Sort of, with some work)
>>> # Solve x=Sin(y)+Cos(y) for y as Taylor series in x at x=1.
>>> #      TestYacas(InverseTaylor(y,0,4) Sin(y)+Cos(y),
>>> #        (y-1)+(y-1)^2/2+2*(y-1)^3/3+(y-1)^4);
>>> #       Note that InverseTaylor does not give the series in terms of x but in␣
↪terms of y which is semantically
>>> # wrong. But other CAS do the same.
>>> f = sin(y) + cos(y)
>>> g = f.taylor(y, Integer(0), Integer(10))
>>> h = g.power_series(QQ)
>>> k = (h - Integer(1)).reverse()
>>> k
y + 1/2*y^2 + 2/3*y^3 + y^4 + 17/10*y^5 + 37/12*y^6 + 41/7*y^7 + 23/2*y^8 + 1667/72*y^
↪9 + 3803/80*y^10 + O(y^11)
```

```
sage: # (OK) Compute Legendre polynomials directly from Rodrigues's formula, P[n]=1/
↪(2^n*n!) *(Deriv(x,n)(x^2-1)^n).
sage: #      P(n,x) := Simplify( 1/(2*n)!! *
sage: #        Deriv(x,n) (x^2-1)^n );
sage: #      TestYacas(P(4,x), (35*x^4)/8+(-15*x^2)/4+3/8);
sage: P = lambda n, x: simplify(diff((x^2-1)^n,x,n) / (2^n * factorial(n)))
sage: P(4,x).expand()
35/8*x^4 - 15/4*x^2 + 3/8
```

```
>>> from sage.all import *
>>> # (OK) Compute Legendre polynomials directly from Rodrigues's formula, P[n]=1/(2^
↪n*n!) *(Deriv(x,n)(x^2-1)^n).
>>> #      P(n,x) := Simplify( 1/(2*n)!! *
>>> #        Deriv(x,n) (x^2-1)^n );
>>> #      TestYacas(P(4,x), (35*x^4)/8+(-15*x^2)/4+3/8);
>>> P = lambda n, x: simplify(diff((x**Integer(2)-Integer(1))**n,x,n) /␣
↪(Integer(2)**n * factorial(n)))
>>> P(Integer(4),x).expand()
35/8*x^4 - 15/4*x^2 + 3/8
```

```
sage: # (YES) Define the polynomial p=Sum(i,1,5,a[i]*x^i).
sage: # symbolically
sage: ps = sum(var('a%s'%i)*x^i for i in range(1,6)); ps
a5*x^5 + a4*x^4 + a3*x^3 + a2*x^2 + a1*x
sage: ps.parent()
Symbolic Ring
sage: # algebraically
sage: R = PolynomialRing(QQ,5,names='a')
sage: S.<x> = PolynomialRing(R)
sage: p = S(list(R.gens()))*x; p
a4*x^5 + a3*x^4 + a2*x^3 + a1*x^2 + a0*x
sage: p.parent()
Univariate Polynomial Ring in x over Multivariate Polynomial Ring in a0, a1, a2, a3,␣
↪a4 over Rational Field
```

```
>>> from sage.all import *
>>> # (YES) Define the polynomial p=Sum(i,1,5,a[i]*x^i).
>>> # symbolically
>>> ps = sum(var('a%s'%i)*x**i for i in range(Integer(1),Integer(6))); ps
a5*x^5 + a4*x^4 + a3*x^3 + a2*x^2 + a1*x
>>> ps.parent()
Symbolic Ring
>>> # algebraically
>>> R = PolynomialRing(QQ,Integer(5),names='a')
>>> S = PolynomialRing(R, names=('x',)); (x,) = S._first_ngens(1)
>>> p = S(list(R.gens()))*x; p
a4*x^5 + a3*x^4 + a2*x^3 + a1*x^2 + a0*x
>>> p.parent()
Univariate Polynomial Ring in x over Multivariate Polynomial Ring in a0, a1, a2, a3,␣
↪a4 over Rational Field
```

```
sage: # (YES) Convert the above to Horner's form.
sage: #      Verify(Horner(p, x), ((((a[5]*x+a[4])*x
sage: #        +a[3])*x+a[2])*x+a[1])*x);
sage: restore('x')
sage: SR(p).horner(x)
((((a4*x + a3)*x + a2)*x + a1)*x + a0)*x
```

```
>>> from sage.all import *
>>> # (YES) Convert the above to Horner's form.
>>> #      Verify(Horner(p, x), ((((a[5]*x+a[4])*x
>>> #        +a[3])*x+a[2])*x+a[1])*x);
>>> restore('x')
>>> SR(p).horner(x)
((((a4*x + a3)*x + a2)*x + a1)*x + a0)*x
```

```
sage: # (NO) Convert the result of problem 127 to Fortran syntax.
sage: #      CForm(Horner(p, x));
```

```
>>> from sage.all import *
>>> # (NO) Convert the result of problem 127 to Fortran syntax.
>>> #      CForm(Horner(p, x));
```

```
sage: # (YES) Verify that True And False=False.
sage: (True and False) is False
True
```

```
>>> from sage.all import *
>>> # (YES) Verify that True And False=False.
>>> (True and False) is False
True
```

```
sage: # (YES) Prove x Or Not x.
sage: for x in [True, False]:
....:     print(x or (not x))
True
```

(continues on next page)

(continued from previous page)

```
True
```

```
>>> from sage.all import *
>>> # (YES) Prove x Or Not x.
>>> for x in [True, False]:
...     print(x or (not x))
True
True
```

```
sage: # (YES) Prove x Or y Or x And y=>x Or y.
sage: for x in [True, False]:
....:     for y in [True, False]:
....:         if x or y or x and y:
....:             if not (x or y):
....:                 print("failed!")
```

```
>>> from sage.all import *
>>> # (YES) Prove x Or y Or x And y=>x Or y.
>>> for x in [True, False]:
...     for y in [True, False]:
...         if x or y or x and y:
...             if not (x or y):
...                 print("failed!")
```

2.20 Solving ordinary differential equations

This file contains functions useful for solving differential equations which occur commonly in a 1st semester differential equations course. For another numerical solver see the `ode_solver()` function and the optional package Octave.

Solutions from the Maxima package can contain the three constants `_C`, `_K1`, and `_K2` where the underscore is used to distinguish them from symbolic variables that the user might have used. You can substitute values for them, and make them into accessible usable symbolic variables, for example with `var("_C")`.

Commands:

- `desolve()` – compute the "general solution" to a 1st or 2nd order ODE via Maxima
- `desolve_laplace()` – solve an ODE using Laplace transforms via Maxima. Initial conditions are optional
- `desolve_rk4()` – solve numerically an IVP for one first order equation, return list of points or plot
- `desolve_system_rk4()` – solve numerically an IVP for a system of first order equations, return list of points
- `desolve_odeint()` – solve numerically a system of first-order ordinary differential equations using `odeint()` from the module `scipy.integrate`.
- `desolve_system()` – solve a system of 1st order ODEs of any size using Maxima. Initial conditions are optional
- `eulers_method()` – approximate solution to a 1st order DE, presented as a table
- `eulers_method_2x2()` – approximate solution to a 1st order system of DEs, presented as a table
- `eulers_method_2x2_plot()` – plot the sequence of points obtained from Euler's method

The following functions require the optional package `tides`:

- `desolve_mintides()` – numerical solution of a system of 1st order ODEs via the Taylor series integrator method implemented in TIDES
- `desolve_tides_mpfr()` – arbitrary precision Taylor series integrator implemented in TIDES

AUTHORS:

- David Joyner (3-2006) - Initial version of functions
- Marshall Hampton (7-2007) - Creation of Python module and testing
- Robert Bradshaw (10-2008) - Some interface cleanup.
- Robert Marik (10-2009) - Some bugfixes and enhancements
- Miguel Marco (06-2014) - Tides desolvers

`sage.calculus.desolvers.`**`desolve`**`(`*de*, *dvar*, *ics=None*, *ivar=None*, *show_method=False*, *contrib_ode=False*, *algorithm='maxima'*`)`

Solve a 1st or 2nd order linear ODE, including IVP and BVP.

INPUT:

- `de` – an expression or equation representing the ODE
- `dvar` – the dependent variable (hereafter called y)
- `ics` – (optional) the initial or boundary conditions
  - for a first-order equation, specify the initial x and y
  - for a second-order equation, specify the initial x, y, and dy/dx, i.e. write $[x_0, y(x_0), y'(x_0)]$
  - for a second-order boundary solution, specify initial and final x and y boundary conditions, i.e. write $[x_0, y(x_0), x_1, y(x_1)]$.
  - gives an error if the solution is not SymbolicEquation (as happens for example for a Clairaut equation)
- `ivar` – (optional) the independent variable (hereafter called x), which must be specified if there is more than one independent variable in the equation
- `show_method` – (optional) if `True`, then Sage returns pair `[solution, method]`, where method is the string describing the method which has been used to get a solution (Maxima uses the following order for first order equations: linear, separable, exact (including exact with integrating factor), homogeneous, bernoulli, generalized homogeneous) - use carefully in class, see below the example of an equation which is separable but this property is not recognized by Maxima and the equation is solved as exact.
- `contrib_ode` – (optional) if `True`, `desolve` allows to solve Clairaut, Lagrange, Riccati and some other equations. This may take a long time and is thus turned off by default. Initial conditions can be used only if the result is one SymbolicEquation (does not contain a singular solution, for example).
- `algorithm` – (default: `'maxima'`) one of
  - `'maxima'` – use maxima
  - `'fricas'` – use FriCAS (the optional fricas spkg has to be installed)

OUTPUT:

In most cases return a SymbolicEquation which defines the solution implicitly. If the result is in the form $y(x) = \ldots$ (happens for linear eqs.), return the right-hand side only. The possible constant solutions of separable ODEs are omitted.

Note

Use `desolve? <tab>` if the output in the Sage notebook is truncated.

EXAMPLES:

```
sage: x = var('x')
sage: y = function('y')(x)
sage: desolve(diff(y,x) + y - 1, y)
(_C + e^x)*e^(-x)
```

```
>>> from sage.all import *
>>> x = var('x')
>>> y = function('y')(x)
>>> desolve(diff(y,x) + y - Integer(1), y)
(_C + e^x)*e^(-x)
```

```
sage: f = desolve(diff(y,x) + y - 1, y, ics=[10,2]); f
(e^10 + e^x)*e^(-x)
```

```
>>> from sage.all import *
>>> f = desolve(diff(y,x) + y - Integer(1), y, ics=[Integer(10),Integer(2)]); f
(e^10 + e^x)*e^(-x)
```

```
sage: plot(f)
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> plot(f)
Graphics object consisting of 1 graphics primitive
```

We can also solve second-order differential equations:

```
sage: x = var('x')
sage: y = function('y')(x)
sage: de = diff(y,x,2) - y == x
sage: desolve(de, y)
_K2*e^(-x) + _K1*e^x - x
```

```
>>> from sage.all import *
>>> x = var('x')
>>> y = function('y')(x)
>>> de = diff(y,x,Integer(2)) - y == x
>>> desolve(de, y)
_K2*e^(-x) + _K1*e^x - x
```

```
sage: f = desolve(de, y, [10,2,1]); f
-x + 7*e^(x - 10) + 5*e^(-x + 10)
```

```
>>> from sage.all import *
>>> f = desolve(de, y, [Integer(10),Integer(2),Integer(1)]); f
-x + 7*e^(x - 10) + 5*e^(-x + 10)
```

```
sage: f(x=10)
2
```

```
>>> from sage.all import *
>>> f(x=Integer(10))
2
```

```
sage: diff(f,x)(x=10)
1
```

```
>>> from sage.all import *
>>> diff(f,x)(x=Integer(10))
1
```

```
sage: de = diff(y,x,2) + y == 0
sage: desolve(de, y)
_K2*cos(x) + _K1*sin(x)
```

```
>>> from sage.all import *
>>> de = diff(y,x,Integer(2)) + y == Integer(0)
>>> desolve(de, y)
_K2*cos(x) + _K1*sin(x)
```

```
sage: desolve(de, y, [0,1,pi/2,4])
cos(x) + 4*sin(x)
```

```
>>> from sage.all import *
>>> desolve(de, y, [Integer(0),Integer(1),pi/Integer(2),Integer(4)])
cos(x) + 4*sin(x)
```

```
sage: desolve(y*diff(y,x)+sin(x)==0,y)
-1/2*y(x)^2 == _C - cos(x)
```

```
>>> from sage.all import *
>>> desolve(y*diff(y,x)+sin(x)==Integer(0),y)
-1/2*y(x)^2 == _C - cos(x)
```

Clairaut equation: general and singular solutions:

```
sage: desolve(diff(y,x)^2+x*diff(y,x)-y==0,y,contrib_ode=True,show_method=True)
[[y(x) == _C^2 + _C*x, y(x) == -1/4*x^2], 'clairau...']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x)**Integer(2)+x*diff(y,x)-y==Integer(0),y,contrib_ode=True,
↪show_method=True)
[[y(x) == _C^2 + _C*x, y(x) == -1/4*x^2], 'clairau...']
```

For equations involving more variables we specify an independent variable:

```
sage: a,b,c,n=var('a b c n')
sage: desolve(x^2*diff(y,x)==a+b*x^n+c*x^2*y^2,y,ivar=x,contrib_ode=True)
[[y(x) == 0, (b*x^(n - 2) + a/x^2)*c^2*u == 0]]
```

```
>>> from sage.all import *
>>> a,b,c,n=var('a b c n')
>>> desolve(x**Integer(2)*diff(y,x)==a+b*x**n+c*x**Integer(2)*y**Integer(2),y,
↪ivar=x,contrib_ode=True)
[[y(x) == 0, (b*x^(n - 2) + a/x^2)*c^2*u == 0]]
```

```
sage: desolve(x^2*diff(y,x)==a+b*x^n+c*x^2*y^2,y,ivar=x,contrib_ode=True,show_
↪method=True)
[[[y(x) == 0, (b*x^(n - 2) + a/x^2)*c^2*u == 0]], 'riccati']
```

```
>>> from sage.all import *
>>> desolve(x**Integer(2)*diff(y,x)==a+b*x**n+c*x**Integer(2)*y**Integer(2),y,
↪ivar=x,contrib_ode=True,show_method=True)
[[[y(x) == 0, (b*x^(n - 2) + a/x^2)*c^2*u == 0]], 'riccati']
```

Higher order equations, not involving independent variable:

```
sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y).expand()
1/6*y(x)^3 + _K1*y(x) == _K2 + x
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+y*(diff(y,x,Integer(1)))**Integer(3)==Integer(0),
↪y).expand()
1/6*y(x)^3 + _K1*y(x) == _K2 + x
```

```
sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y,[0,1,1,3]).expand()
1/6*y(x)^3 - 5/3*y(x) == x - 3/2
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+y*(diff(y,x,Integer(1)))**Integer(3)==Integer(0),
↪y,[Integer(0),Integer(1),Integer(1),Integer(3)]).expand()
1/6*y(x)^3 - 5/3*y(x) == x - 3/2
```

```
sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y,[0,1,1,3],show_method=True)
[1/6*y(x)^3 - 5/3*y(x) == x - 3/2, 'freeofx']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+y*(diff(y,x,Integer(1)))**Integer(3)==Integer(0),
↪y,[Integer(0),Integer(1),Integer(1),Integer(3)],show_method=True)
[1/6*y(x)^3 - 5/3*y(x) == x - 3/2, 'freeofx']
```

Separable equations - Sage returns solution in implicit form:

```
sage: desolve(diff(y,x)*sin(y) == cos(x),y)
-cos(y(x)) == _C + sin(x)
```

```
>>> from sage.all import *
>>> desolve(diff(y,x)*sin(y) == cos(x),y)
-cos(y(x)) == _C + sin(x)
```

```
sage: desolve(diff(y,x)*sin(y) == cos(x),y,show_method=True)
[-cos(y(x)) == _C + sin(x), 'separable']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x)*sin(y) == cos(x),y,show_method=True)
[-cos(y(x)) == _C + sin(x), 'separable']
```

```
sage: desolve(diff(y,x)*sin(y) == cos(x),y,[pi/2,1])
-cos(y(x)) == -cos(1) + sin(x) - 1
```

```
>>> from sage.all import *
>>> desolve(diff(y,x)*sin(y) == cos(x),y,[pi/Integer(2),Integer(1)])
-cos(y(x)) == -cos(1) + sin(x) - 1
```

Linear equation - Sage returns the expression on the right hand side only:

```
sage: desolve(diff(y,x)+(y) == cos(x),y)
1/2*((cos(x) + sin(x))*e^x + 2*_C)*e^(-x)
```

```
>>> from sage.all import *
>>> desolve(diff(y,x)+(y) == cos(x),y)
1/2*((cos(x) + sin(x))*e^x + 2*_C)*e^(-x)
```

```
sage: desolve(diff(y,x)+(y) == cos(x),y,show_method=True)
[1/2*((cos(x) + sin(x))*e^x + 2*_C)*e^(-x), 'linear']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x)+(y) == cos(x),y,show_method=True)
[1/2*((cos(x) + sin(x))*e^x + 2*_C)*e^(-x), 'linear']
```

```
sage: desolve(diff(y,x)+(y) == cos(x),y,[0,1])
1/2*(cos(x)*e^x + e^x*sin(x) + 1)*e^(-x)
```

```
>>> from sage.all import *
>>> desolve(diff(y,x)+(y) == cos(x),y,[Integer(0),Integer(1)])
1/2*(cos(x)*e^x + e^x*sin(x) + 1)*e^(-x)
```

This ODE with separated variables is solved as exact. Explanation - factor does not split e^{x-y} in Maxima into $e^x e^y$:

```
sage: desolve(diff(y,x)==exp(x-y),y,show_method=True)
[-e^x + e^y(x) == _C, 'exact']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x)==exp(x-y),y,show_method=True)
[-e^x + e^y(x) == _C, 'exact']
```

You can solve Bessel equations, also using initial conditions, but you cannot put (sometimes desired) the initial condition at $x = 0$, since this point is a singular point of the equation. Anyway, if the solution should be bounded at $x = 0$, then `_K2=0`.

```
sage: desolve(x^2*diff(y,x,x)+x*diff(y,x)+(x^2-4)*y==0,y)
_K1*bessel_J(2, x) + _K2*bessel_Y(2, x)
```

```
>>> from sage.all import *
>>> desolve(x**Integer(2)*diff(y,x,x)+x*diff(y,x)+(x**Integer(2)-
↪Integer(4))*y==Integer(0),y)
_K1*bessel_J(2, x) + _K2*bessel_Y(2, x)
```

Example of difficult ODE producing an error:

```
sage: desolve(sqrt(y)*diff(y,x)+e^(y)+cos(x)-sin(x+y)==0,y) # not tested
Traceback (click to the left for traceback)
...
NotImplementedError, "Maxima was unable to solve this ODE. Consider to set option␣
↪contrib_ode to True."
```

```
>>> from sage.all import *
>>> desolve(sqrt(y)*diff(y,x)+e**(y)+cos(x)-sin(x+y)==Integer(0),y) # not tested
Traceback (click to the left for traceback)
...
NotImplementedError, "Maxima was unable to solve this ODE. Consider to set option␣
↪contrib_ode to True."
```

Another difficult ODE with error - moreover, it takes a long time:

```
sage: desolve(sqrt(y)*diff(y,x)+e^(y)+cos(x)-sin(x+y)==0,y,contrib_ode=True) #␣
↪not tested
```

```
>>> from sage.all import *
>>> desolve(sqrt(y)*diff(y,x)+e**(y)+cos(x)-sin(x+y)==Integer(0),y,contrib_
↪ode=True) # not tested
```

Some more types of ODEs:

```
sage: desolve(x*diff(y,x)^2-(1+x*y)*diff(y,x)+y==0,y,contrib_ode=True,show_
↪method=True)
[[y(x) == _C + log(x), y(x) == _C*e^x], 'factor']
```

```
>>> from sage.all import *
>>> desolve(x*diff(y,x)**Integer(2)-(Integer(1)+x*y)*diff(y,x)+y==Integer(0),y,
↪contrib_ode=True,show_method=True)
[[y(x) == _C + log(x), y(x) == _C*e^x], 'factor']
```

```
sage: desolve(diff(y,x)==(x+y)^2,y,contrib_ode=True,show_method=True)
[[[x == _C - arctan(sqrt(t)), y(x) == -x - sqrt(t)], [x == _C + arctan(sqrt(t)),␣
↪y(x) == -x + sqrt(t)]], 'lagrange']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x)==(x+y)**Integer(2),y,contrib_ode=True,show_method=True)
[[[x == _C - arctan(sqrt(t)), y(x) == -x - sqrt(t)], [x == _C + arctan(sqrt(t)),␣
↪y(x) == -x + sqrt(t)]], 'lagrange']
```

These two examples produce an error (as expected, Maxima 5.18 cannot solve equations from initial conditions). Maxima 5.18 returns false answer in this case!

```
sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y,[0,1,2]).expand() # not tested
Traceback (click to the left for traceback)
...
NotImplementedError, "Maxima was unable to solve this ODE. Consider to set option␣
↪contrib_ode to True."
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+y*(diff(y,x,Integer(1)))**Integer(3)==Integer(0),
↪y,[Integer(0),Integer(1),Integer(2)]).expand() # not tested
Traceback (click to the left for traceback)
...
NotImplementedError, "Maxima was unable to solve this ODE. Consider to set option␣
↪contrib_ode to True."
```

```
sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y,[0,1,2],show_method=True) # not␣
↪tested
Traceback (click to the left for traceback)
...
NotImplementedError, "Maxima was unable to solve this ODE. Consider to set option␣
↪contrib_ode to True."
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+y*(diff(y,x,Integer(1)))**Integer(3)==Integer(0),
↪y,[Integer(0),Integer(1),Integer(2)],show_method=True) # not tested
Traceback (click to the left for traceback)
...
NotImplementedError, "Maxima was unable to solve this ODE. Consider to set option␣
↪contrib_ode to True."
```

Second order linear ODE:

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y)
(_K2*x + _K1)*e^(-x) + 1/2*sin(x)
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == cos(x),y)
(_K2*x + _K1)*e^(-x) + 1/2*sin(x)
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,show_method=True)
[(_K2*x + _K1)*e^(-x) + 1/2*sin(x), 'variationofparameters']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == cos(x),y,show_
```

(continues on next page)

(continued from previous page)

```
↪method=True)
[(_K2*x + _K1)*e^(-x) + 1/2*sin(x), 'variationofparameters']
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,[0,3,1])
1/2*(7*x + 6)*e^(-x) + 1/2*sin(x)
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == cos(x),y,[Integer(0),
↪Integer(3),Integer(1)])
1/2*(7*x + 6)*e^(-x) + 1/2*sin(x)
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,[0,3,1],show_method=True)
[1/2*(7*x + 6)*e^(-x) + 1/2*sin(x), 'variationofparameters']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == cos(x),y,[Integer(0),
↪Integer(3),Integer(1)],show_method=True)
[1/2*(7*x + 6)*e^(-x) + 1/2*sin(x), 'variationofparameters']
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,[0,3,pi/2,2])
3*(x*(e^(1/2*pi) - 2)/pi + 1)*e^(-x) + 1/2*sin(x)
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == cos(x),y,[Integer(0),
↪Integer(3),pi/Integer(2),Integer(2)])
3*(x*(e^(1/2*pi) - 2)/pi + 1)*e^(-x) + 1/2*sin(x)
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,[0,3,pi/2,2],show_method=True)
[3*(x*(e^(1/2*pi) - 2)/pi + 1)*e^(-x) + 1/2*sin(x), 'variationofparameters']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == cos(x),y,[Integer(0),
↪Integer(3),pi/Integer(2),Integer(2)],show_method=True)
[3*(x*(e^(1/2*pi) - 2)/pi + 1)*e^(-x) + 1/2*sin(x), 'variationofparameters']
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y)
(_K2*x + _K1)*e^(-x)
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == Integer(0),y)
(_K2*x + _K1)*e^(-x)
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,show_method=True)
[(_K2*x + _K1)*e^(-x), 'constcoeff']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == Integer(0),y,show_
↪method=True)
[(_K2*x + _K1)*e^(-x), 'constcoeff']
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,[0,3,1])
(4*x + 3)*e^(-x)
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == Integer(0),y,
↪[Integer(0),Integer(3),Integer(1)])
(4*x + 3)*e^(-x)
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,[0,3,1],show_method=True)
[(4*x + 3)*e^(-x), 'constcoeff']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == Integer(0),y,
↪[Integer(0),Integer(3),Integer(1)],show_method=True)
[(4*x + 3)*e^(-x), 'constcoeff']
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,[0,3,pi/2,2])
(2*x*(2*e^(1/2*pi) - 3)/pi + 3)*e^(-x)
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == Integer(0),y,
↪[Integer(0),Integer(3),pi/Integer(2),Integer(2)])
(2*x*(2*e^(1/2*pi) - 3)/pi + 3)*e^(-x)
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,[0,3,pi/2,2],show_method=True)
[(2*x*(2*e^(1/2*pi) - 3)/pi + 3)*e^(-x), 'constcoeff']
```

```
>>> from sage.all import *
>>> desolve(diff(y,x,Integer(2))+Integer(2)*diff(y,x)+y == Integer(0),y,
↪[Integer(0),Integer(3),pi/Integer(2),Integer(2)],show_method=True)
[(2*x*(2*e^(1/2*pi) - 3)/pi + 3)*e^(-x), 'constcoeff']
```

Using `algorithm='fricas'` we can invoke the differential equation solver from FriCAS. For example, it can solve higher order linear equations:

```
sage: de = x^3*diff(y, x, 3) + x^2*diff(y, x, 2) - 2*x*diff(y, x) + 2*y - 2*x^4
sage: Y = desolve(de, y, algorithm='fricas'); Y             # optional - fricas
(2*x^3 - 3*x^2 + 1)*_C0/x + (x^3 - 1)*_C1/x
 + (x^3 - 3*x^2 - 1)*_C2/x + 1/15*(x^5 - 10*x^3 + 20*x^2 + 4)/x
```

```
>>> from sage.all import *
>>> de = x**Integer(3)*diff(y, x, Integer(3)) + x**Integer(2)*diff(y, x,␣
↪Integer(2)) - Integer(2)*x*diff(y, x) + Integer(2)*y - Integer(2)*x**Integer(4)
>>> Y = desolve(de, y, algorithm='fricas'); Y             # optional - fricas
(2*x^3 - 3*x^2 + 1)*_C0/x + (x^3 - 1)*_C1/x
 + (x^3 - 3*x^2 - 1)*_C2/x + 1/15*(x^5 - 10*x^3 + 20*x^2 + 4)/x
```

The initial conditions are then interpreted as $[x_0, y(x_0), y'(x_0), \ldots, y^(n)(x_0)]$:

```
sage: Y = desolve(de, y, ics=[1,3,7], algorithm='fricas'); Y  # optional - fricas
1/15*(x^5 + 15*x^3 + 50*x^2 - 21)/x
```

```
>>> from sage.all import *
>>> Y = desolve(de, y, ics=[Integer(1),Integer(3),Integer(7)], algorithm='fricas
↪'); Y  # optional - fricas
1/15*(x^5 + 15*x^3 + 50*x^2 - 21)/x
```

FriCAS can also solve some non-linear equations:

```
sage: de = diff(y, x) == y / (x+y*log(y))
sage: Y = desolve(de, y, algorithm='fricas'); Y            # optional - fricas
1/2*(log(y(x))^2*y(x) - 2*x)/y(x)
```

```
>>> from sage.all import *
>>> de = diff(y, x) == y / (x+y*log(y))
>>> Y = desolve(de, y, algorithm='fricas'); Y            # optional - fricas
1/2*(log(y(x))^2*y(x) - 2*x)/y(x)
```

AUTHORS:

- David Joyner (1-2006)
- Robert Bradshaw (10-2008)
- Robert Marik (10-2009)

sage.calculus.desolvers.**desolve_laplace**(*de*, *dvar*, *ics=None*, *ivar=None*)

Solve an ODE using Laplace transforms. Initial conditions are optional.

INPUT:

- `de` – a lambda expression representing the ODE (e.g. `de = diff(y,x,2) == diff(y,x)+sin(x)`)
- `dvar` – the dependent variable (e.g. `y`)
- `ivar` – (optional) the independent variable (hereafter called x), which must be specified if there is more than one independent variable in the equation.
- `ics` – list of numbers representing initial conditions, (e.g. `f(0)=1`, `f'(0)=2` corresponds to `ics = [0,1, 2]`)

OUTPUT: solution of the ODE as symbolic expression

EXAMPLES:

```
sage: u=function('u')(x)
sage: eq = diff(u,x) - exp(-x) - u == 0
sage: desolve_laplace(eq,u)
1/2*(2*u(0) + 1)*e^x - 1/2*e^(-x)
```

```
>>> from sage.all import *
>>> u=function('u')(x)
>>> eq = diff(u,x) - exp(-x) - u == Integer(0)
>>> desolve_laplace(eq,u)
1/2*(2*u(0) + 1)*e^x - 1/2*e^(-x)
```

We can use initial conditions:

```
sage: desolve_laplace(eq,u,ics=[0,3])
-1/2*e^(-x) + 7/2*e^x
```

```
>>> from sage.all import *
>>> desolve_laplace(eq,u,ics=[Integer(0),Integer(3)])
-1/2*e^(-x) + 7/2*e^x
```

The initial conditions do not persist in the system (as they persisted in previous versions):

```
sage: desolve_laplace(eq,u)
1/2*(2*u(0) + 1)*e^x - 1/2*e^(-x)
```

```
>>> from sage.all import *
>>> desolve_laplace(eq,u)
1/2*(2*u(0) + 1)*e^x - 1/2*e^(-x)
```

```
sage: f=function('f')(x)
sage: eq = diff(f,x) + f == 0
sage: desolve_laplace(eq,f,[0,1])
e^(-x)
```

```
>>> from sage.all import *
>>> f=function('f')(x)
>>> eq = diff(f,x) + f == Integer(0)
>>> desolve_laplace(eq,f,[Integer(0),Integer(1)])
e^(-x)
```

```
sage: x = var('x')
sage: f = function('f')(x)
sage: de = diff(f,x,x) - 2*diff(f,x) + f
sage: desolve_laplace(de,f)
-x*e^x*f(0) + x*e^x*D[0](f)(0) + e^x*f(0)
```

```
>>> from sage.all import *
>>> x = var('x')
>>> f = function('f')(x)
>>> de = diff(f,x,x) - Integer(2)*diff(f,x) + f
>>> desolve_laplace(de,f)
-x*e^x*f(0) + x*e^x*D[0](f)(0) + e^x*f(0)
```

```
sage: desolve_laplace(de,f,ics=[0,1,2])
x*e^x + e^x
```

```
>>> from sage.all import *
>>> desolve_laplace(de,f,ics=[Integer(0),Integer(1),Integer(2)])
x*e^x + e^x
```

AUTHORS:

- David Joyner (1-2006,8-2007)
- Robert Marik (10-2009)

`sage.calculus.desolvers.`**`desolve_mintides`**(*f*, *ics*, *initial*, *final*, *delta*, *tolrel=1e-16*, *tolabs=1e-16*)

Solve numerically a system of first order differential equations using the taylor series integrator implemented in mintides.

INPUT:

- `f` – symbolic function; its first argument will be the independent variable, . Its output should be de derivatives of the dependent variables.
- `ics` – list or tuple with the initial conditions
- `initial` – the starting value for the independent variable
- `final` – the final value for the independent value
- `delta` – the size of the steps in the output
- `tolrel` – the relative tolerance for the method
- `tolabs` – the absolute tolerance for the method

OUTPUT: list with the positions of the IVP

EXAMPLES:

We integrate a periodic orbit of the Kepler problem along 50 periods:

```
sage: var('t,x,y,X,Y')
(t, x, y, X, Y)
sage: f(t,x,y,X,Y)=[X, Y, -x/(x^2+y^2)^(3/2), -y/(x^2+y^2)^(3/2)]
sage: ics = [0.8, 0, 0, 1.22474487139159]
sage: t = 100*pi
sage: sol = desolve_mintides(f, ics, 0, t, t, 1e-12, 1e-12) # optional -tides
sage: sol # optional -tides # abs tol 1e-5
[[0.000000000000000,
0.800000000000000,
0.000000000000000,
0.000000000000000,
1.22474487139159],
[314.159265358979,
0.800000000028622,
-5.91973525754241e-9,
7.56887091890590e-9,
1.22474487136329]]
```

```
>>> from sage.all import *
>>> var('t,x,y,X,Y')
(t, x, y, X, Y)
>>> __tmp__=var("t,x,y,X,Y"); f = symbolic_expression([X, Y, -x/
↪(x**Integer(2)+y**Integer(2))**(Integer(3)/Integer(2)), -y/
↪(x**Integer(2)+y**Integer(2))**(Integer(3)/Integer(2))]).function(t,x,y,X,Y)
>>> ics = [RealNumber('0.8'), Integer(0), Integer(0), RealNumber('1.22474487139159
↪')]
>>> t = Integer(100)*pi
>>> sol = desolve_mintides(f, ics, Integer(0), t, t, RealNumber('1e-12'),␣
↪RealNumber('1e-12')) # optional -tides
>>> sol # optional -tides # abs tol 1e-5
[[0.000000000000000,
0.800000000000000,
0.000000000000000,
0.000000000000000,
1.22474487139159],
```

(continues on next page)

(continued from previous page)

```
[314.159265358979,
0.800000000028622,
-5.91973525754241e-9,
7.56887091890590e-9,
1.22474487136329]]
```

ALGORITHM:

Uses TIDES.

REFERENCES:

- A. Abad, R. Barrio, F. Blesa, M. Rodriguez. Algorithm 924. *ACM Transactions on Mathematical Software* , *39* (1), 1-28.
- A. Abad, R. Barrio, F. Blesa, M. Rodriguez. TIDES tutorial: Integrating ODEs by using the Taylor Series Method.

sage.calculus.desolvers.**desolve_odeint**(*des*, *ics*, *times*, *dvars*, *ivar=None*, *compute_jac=False*, *args=()*, *rtol=None*, *atol=None*, *tcrit=None*, *h0=0.0*, *hmax=0.0*, *hmin=0.0*, *ixpr=0*, *mxstep=0*, *mxhnil=0*, *mxordn=12*, *mxords=5*, *printmessg=0*)

Solve numerically a system of first-order ordinary differential equations using `scipy.integrate.odeint()`.

INPUT:

- `des` – right hand sides of the system
- `ics` – initial conditions
- `times` – a sequence of time points in which the solution must be found
- `dvars` – dependent variables. ATTENTION: the order must be the same as in `des`, that means: `d(dvars[i])/dt=des[i]`
- `ivar` – independent variable, optional
- `compute_jac` – boolean (default: `False`); if `True`, the Jacobian of `des` is computed and used during the integration of stiff systems

Other Parameters (taken from the documentation of the `odeint()` function from `scipy.integrate`):

- `rtol`, `atol` : float The input parameters `rtol` and `atol` determine the error control performed by the solver. The solver will control the vector, e, of estimated local errors in y, according to an inequality of the form:

  max-norm of (e / ewt) <= 1

  where ewt is a vector of positive error weights computed as:

  ewt = rtol * abs(y) + atol

  `rtol` and `atol` can be either vectors the same length as y or scalars.
- `tcrit` : array Vector of critical points (e.g. singularities) where integration care should be taken.
- `h0` : float, (0: solver-determined) The step size to be attempted on the first step.
- `hmax` : float, (0: solver-determined) The maximum absolute step size allowed.
- `hmin` : float, (0: solver-determined) The minimum absolute step size allowed.
- `ixpr` : boolean. Whether to generate extra printing at method switches.

- `mxstep` : integer, (0: solver-determined) Maximum number of (internally defined) steps allowed for each integration point in t.
- `mxhnil` : integer, (0: solver-determined) Maximum number of messages printed.
- `mxordn` : integer, (0: solver-determined) Maximum order to be allowed for the nonstiff (Adams) method.
- `mxords` : integer, (0: solver-determined) Maximum order to be allowed for the stiff (BDF) method.

OUTPUT: a list with the solution of the system at each time in `times`

EXAMPLES:

Lotka Volterra Equations:

```
sage: from sage.calculus.desolvers import desolve_odeint
sage: x,y = var('x,y')
sage: f = [x*(1-y), -y*(1-x)]
sage: sol = desolve_odeint(f, [0.5,2], srange(0,10,0.1), [x,y])             #␣
↪needs scipy
sage: p = line(zip(sol[:,0],sol[:,1]))                                       #␣
↪needs scipy sage.plot
sage: p.show()                                                               #␣
↪needs scipy sage.plot
```

```
>>> from sage.all import *
>>> from sage.calculus.desolvers import desolve_odeint
>>> x,y = var('x,y')
>>> f = [x*(Integer(1)-y), -y*(Integer(1)-x)]
>>> sol = desolve_odeint(f, [RealNumber('0.5'),Integer(2)], srange(Integer(0),
↪Integer(10),RealNumber('0.1')), [x,y])                 # needs scipy
>>> p = line(zip(sol[:,Integer(0)],sol[:,Integer(1)]))                        ␣
↪                # needs scipy sage.plot
>>> p.show()                                                                #␣
↪needs scipy sage.plot
```

Lorenz Equations:

```
sage: x,y,z = var('x,y,z')
sage: # Next we define the parameters
sage: sigma = 10
sage: rho = 28
sage: beta = 8/3
sage: # The Lorenz equations
sage: lorenz = [sigma*(y-x),x*(rho-z)-y,x*y-beta*z]
sage: # Time and initial conditions
sage: times = srange(0,50.05,0.05)
sage: ics = [0,1,1]
sage: sol = desolve_odeint(lorenz, ics, times, [x,y,z],                      #␣
↪needs scipy
....:                       rtol=1e-13, atol=1e-14)
```

```
>>> from sage.all import *
>>> x,y,z = var('x,y,z')
>>> # Next we define the parameters
>>> sigma = Integer(10)
```

(continues on next page)

(continued from previous page)

```
>>> rho = Integer(28)
>>> beta = Integer(8)/Integer(3)
>>> # The Lorenz equations
>>> lorenz = [sigma*(y-x),x*(rho-z)-y,x*y-beta*z]
>>> # Time and initial conditions
>>> times = srange(Integer(0),RealNumber('50.05'),RealNumber('0.05'))
>>> ics = [Integer(0),Integer(1),Integer(1)]
>>> sol = desolve_odeint(lorenz, ics, times, [x,y,z],                    #␣
↪needs scipy
...                         rtol=RealNumber('1e-13'), atol=RealNumber('1e-14'))
```

One-dimensional stiff system:

```
sage: y = var('y')
sage: epsilon = 0.01
sage: f = y^2*(1-y)
sage: ic = epsilon
sage: t = srange(0,2/epsilon,1)
sage: sol = desolve_odeint(f, ic, t, y,                                  #␣
↪needs scipy
....:                         rtol=1e-9, atol=1e-10, compute_jac=True)
sage: p = points(zip(t,sol[:,0]))                                        #␣
↪needs scipy sage.plot
sage: p.show()                                                           #␣
↪needs scipy sage.plot
```

```
>>> from sage.all import *
>>> y = var('y')
>>> epsilon = RealNumber('0.01')
>>> f = y**Integer(2)*(Integer(1)-y)
>>> ic = epsilon
>>> t = srange(Integer(0),Integer(2)/epsilon,Integer(1))
>>> sol = desolve_odeint(f, ic, t, y,                                   #␣
↪needs scipy
...                         rtol=RealNumber('1e-9'), atol=RealNumber('1e-10'),␣
↪compute_jac=True)
>>> p = points(zip(t,sol[:,Integer(0)]))                                 ␣
↪      # needs scipy sage.plot
>>> p.show()                                                            #␣
↪needs scipy sage.plot
```

Another stiff system with some optional parameters with no default value:

```
sage: y1,y2,y3 = var('y1,y2,y3')
sage: f1 = 77.27*(y2+y1*(1-8.375*1e-6*y1-y2))
sage: f2 = 1/77.27*(y3-(1+y1)*y2)
sage: f3 = 0.16*(y1-y3)
sage: f = [f1,f2,f3]
sage: ci = [0.2,0.4,0.7]
sage: t = srange(0,10,0.01)
sage: v = [y1,y2,y3]
sage: sol = desolve_odeint(f, ci, t, v, rtol=1e-3, atol=1e-4,            #␣
```

(continues on next page)

(continued from previous page)

```
↪needs scipy
....:                         h0=0.1, hmax=1, hmin=1e-4, mxstep=1000, mxords=17)
```

```
>>> from sage.all import *
>>> y1,y2,y3 = var('y1,y2,y3')
>>> f1 = RealNumber('77.27')*(y2+y1*(Integer(1)-RealNumber('8.375')*RealNumber(
↪'1e-6')*y1-y2))
>>> f2 = Integer(1)/RealNumber('77.27')*(y3-(Integer(1)+y1)*y2)
>>> f3 = RealNumber('0.16')*(y1-y3)
>>> f = [f1,f2,f3]
>>> ci = [RealNumber('0.2'),RealNumber('0.4'),RealNumber('0.7')]
>>> t = srange(Integer(0),Integer(10),RealNumber('0.01'))
>>> v = [y1,y2,y3]
>>> sol = desolve_odeint(f, ci, t, v, rtol=RealNumber('1e-3'), atol=RealNumber(
↪'1e-4'),                       # needs scipy
...                         h0=RealNumber('0.1'), hmax=Integer(1), hmin=RealNumber(
↪'1e-4'), mxstep=Integer(1000), mxords=Integer(17))
```

AUTHOR:

- Oriol Castejon (05-2010)

`sage.calculus.desolvers.`**`desolve_rk4`**`(`*de*, *dvar*, *ics=None*, *ivar=None*, *end_points=None*, *step=0.1*, *output='list'*, ***kwds*`)`

Solve numerically one first-order ordinary differential equation.

INPUT:

Input is similar to `desolve` command. The differential equation can be written in a form close to the `plot_slope_field` or `desolve` command.

- Variant 1 (function in two variables)
  - `de` – right hand side, i.e. the function $f(x,y)$ from ODE $y' = f(x,y)$
  - `dvar` – dependent variable (symbolic variable declared by var)
- Variant 2 (symbolic equation)
  - `de` – equation, including term with `diff(y,x)`
  - `dvar` – dependent variable (declared as function of independent variable)
- Other parameters
  - `ivar` – should be specified, if there are more variables or if the equation is autonomous
  - `ics` – initial conditions in the form `[x0,y0]`
  - `end_points` – the end points of the interval
    * if `end_points` is a or [a], we integrate between `min(ics[0],a)` and `max(ics[0],a)`
    * if `end_points` is None, we use `end_points=ics[0]+10`
    * if end_points is [a,b] we integrate between `min(ics[0], a)` and `max(ics[0], b)`
  - `step` – (default: 0.1) the length of the step (positive number)
  - `output` – (default: `'list'`) one of `'list'`, `'plot'`, `'slope_field'` (graph of the solution with slope field)

OUTPUT:

Return a list of points, or plot produced by `list_plot`, optionally with slope field.

See also

ode_solver().

EXAMPLES:

```
sage: from sage.calculus.desolvers import desolve_rk4
```

```
>>> from sage.all import *
>>> from sage.calculus.desolvers import desolve_rk4
```

Variant 2 for input - more common in numerics:

```
sage: x,y = var('x,y')
sage: desolve_rk4(x*y*(2-y),y,ics=[0,1],end_points=1,step=0.5)
[[0, 1], [0.5, 1.12419127424558], [1.0, 1.46159016228882...]]
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> desolve_rk4(x*y*(Integer(2)-y),y,ics=[Integer(0),Integer(1)],end_
↪points=Integer(1),step=RealNumber('0.5'))
[[0, 1], [0.5, 1.12419127424558], [1.0, 1.46159016228882...]]
```

Variant 1 for input - we can pass ODE in the form used by desolve function In this example we integrate backwards, since `end_points < ics[0]`:

```
sage: y = function('y')(x)
sage: desolve_rk4(diff(y,x)+y*(y-1) == x-2,y,ics=[1,1],step=0.5, end_points=0)
[[0.0, 8.904257108962112], [0.5, 1.90932794536153...], [1, 1]]
```

```
>>> from sage.all import *
>>> y = function('y')(x)
>>> desolve_rk4(diff(y,x)+y*(y-Integer(1)) == x-Integer(2),y,ics=[Integer(1),
↪Integer(1)],step=RealNumber('0.5'), end_points=Integer(0))
[[0.0, 8.904257108962112], [0.5, 1.90932794536153...], [1, 1]]
```

Here we show how to plot simple pictures. For more advanced applications use list_plot instead. To see the resulting picture use `show(P)` in Sage notebook.

```
sage: x,y = var('x,y')
sage: P=desolve_rk4(y*(2-y),y,ics=[0,.1],ivar=x,output='slope_field',end_points=[-
↪4,6],thickness=3)
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> P=desolve_rk4(y*(Integer(2)-y),y,ics=[Integer(0),RealNumber('.1')],ivar=x,
↪output='slope_field',end_points=[-Integer(4),Integer(6)],thickness=Integer(3))
```

ALGORITHM:

4-th order Runge-Kutta method. Wrapper for command `rk` in Maxima's dynamics package. Perhaps could be faster by using fast_float instead.

AUTHORS:

- Robert Marik (10-2009)

`sage.calculus.desolvers.`**`desolve_rk4_determine_bounds`**`(`*ics*, *end_points=None*`)`

Used to determine bounds for numerical integration.

- If `end_points` is None, the interval for integration is from `ics[0]` to `ics[0]+10`
- If `end_points` is `a` or `[a]`, the interval for integration is from `min(ics[0],a)` to `max(ics[0],a)`
- If `end_points` is `[a,b]`, the interval for integration is from `min(ics[0],a)` to `max(ics[0],b)`

EXAMPLES:

```
sage: from sage.calculus.desolvers import desolve_rk4_determine_bounds
sage: desolve_rk4_determine_bounds([0,2],1)
(0, 1)
```

```
>>> from sage.all import *
>>> from sage.calculus.desolvers import desolve_rk4_determine_bounds
>>> desolve_rk4_determine_bounds([Integer(0),Integer(2)],Integer(1))
(0, 1)
```

```
sage: desolve_rk4_determine_bounds([0,2])
(0, 10)
```

```
>>> from sage.all import *
>>> desolve_rk4_determine_bounds([Integer(0),Integer(2)])
(0, 10)
```

```
sage: desolve_rk4_determine_bounds([0,2],[-2])
(-2, 0)
```

```
>>> from sage.all import *
>>> desolve_rk4_determine_bounds([Integer(0),Integer(2)],[-Integer(2)])
(-2, 0)
```

```
sage: desolve_rk4_determine_bounds([0,2],[-2,4])
(-2, 4)
```

```
>>> from sage.all import *
>>> desolve_rk4_determine_bounds([Integer(0),Integer(2)],[-Integer(2),Integer(4)])
(-2, 4)
```

`sage.calculus.desolvers.`**`desolve_system`**`(`*des*, *vars*, *ics=None*, *ivar=None*, *algorithm='maxima'*`)`

Solve a system of any size of 1st order ODEs. Initial conditions are optional.

One dimensional systems are passed to `desolve_laplace()`.

INPUT:

- `des` – list of ODEs

- `vars` – list of dependent variables
- `ics` – (optional) list of initial values for `ivar` and `vars`; if `ics` is defined, it should provide initial conditions for each variable, otherwise an exception would be raised
- `ivar` – (optional) the independent variable, which must be specified if there is more than one independent variable in the equation
- `algorithm` – (default: `'maxima'`) one of
  - `'maxima'` – use maxima
  - `'fricas'` – use FriCAS (the optional fricas spkg has to be installed)

EXAMPLES:

```
sage: t = var('t')
sage: x = function('x')(t)
sage: y = function('y')(t)
sage: de1 = diff(x,t) + y - 1 == 0
sage: de2 = diff(y,t) - x + 1 == 0
sage: desolve_system([de1, de2], [x,y])
[x(t) == (x(0) - 1)*cos(t) - (y(0) - 1)*sin(t) + 1,
 y(t) == (y(0) - 1)*cos(t) + (x(0) - 1)*sin(t) + 1]
```

```
>>> from sage.all import *
>>> t = var('t')
>>> x = function('x')(t)
>>> y = function('y')(t)
>>> de1 = diff(x,t) + y - Integer(1) == Integer(0)
>>> de2 = diff(y,t) - x + Integer(1) == Integer(0)
>>> desolve_system([de1, de2], [x,y])
[x(t) == (x(0) - 1)*cos(t) - (y(0) - 1)*sin(t) + 1,
 y(t) == (y(0) - 1)*cos(t) + (x(0) - 1)*sin(t) + 1]
```

The same system solved using FriCAS:

```
sage: desolve_system([de1, de2], [x,y], algorithm='fricas')  # optional - fricas
[x(t) == _C0*cos(t) + cos(t)^2 + _C1*sin(t) + sin(t)^2,
 y(t) == -_C1*cos(t) + _C0*sin(t) + 1]
```

```
>>> from sage.all import *
>>> desolve_system([de1, de2], [x,y], algorithm='fricas')  # optional - fricas
[x(t) == _C0*cos(t) + cos(t)^2 + _C1*sin(t) + sin(t)^2,
 y(t) == -_C1*cos(t) + _C0*sin(t) + 1]
```

Now we give some initial conditions:

```
sage: sol = desolve_system([de1, de2], [x,y], ics=[0,1,2]); sol
[x(t) == -sin(t) + 1, y(t) == cos(t) + 1]
```

```
>>> from sage.all import *
>>> sol = desolve_system([de1, de2], [x,y], ics=[Integer(0),Integer(1),
↪Integer(2)]); sol
[x(t) == -sin(t) + 1, y(t) == cos(t) + 1]
```

```
sage: solnx, solny = sol[0].rhs(), sol[1].rhs()
sage: plot([solnx,solny],(0,1))   # not tested
sage: parametric_plot((solnx,solny),(0,1))   # not tested
```

```
>>> from sage.all import *
>>> solnx, solny = sol[Integer(0)].rhs(), sol[Integer(1)].rhs()
>>> plot([solnx,solny],(Integer(0),Integer(1)))   # not tested
>>> parametric_plot((solnx,solny),(Integer(0),Integer(1)))   # not tested
```

AUTHORS:

- Robert Bradshaw (10-2008)
- Sergey Bykov (10-2014)

sage.calculus.desolvers.**desolve_system_rk4**(*des*, *vars*, *ics=None*, *ivar=None*, *end_points=None*, *step=0.1*)

Solve numerically a system of first-order ordinary differential equations using the 4-th order Runge-Kutta method. Wrapper for Maxima command `rk`.

INPUT:

Input is similar to `desolve_system` and `desolve_rk4` commands

- `des` – right hand sides of the system
- `vars` – dependent variables
- `ivar` – (optional) should be specified, if there are more variables or if the equation is autonomous and the independent variable is missing
- `ics` – initial conditions in the form `[x0,y01,y02,y03,....]`
- `end_points` – the end points of the interval
  - if `end_points` is a or [a], we integrate on between `min(ics[0], a)` and `max(ics[0], a)`
  - if `end_points` is None, we use `end_points=ics[0]+10`
  - if `end_points` is [a,b] we integrate on between `min(ics[0], a)` and `max(ics[0], b)`
- `step` – (default: 0.1) the length of the step

OUTPUT: a list of points

See also

ode_solver().

EXAMPLES:

```
sage: from sage.calculus.desolvers import desolve_system_rk4
```

```
>>> from sage.all import *
>>> from sage.calculus.desolvers import desolve_system_rk4
```

Lotka Volterra system:

```
sage: from sage.calculus.desolvers import desolve_system_rk4
sage: x,y,t = var('x y t')
sage: P = desolve_system_rk4([x*(1-y),-y*(1-x)], [x,y], ics=[0,0.5,2],
....:                         ivar=t, end_points=20)
sage: Q = [[i,j] for i,j,k in P]
sage: LP = list_plot(Q)                                                  #␣
↪needs sage.plot

sage: Q = [[j,k] for i,j,k in P]
sage: LP = list_plot(Q)                                                  #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.calculus.desolvers import desolve_system_rk4
>>> x,y,t = var('x y t')
>>> P = desolve_system_rk4([x*(Integer(1)-y),-y*(Integer(1)-x)], [x,y],␣
↪ics=[Integer(0),RealNumber('0.5'),Integer(2)],
...                         ivar=t, end_points=Integer(20))
>>> Q = [[i,j] for i,j,k in P]
>>> LP = list_plot(Q)                                                    #␣
↪needs sage.plot

>>> Q = [[j,k] for i,j,k in P]
>>> LP = list_plot(Q)                                                    #␣
↪needs sage.plot
```

ALGORITHM:

4-th order Runge-Kutta method. Wrapper for command `rk` in Maxima's dynamics package. Perhaps could be faster by using `fast_float` instead.

AUTHOR:

- Robert Marik (10-2009)

`sage.calculus.desolvers.`**`desolve_tides_mpfr`**(*f*, *ics*, *initial*, *final*, *delta*, *tolrel=1e-16*, *tolabs=1e-16*, *digits=50*)

Solve numerically a system of first order differential equations using the taylor series integrator in arbitrary precision implemented in tides.

INPUT:

- `f` – symbolic function; its first argument will be the independent variable. Its output should be de derivatives of the dependent variables.
- `ics` – list or tuple with the initial conditions
- `initial` – the starting value for the independent variable
- `final` – the final value for the independent value
- `delta` – the size of the steps in the output
- `tolrel` – the relative tolerance for the method
- `tolabs` – the absolute tolerance for the method
- `digits` – the digits of precision used in the computation

OUTPUT: list with the positions of the IVP

EXAMPLES:

We integrate the Lorenz equations with Saltzman values for the parameters along 10 periodic orbits with 100 digits of precision:

```
sage: var('t,x,y,z')
(t, x, y, z)
sage: s = 10
sage: r = 28
sage: b = 8/3
sage: f(t,x,y,z)= [s*(y-x),x*(r-z)-y,x*y-b*z]
sage: x0 = -13.7636106821342005250144010543616538641008648540923684535378642921202
↪82774726811585294023934639503828 4
sage: y0 = -19.5787519424517955388380414460095588661142400534276438649791334295426
↪35474614752641597316550670467617 1
sage: z0 = 27
sage: T = 15.5865221071617472756787020921269607052848054899724393588952157831901 98
↪7562588808543558510826601423 74
sage: sol = desolve_tides_mpfr(f, [x0, y0, z0], 0, T, T, 1e-100, 1e-100, 100) #␣
↪optional - tides
sage: sol # optional -tides # abs tol 1e-50
[[0.000000000000000000000000000000000000000000000000000000000000000000000000000000
↪000000000000000000000,
-13.763610682134200525014401054361653864100864854092368453537864292120282774726811
↪5852940239346395038,
-19.578751942451795538838041446009558866114240053427643864979133429542635474614752
↪6415973165506704676,
27.0000000000000000000000000000000000000000000000000000000000000000000000000000000
↪000000000000000000],
[15.586522107161747275678702092126960705284805489972439358895215783190198756258880
↪8543558510826601424,
-13.763610682134200525014401054361653864100864854092368453537864292120282774726811
↪5852940239346315658,
-19.578751942451795538838041446009558866114240053427643864979133429542635474614752
↪6415973165506778440,
26.9999999999999999999999999999999999999999999999999999999999999999999999999999999
↪999999999999636628]]
```

```
>>> from sage.all import *
>>> var('t,x,y,z')
(t, x, y, z)
>>> s = Integer(10)
>>> r = Integer(28)
>>> b = Integer(8)/Integer(3)
>>> __tmp__=var("t,x,y,z"); f = symbolic_expression([s*(y-x),x*(r-z)-y,x*y-b*z]).
↪function(t,x,y,z)
>>> x0 = -RealNumber('13.763610682134200525014401054361653864100864854092368453537
↪864292120282774726811585294023934639503828 4')
>>> y0 = -RealNumber('19.578751942451795538838041446009558866114240053427643864979
↪133429542635474614752641597316550670467617 1')
>>> z0 = Integer(27)
>>> T = RealNumber('15.586522107161747275678702092126960705284805489972439358895 21
```

(continues on next page)

(continued from previous page)

```
↪578319019875625888085435585108266014237 4')
>>> sol = desolve_tides_mpfr(f, [x0, y0, z0], Integer(0), T, T, RealNumber('1e-100
↪'), RealNumber('1e-100'), Integer(100)) # optional - tides
>>> sol # optional -tides # abs tol 1e-50
[[0.000000000000000000000000000000000000000000000000000000000000000000000000000000
↪000000000000000000000,
-13.763610682134200525014401054361653864100864854092368453537864292120282774726811
↪5852940239346395038,
-19.578751942451795538838041446009558866114240053427643864979133429542635474614752
↪6415973165506704676,
27.0000000000000000000000000000000000000000000000000000000000000000000000000000000
↪000000000000000000],
[15.586522107161747275678702092126960705284805489972439358895215783190198756258880
↪8543558510826601424,
-13.763610682134200525014401054361653864100864854092368453537864292120282774726811
↪5852940239346315658,
-19.578751942451795538838041446009558866114240053427643864979133429542635474614752
↪6415973165506778440,
26.9999999999999999999999999999999999999999999999999999999999999999999999999999999
↪999999999999636628]]
```

ALGORITHM:

Uses TIDES.

> **Warning**
>
> This requires the package tides.

REFERENCES:

- [ABBR2011]
- [ABBR2012]

`sage.calculus.desolvers.`**`eulers_method`**`(`*f*, *x0*, *y0*, *h*, *x1*, *algorithm='table'*`)`

This implements Euler's method for finding numerically the solution of the 1st order ODE $y' = f(x, y)$, $y(a) = c$. The `x` column of the table increments from x_0 to x_1 by h (so $(x_1 - x_0)/h$ must be an integer). In the `y` column, the new y-value equals the old y-value plus the corresponding entry in the last column.

> **Note**
>
> This function is for pedagogical purposes only.

EXAMPLES:

```
sage: from sage.calculus.desolvers import eulers_method
sage: x,y = PolynomialRing(QQ,2,"xy").gens()
sage: eulers_method(5*x+y-5,0,1,1/2,1)
     x                    y                  h*f(x,y)
     0                    1                   -2
```

(continues on next page)

(continued from previous page)

```
  1/2                   -1                   -7/4
    1                 -11/4                 -11/8
```

```
>>> from sage.all import *
>>> from sage.calculus.desolvers import eulers_method
>>> x,y = PolynomialRing(QQ,Integer(2),"xy").gens()
>>> eulers_method(Integer(5)*x+y-Integer(5),Integer(0),Integer(1),Integer(1)/
↪Integer(2),Integer(1))
     x                    y                  h*f(x,y)
     0                    1                    -2
   1/2                   -1                   -7/4
     1                 -11/4                 -11/8
```

```
sage: x,y = PolynomialRing(QQ,2,"xy").gens()
sage: eulers_method(5*x+y-5,0,1,1/2,1,algorithm='none')
[[0, 1], [1/2, -1], [1, -11/4], [3/2, -33/8]]
```

```
>>> from sage.all import *
>>> x,y = PolynomialRing(QQ,Integer(2),"xy").gens()
>>> eulers_method(Integer(5)*x+y-Integer(5),Integer(0),Integer(1),Integer(1)/
↪Integer(2),Integer(1),algorithm='none')
[[0, 1], [1/2, -1], [1, -11/4], [3/2, -33/8]]
```

```
sage: RR = RealField(sci_not=0, prec=4, rnd='RNDU')
sage: x,y = PolynomialRing(RR,2,"xy").gens()
sage: eulers_method(5*x+y-5,0,1,1/2,1,algorithm="None")
[[0, 1], [1/2, -1.0], [1, -2.7], [3/2, -4.0]]
```

```
>>> from sage.all import *
>>> RR = RealField(sci_not=Integer(0), prec=Integer(4), rnd='RNDU')
>>> x,y = PolynomialRing(RR,Integer(2),"xy").gens()
>>> eulers_method(Integer(5)*x+y-Integer(5),Integer(0),Integer(1),Integer(1)/
↪Integer(2),Integer(1),algorithm="None")
[[0, 1], [1/2, -1.0], [1, -2.7], [3/2, -4.0]]
```

```
sage: RR = RealField(sci_not=0, prec=4, rnd='RNDU')
sage: x,y=PolynomialRing(RR,2,"xy").gens()
sage: eulers_method(5*x+y-5,0,1,1/2,1)
     x                    y                  h*f(x,y)
     0                    1                  -2.0
   1/2                  -1.0                 -1.7
     1                  -2.7                 -1.3
```

```
>>> from sage.all import *
>>> RR = RealField(sci_not=Integer(0), prec=Integer(4), rnd='RNDU')
>>> x,y=PolynomialRing(RR,Integer(2),"xy").gens()
>>> eulers_method(Integer(5)*x+y-Integer(5),Integer(0),Integer(1),Integer(1)/
↪Integer(2),Integer(1))
     x                    y                  h*f(x,y)
     0                    1                  -2.0
```

(continues on next page)

(continued from previous page)

```
  1/2                   -1.0                    -1.7
    1                   -2.7                    -1.3
```

```
sage: x,y=PolynomialRing(QQ,2,"xy").gens()
sage: eulers_method(5*x+y-5,1,1,1/3,2)
         x                    y                  h*f(x,y)
         1                    1                   1/3
       4/3                  4/3                     1
       5/3                  7/3                  17/9
         2                 38/9                 83/27
```

```
>>> from sage.all import *
>>> x,y=PolynomialRing(QQ,Integer(2),"xy").gens()
>>> eulers_method(Integer(5)*x+y-Integer(5),Integer(1),Integer(1),Integer(1)/
↪Integer(3),Integer(2))
         x                    y                  h*f(x,y)
         1                    1                   1/3
       4/3                  4/3                     1
       5/3                  7/3                  17/9
         2                 38/9                 83/27
```

```
sage: eulers_method(5*x+y-5,0,1,1/2,1,algorithm='none')
[[0, 1], [1/2, -1], [1, -11/4], [3/2, -33/8]]
```

```
>>> from sage.all import *
>>> eulers_method(Integer(5)*x+y-Integer(5),Integer(0),Integer(1),Integer(1)/
↪Integer(2),Integer(1),algorithm='none')
[[0, 1], [1/2, -1], [1, -11/4], [3/2, -33/8]]
```

```
sage: pts = eulers_method(5*x+y-5,0,1,1/2,1,algorithm='none')
sage: P1 = list_plot(pts)                                                   #␣
↪needs sage.plot
sage: P2 = line(pts)                                                        #␣
↪needs sage.plot
sage: (P1 + P2).show()                                                      #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> pts = eulers_method(Integer(5)*x+y-Integer(5),Integer(0),Integer(1),
↪Integer(1)/Integer(2),Integer(1),algorithm='none')
>>> P1 = list_plot(pts)                                                   #␣
↪needs sage.plot
>>> P2 = line(pts)                                                        #␣
↪needs sage.plot
>>> (P1 + P2).show()                                                      #␣
↪needs sage.plot
```

AUTHORS:

- David Joyner

sage.calculus.desolvers.**eulers_method_2x2**(*f*, *g*, *t0*, *x0*, *y0*, *h*, *t1*, *algorithm='table'*)

This implements Euler's method for finding numerically the solution of the 1st order system of two ODEs

$$x' = f(t, x, y), x(t_0) = x_0$$
$$y' = g(t, x, y), y(t_0) = y_0.$$

The `t` column of the table increments from t_0 to t_1 by h (so $\frac{t_1 - t_0}{h}$ must be an integer). In the `x` column, the new x-value equals the old x-value plus the corresponding entry in the next (third) column. In the `y` column, the new y-value equals the old y-value plus the corresponding entry in the next (last) column.

Note

This function is for pedagogical purposes only.

EXAMPLES:

```
sage: from sage.calculus.desolvers import eulers_method_2x2
sage: t, x, y = PolynomialRing(QQ,3,"txy").gens()
sage: f = x+y+t; g = x-y
sage: eulers_method_2x2(f,g, 0, 0, 0, 1/3, 1,algorithm='none')
[[0, 0, 0], [1/3, 0, 0], [2/3, 1/9, 0], [1, 10/27, 1/27], [4/3, 68/81, 4/27]]
```

```
>>> from sage.all import *
>>> from sage.calculus.desolvers import eulers_method_2x2
>>> t, x, y = PolynomialRing(QQ,Integer(3),"txy").gens()
>>> f = x+y+t; g = x-y
>>> eulers_method_2x2(f,g, Integer(0), Integer(0), Integer(0), Integer(1)/
↪Integer(3), Integer(1),algorithm='none')
[[0, 0, 0], [1/3, 0, 0], [2/3, 1/9, 0], [1, 10/27, 1/27], [4/3, 68/81, 4/27]]
```

```
sage: eulers_method_2x2(f,g, 0, 0, 0, 1/3, 1)
     t                    x                h*f(t,x,y)                    y       ␣
↪    h*g(t,x,y)
     0                    0                         0                    0       ␣
↪             0
   1/3                    0                       1/9                    0       ␣
↪             0
   2/3                  1/9                      7/27                    0       ␣
↪          1/27
     1                10/27                     38/81                 1/27       ␣
↪           1/9
```

```
>>> from sage.all import *
>>> eulers_method_2x2(f,g, Integer(0), Integer(0), Integer(0), Integer(1)/
↪Integer(3), Integer(1))
     t                    x                h*f(t,x,y)                    y       ␣
↪    h*g(t,x,y)
     0                    0                         0                    0       ␣
↪             0
   1/3                    0                       1/9                    0       ␣
↪             0
   2/3                  1/9                      7/27                    0       ␣
```

(continues on next page)

(continued from previous page)

```
↪         1/27
    1              10/27                     38/81                 1/27        ␣
↪          1/9
```

```
sage: RR = RealField(sci_not=0, prec=4, rnd='RNDU')
sage: t,x,y=PolynomialRing(RR,3,"txy").gens()
sage: f = x+y+t; g = x-y
sage: eulers_method_2x2(f,g, 0, 0, 0, 1/3, 1)
     t                    x                h*f(t,x,y)                    y        ␣
↪    h*g(t,x,y)
     0                    0                      0.00                    0        ␣
↪         0.00
   1/3                 0.00                      0.13                 0.00        ␣
↪         0.00
   2/3                 0.13                      0.29                 0.00        ␣
↪        0.043
     1                 0.41                      0.57                0.043        ␣
↪         0.15
```

```
>>> from sage.all import *
>>> RR = RealField(sci_not=Integer(0), prec=Integer(4), rnd='RNDU')
>>> t,x,y=PolynomialRing(RR,Integer(3),"txy").gens()
>>> f = x+y+t; g = x-y
>>> eulers_method_2x2(f,g, Integer(0), Integer(0), Integer(0), Integer(1)/
↪Integer(3), Integer(1))
     t                    x                h*f(t,x,y)                    y        ␣
↪    h*g(t,x,y)
     0                    0                      0.00                    0        ␣
↪         0.00
   1/3                 0.00                      0.13                 0.00        ␣
↪         0.00
   2/3                 0.13                      0.29                 0.00        ␣
↪        0.043
     1                 0.41                      0.57                0.043        ␣
↪         0.15
```

To numerically approximate $y(1)$, where $(1+t^2)y''+y'-y=0$, $y(0)=1$, $y'(0)=-1$, using 4 steps of Euler's method, first convert to a system: $y_1' = y_2$, $y_1(0)=1$; $y_2' = \frac{y_1-y_2}{1+t^2}$, $y_2(0)=-1$.:

```
sage: RR = RealField(sci_not=0, prec=4, rnd='RNDU')
sage: t, x, y=PolynomialRing(RR,3,"txy").gens()
sage: f = y; g = (x-y)/(1+t^2)
sage: eulers_method_2x2(f,g, 0, 1, -1, 1/4, 1)
    t                    x                h*f(t,x,y)                    y        ␣
↪   h*g(t,x,y)
    0                    1                     -0.25                   -1        ␣
↪        0.50
  1/4                 0.75                     -0.12                -0.50        ␣
↪        0.29
  1/2                 0.63                    -0.054                -0.21        ␣
↪        0.19
  3/4                 0.63                   -0.0078               -0.031        ␣
```

(continues on next page)

(continued from previous page)

```
↪         0.11
   1                  0.63                      0.020                   0.079          ␣
↪        0.071
```

```
>>> from sage.all import *
>>> RR = RealField(sci_not=Integer(0), prec=Integer(4), rnd='RNDU')
>>> t, x, y=PolynomialRing(RR,Integer(3),"txy").gens()
>>> f = y; g = (x-y)/(Integer(1)+t**Integer(2))
>>> eulers_method_2x2(f,g, Integer(0), Integer(1), -Integer(1), Integer(1)/
↪Integer(4), Integer(1))
   t                    x                h*f(t,x,y)                    y          ␣
↪  h*g(t,x,y)
   0                    1                     -0.25                   -1          ␣
↪         0.50
 1/4                  0.75                      -0.12                -0.50          ␣
↪         0.29
 1/2                  0.63                     -0.054                -0.21          ␣
↪         0.19
 3/4                  0.63                    -0.0078               -0.031          ␣
↪         0.11
   1                  0.63                      0.020                0.079          ␣
↪        0.071
```

To numerically approximate $y(1)$, where $y'' + ty' + y = 0$, $y(0) = 1$, $y'(0) = 0$:

```
sage: t,x,y=PolynomialRing(RR,3,"txy").gens()
sage: f = y; g = -x-y*t
sage: eulers_method_2x2(f,g, 0, 1, 0, 1/4, 1)
    t                    x                h*f(t,x,y)                    y          ␣
↪   h*g(t,x,y)
    0                    1                      0.00                    0          ␣
↪        -0.25
  1/4                  1.0                     -0.062                -0.25          ␣
↪        -0.23
  1/2                 0.94                      -0.11                -0.46          ␣
↪        -0.17
  3/4                 0.88                      -0.15                -0.62          ␣
↪        -0.10
    1                 0.75                      -0.17                -0.68          ␣
↪       -0.015
```

```
>>> from sage.all import *
>>> t,x,y=PolynomialRing(RR,Integer(3),"txy").gens()
>>> f = y; g = -x-y*t
>>> eulers_method_2x2(f,g, Integer(0), Integer(1), Integer(0), Integer(1)/
↪Integer(4), Integer(1))
    t                    x                h*f(t,x,y)                    y          ␣
↪   h*g(t,x,y)
    0                    1                      0.00                    0          ␣
↪        -0.25
  1/4                  1.0                     -0.062                -0.25          ␣
↪        -0.23
```

(continues on next page)

(continued from previous page)

```
  1/2                 0.94                     -0.11                   -0.46          ␣
↪          -0.17
  3/4                 0.88                     -0.15                   -0.62          ␣
↪          -0.10
    1                 0.75                     -0.17                   -0.68          ␣
↪         -0.015
```

AUTHORS:

- David Joyner

sage.calculus.desolvers.**eulers_method_2x2_plot**(*f*, *g*, *t0*, *x0*, *y0*, *h*, *t1*)

Plot solution of ODE.

This plots the solution in the rectangle with sides `(xrange[0],xrange[1])` and `(yrange[0],yrange[1])`, and plots using Euler's method the numerical solution of the 1st order ODEs $x' = f(t, x, y)$, $x(a) = x_0$, $y' = g(t, x, y)$, $y(a) = y_0$.

Note

This function is for pedagogical purposes only.

EXAMPLES:

The following example plots the solution to $\theta'' + \sin(\theta) = 0$, $\theta(0) = \frac{3}{4}$, $\theta'(0) = 0$. Type `P[0].show()` to plot the solution, `(P[0]+P[1]).show()` to plot $(t, \theta(t))$ and $(t, \theta'(t))$:

```
sage: from sage.calculus.desolvers import eulers_method_2x2_plot
sage: f = lambda z : z[2]; g = lambda z : -sin(z[1])
sage: P = eulers_method_2x2_plot(f,g, 0.0, 0.75, 0.0, 0.1, 1.0)                 #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.calculus.desolvers import eulers_method_2x2_plot
>>> f = lambda z : z[Integer(2)]; g = lambda z : -sin(z[Integer(1)])
>>> P = eulers_method_2x2_plot(f,g, RealNumber('0.0'), RealNumber('0.75'),␣
↪RealNumber('0.0'), RealNumber('0.1'), RealNumber('1.0'))             #␣
↪needs sage.plot
```

sage.calculus.desolvers.**fricas_desolve**(*de*, *dvar*, *ics*, *ivar*)

Solve an ODE using FriCAS.

EXAMPLES:

```
sage: x = var('x')
sage: y = function('y')(x)
sage: desolve(diff(y,x) + y - 1, y, algorithm='fricas')             # optional -␣
↪fricas
_C0*e^(-x) + 1

sage: desolve(diff(y, x) + y == y^3*sin(x), y, algorithm='fricas')  # optional -␣
↪fricas
-1/5*(2*cos(x)*y(x)^2 + 4*sin(x)*y(x)^2 - 5)*e^(-2*x)/y(x)^2
```

```
>>> from sage.all import *
>>> x = var('x')
>>> y = function('y')(x)
>>> desolve(diff(y,x) + y - Integer(1), y, algorithm='fricas')              #␣
↪optional - fricas
_C0*e^(-x) + 1

>>> desolve(diff(y, x) + y == y**Integer(3)*sin(x), y, algorithm='fricas')   #␣
↪optional - fricas
-1/5*(2*cos(x)*y(x)^2 + 4*sin(x)*y(x)^2 - 5)*e^(-2*x)/y(x)^2
```

sage.calculus.desolvers.**fricas_desolve_system**(*des*, *dvars*, *ics*, *ivar*)

Solve a system of first order ODEs using FriCAS.

EXAMPLES:

```
sage: t = var('t')
sage: x = function('x')(t)
sage: y = function('y')(t)
sage: de1 = diff(x,t) + y - 1 == 0
sage: de2 = diff(y,t) - x + 1 == 0
sage: desolve_system([de1, de2], [x, y], algorithm='fricas')          # optional -
↪ fricas
[x(t) == _C0*cos(t) + cos(t)^2 + _C1*sin(t) + sin(t)^2,
 y(t) == -_C1*cos(t) + _C0*sin(t) + 1]

sage: desolve_system([de1, de2], [x,y], [0,1,2], algorithm='fricas')  # optional -
↪ fricas
[x(t) == cos(t)^2 + sin(t)^2 - sin(t), y(t) == cos(t) + 1]
```

```
>>> from sage.all import *
>>> t = var('t')
>>> x = function('x')(t)
>>> y = function('y')(t)
>>> de1 = diff(x,t) + y - Integer(1) == Integer(0)
>>> de2 = diff(y,t) - x + Integer(1) == Integer(0)
>>> desolve_system([de1, de2], [x, y], algorithm='fricas')          # optional -␣
↪fricas
[x(t) == _C0*cos(t) + cos(t)^2 + _C1*sin(t) + sin(t)^2,
 y(t) == -_C1*cos(t) + _C0*sin(t) + 1]

>>> desolve_system([de1, de2], [x,y], [Integer(0),Integer(1),Integer(2)],␣
↪algorithm='fricas')  # optional - fricas
[x(t) == cos(t)^2 + sin(t)^2 - sin(t), y(t) == cos(t) + 1]
```

2.21 Discrete Wavelet Transform

Wraps `GSL's gsl_wavelet_transform_forward()`, and `gsl_wavelet_transform_inverse()` and creates plot methods.

AUTHOR:

- Josh Kantor (2006-10-07) - initial version

- David Joyner (2006-10-09) - minor changes to docstrings and examples.

`sage.calculus.transforms.dwt.DWT(n, wavelet_type, wavelet_k)`

This function initializes an GSLDoubleArray of length n which can perform a discrete wavelet transform.

INPUT:

- `n` – a power of 2
- `T` – the data in the GSLDoubleArray must be real
- `wavelet_type` – the name of the type of wavelet, valid choices are:
  - `'daubechies'`
  - `'daubechies_centered'`
  - `'haar'`
  - `'haar_centered'`
  - `'bspline'`
  - `'bspline_centered'`

For daubechies wavelets, `wavelet_k` specifies a daubechie wavelet with $k/2$ vanishing moments. $k = 4, 6, ..., 20$ for k even are the only ones implemented.

For Haar wavelets, `wavelet_k` must be 2.

For bspline wavelets, `wavelet_k` of $103, 105, 202, 204, 206, 208, 301, 305, 307, 309$ will give biorthogonal B-spline wavelets of order (i, j) where `wavelet_k` is $100 * i + j$. The wavelet transform uses $J = \log_2(n)$ levels.

OUTPUT:

An array of the form $(s_{-1,0}, d_{0,0}, d_{1,0}, d_{1,1}, d_{2,0}, \ldots, d_{J-1,2^{J-1}-1})$ for $d_{j,k}$ the detail coefficients of level j. The centered forms align the coefficients of the sub-bands on edges.

EXAMPLES:

```
sage: a = WaveletTransform(128,'daubechies',4)
sage: for i in range(1, 11):
....:     a[i] = 1
....:     a[128-i] = 1
sage: a.plot().show(ymin=0)                                                    #␣
↪needs sage.plot
sage: a.forward_transform()
sage: a.plot().show()                                                          #␣
↪needs sage.plot
sage: a = WaveletTransform(128,'haar',2)
sage: for i in range(1, 11): a[i] = 1; a[128-i] = 1
sage: a.forward_transform()
sage: a.plot().show(ymin=0)                                                    #␣
↪needs sage.plot
sage: a = WaveletTransform(128,'bspline_centered',103)
sage: for i in range(1, 11): a[i] = 1; a[100+i] = 1
sage: a.forward_transform()
sage: a.plot().show(ymin=0)                                                    #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> a = WaveletTransform(Integer(128),'daubechies',Integer(4))
>>> for i in range(Integer(1), Integer(11)):
...     a[i] = Integer(1)
...     a[Integer(128)-i] = Integer(1)
>>> a.plot().show(ymin=Integer(0))                                         ␣
↪     # needs sage.plot
>>> a.forward_transform()
>>> a.plot().show()                                                      #␣
↪needs sage.plot
>>> a = WaveletTransform(Integer(128),'haar',Integer(2))
>>> for i in range(Integer(1), Integer(11)): a[i] = Integer(1); a[Integer(128)-i]␣
↪= Integer(1)
>>> a.forward_transform()
>>> a.plot().show(ymin=Integer(0))                                         ␣
↪     # needs sage.plot
>>> a = WaveletTransform(Integer(128),'bspline_centered',Integer(103))
>>> for i in range(Integer(1), Integer(11)): a[i] = Integer(1); a[Integer(100)+i]␣
↪= Integer(1)
>>> a.forward_transform()
>>> a.plot().show(ymin=Integer(0))                                         ␣
↪     # needs sage.plot
```

This example gives a simple example of wavelet compression:

```
sage: # needs sage.symbolic
sage: a = DWT(2048,'daubechies',6)
sage: for i in range(2048): a[i]=float(sin((i*5/2048)**2))
sage: a.plot().show()                   # long time (7s on sage.math, 2011),␣
↪needs sage.plot
sage: a.forward_transform()
sage: for i in range(1800): a[2048-i-1] = 0
sage: a.backward_transform()
sage: a.plot().show()                   # long time (7s on sage.math, 2011),␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> a = DWT(Integer(2048),'daubechies',Integer(6))
>>> for i in range(Integer(2048)): a[i]=float(sin((i*Integer(5)/
↪Integer(2048))**Integer(2)))
>>> a.plot().show()                   # long time (7s on sage.math, 2011), needs␣
↪sage.plot
>>> a.forward_transform()
>>> for i in range(Integer(1800)): a[Integer(2048)-i-Integer(1)] = Integer(0)
>>> a.backward_transform()
>>> a.plot().show()                   # long time (7s on sage.math, 2011), needs␣
↪sage.plot
```

class sage.calculus.transforms.dwt.**DiscreteWaveletTransform**

Bases: GSLDoubleArray

Discrete wavelet transform class.

backward_transform()

forward_transform()

plot(*xmin=None*, *xmax=None*, ***args*)

sage.calculus.transforms.dwt.**WaveletTransform**(*n*, *wavelet_type*, *wavelet_k*)

This function initializes an GSLDoubleArray of length n which can perform a discrete wavelet transform.

INPUT:

- `n` – a power of 2
- `T` – the data in the GSLDoubleArray must be real
- `wavelet_type` – the name of the type of wavelet, valid choices are:
  - `'daubechies'`
  - `'daubechies_centered'`
  - `'haar'`
  - `'haar_centered'`
  - `'bspline'`
  - `'bspline_centered'`

For daubechies wavelets, `wavelet_k` specifies a daubechie wavelet with $k/2$ vanishing moments. $k = 4, 6, ..., 20$ for k even are the only ones implemented.

For Haar wavelets, `wavelet_k` must be 2.

For bspline wavelets, `wavelet_k` of $103, 105, 202, 204, 206, 208, 301, 305, 307, 309$ will give biorthogonal B-spline wavelets of order (i, j) where `wavelet_k` is $100 * i + j$. The wavelet transform uses $J = \log_2(n)$ levels.

OUTPUT:

An array of the form $(s_{-1,0}, d_{0,0}, d_{1,0}, d_{1,1}, d_{2,0}, \ldots, d_{J-1,2^{J-1}-1})$ for $d_{j,k}$ the detail coefficients of level j. The centered forms align the coefficients of the sub-bands on edges.

EXAMPLES:

```
sage: a = WaveletTransform(128,'daubechies',4)
sage: for i in range(1, 11):
....:     a[i] = 1
....:     a[128-i] = 1
sage: a.plot().show(ymin=0)                                                    #␣
↪needs sage.plot
sage: a.forward_transform()
sage: a.plot().show()                                                          #␣
↪needs sage.plot
sage: a = WaveletTransform(128,'haar',2)
sage: for i in range(1, 11): a[i] = 1; a[128-i] = 1
sage: a.forward_transform()
sage: a.plot().show(ymin=0)                                                    #␣
↪needs sage.plot
sage: a = WaveletTransform(128,'bspline_centered',103)
sage: for i in range(1, 11): a[i] = 1; a[100+i] = 1
sage: a.forward_transform()
```

(continues on next page)

(continued from previous page)

```
sage: a.plot().show(ymin=0)                                                   #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> a = WaveletTransform(Integer(128),'daubechies',Integer(4))
>>> for i in range(Integer(1), Integer(11)):
...     a[i] = Integer(1)
...     a[Integer(128)-i] = Integer(1)
>>> a.plot().show(ymin=Integer(0))                                             ␣
↪     # needs sage.plot
>>> a.forward_transform()
>>> a.plot().show()                                                           #␣
↪needs sage.plot
>>> a = WaveletTransform(Integer(128),'haar',Integer(2))
>>> for i in range(Integer(1), Integer(11)): a[i] = Integer(1); a[Integer(128)-i]␣
↪= Integer(1)
>>> a.forward_transform()
>>> a.plot().show(ymin=Integer(0))                                             ␣
↪     # needs sage.plot
>>> a = WaveletTransform(Integer(128),'bspline_centered',Integer(103))
>>> for i in range(Integer(1), Integer(11)): a[i] = Integer(1); a[Integer(100)+i]␣
↪= Integer(1)
>>> a.forward_transform()
>>> a.plot().show(ymin=Integer(0))                                             ␣
↪     # needs sage.plot
```

This example gives a simple example of wavelet compression:

```
sage: # needs sage.symbolic
sage: a = DWT(2048,'daubechies',6)
sage: for i in range(2048): a[i]=float(sin((i*5/2048)**2))
sage: a.plot().show()                      # long time (7s on sage.math, 2011),␣
↪needs sage.plot
sage: a.forward_transform()
sage: for i in range(1800): a[2048-i-1] = 0
sage: a.backward_transform()
sage: a.plot().show()                      # long time (7s on sage.math, 2011),␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> a = DWT(Integer(2048),'daubechies',Integer(6))
>>> for i in range(Integer(2048)): a[i]=float(sin((i*Integer(5)/
↪Integer(2048))**Integer(2)))
>>> a.plot().show()                      # long time (7s on sage.math, 2011), needs␣
↪sage.plot
>>> a.forward_transform()
>>> for i in range(Integer(1800)): a[Integer(2048)-i-Integer(1)] = Integer(0)
>>> a.backward_transform()
>>> a.plot().show()                      # long time (7s on sage.math, 2011), needs␣
↪sage.plot
```

`sage.calculus.transforms.dwt.`**`is2pow`**`(n)`

2.22 Discrete Fourier Transforms

This file contains functions useful for computing discrete Fourier transforms and probability distribution functions for discrete random variables for sequences of elements of $\mathbf{Q}$ or $\mathbf{C}$, indexed by a `range(N)`, $\mathbf{Z}/N\mathbf{Z}$, an abelian group, the conjugacy classes of a permutation group, or the conjugacy classes of a matrix group.

This file implements:

- `__eq__()`
- `__mul__()` (for right multiplication by a scalar)
- plotting, printing – *IndexedSequence.plot()*, *IndexedSequence.plot_histogram()*, `_repr_()`, `__str__()`
- `dft()` – computes the discrete Fourier transform for the following cases:
  - a sequence (over $\mathbf{Q}$ or `CyclotomicField`) indexed by `range(N)` or $\mathbf{Z}/N\mathbf{Z}$
  - a sequence (as above) indexed by a finite abelian group
  - a sequence (as above) indexed by a complete set of representatives of the conjugacy classes of a finite permutation group
  - a sequence (as above) indexed by a complete set of representatives of the conjugacy classes of a finite matrix group
- `idft()` – computes the discrete Fourier transform for the following cases:
  - a sequence (over $\mathbf{Q}$ or CyclotomicField) indexed by `range(N)` or $\mathbf{Z}/N\mathbf{Z}$
- `dct()`, `dst()` (for discrete Fourier/Cosine/Sine transform)
- convolution (in *IndexedSequence.convolution()* and *IndexedSequence.convolution_periodic()*)
- `fft()`, `ifft()` – (fast Fourier transforms) wrapping GSL's `gsl_fft_complex_forward()`, `gsl_fft_complex_inverse()`, using William Stein's *FastFourierTransform()*
- `dwt()`, `idwt()` – (fast wavelet transforms) wrapping GSL's `gsl_dwt_forward()`, `gsl_dwt_backward()` using Joshua Kantor's *WaveletTransform()* class. Allows for wavelets of type:
  - "haar"
  - "daubechies"
  - "daubechies_centered"
  - "haar_centered"
  - "bspline"
  - "bspline_centered"

Todo

- "filtered" DFTs
- more idfts
- more examples for probability, stats, theory of FTs

AUTHORS:

- David Joyner (2006-10)
- William Stein (2006-11) – fix many bugs

class sage.calculus.transforms.dft.**IndexedSequence**(*L*, *index_object*)

Bases: SageObject

An indexed sequence.

INPUT:

- L – list
- index_object – must be a Sage object with an __iter__ method containing the same number of elements as self, which is a list of elements taken from a field

base_ring()

This just returns the common parent R of the N list elements. In some applications (say, when computing the discrete Fourier transform, dft), it is more accurate to think of the base_ring as the group ring $\mathbf{Q}(\zeta_N)[R]$.

EXAMPLES:

```
sage: J = list(range(10))
sage: A = [1/10 for j in J]
sage: s = IndexedSequence(A,J)
sage: s.base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> J = list(range(Integer(10)))
>>> A = [Integer(1)/Integer(10) for j in J]
>>> s = IndexedSequence(A,J)
>>> s.base_ring()
Rational Field
```

convolution(*other*)

Convolves two sequences of the same length (automatically expands the shortest one by extending it by 0 if they have different lengths).

If $\{a_n\}$ and $\{b_n\}$ are sequences indexed by $(n = 0, 1, ..., N - 1)$, extended by zero for all n in $\mathbf{Z}$, then the convolution is

$$c_j = \sum_{i=0}^{N-1} a_i b_{j-i}.$$

INPUT:

- other – a collection of elements of a ring with index set a finite abelian group (under $+$)

OUTPUT: the Dirichlet convolution of self and other

EXAMPLES:

```
sage: J = list(range(5))
sage: A = [ZZ(1) for i in J]
sage: B = [ZZ(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: t = IndexedSequence(B,J)
sage: s.convolution(t)
[1, 2, 3, 4, 5, 4, 3, 2, 1]
```

```
>>> from sage.all import *
>>> J = list(range(Integer(5)))
>>> A = [ZZ(Integer(1)) for i in J]
>>> B = [ZZ(Integer(1)) for i in J]
>>> s = IndexedSequence(A,J)
>>> t = IndexedSequence(B,J)
>>> s.convolution(t)
[1, 2, 3, 4, 5, 4, 3, 2, 1]
```

AUTHOR: David Joyner (2006-09)

convolution_periodic(*other*)

Convolves two collections indexed by a `range(...)` of the same length (automatically expands the shortest one by extending it by 0 if they have different lengths).

If $\{a_n\}$ and $\{b_n\}$ are sequences indexed by $(n = 0, 1, ..., N - 1)$, extended periodically for all n in $\mathbf{Z}$, then the convolution is

$$c_j = \sum_{i=0}^{N-1} a_i b_{j-i}.$$

INPUT:

- `other` – a sequence of elements of $\mathbf{C}$, $\mathbf{R}$ or $\mathbf{F}_q$

OUTPUT: the Dirichlet convolution of `self` and `other`

EXAMPLES:

```
sage: I = list(range(5))
sage: A = [ZZ(1) for i in I]
sage: B = [ZZ(1) for i in I]
sage: s = IndexedSequence(A,I)
sage: t = IndexedSequence(B,I)
sage: s.convolution_periodic(t)
[5, 5, 5, 5, 5, 5, 5, 5, 5]
```

```
>>> from sage.all import *
>>> I = list(range(Integer(5)))
>>> A = [ZZ(Integer(1)) for i in I]
>>> B = [ZZ(Integer(1)) for i in I]
>>> s = IndexedSequence(A,I)
>>> t = IndexedSequence(B,I)
>>> s.convolution_periodic(t)
[5, 5, 5, 5, 5, 5, 5, 5, 5]
```

AUTHOR: David Joyner (2006-09)

dct()

A discrete Cosine transform.

EXAMPLES:

```
sage: J = list(range(5))
sage: A = [exp(-2*pi*i*I/5) for i in J]                                    #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
sage: s = IndexedSequence(A, J)                                          #␣
↪needs sage.symbolic
sage: s.dct()                                                            #␣
↪needs sage.symbolic
Indexed sequence: [0, 1/16*(sqrt(5) + I*sqrt(-2*sqrt(5) + 10) + ...
indexed by [0, 1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> J = list(range(Integer(5)))
>>> A = [exp(-Integer(2)*pi*i*I/Integer(5)) for i in J]                   ␣
↪               # needs sage.symbolic
>>> s = IndexedSequence(A, J)                                          #␣
↪needs sage.symbolic
>>> s.dct()                                                            #␣
↪needs sage.symbolic
Indexed sequence: [0, 1/16*(sqrt(5) + I*sqrt(-2*sqrt(5) + 10) + ...
indexed by [0, 1, 2, 3, 4]
```

dft(*chi=None*)

A discrete Fourier transform "over **Q**" using exact N-th roots of unity.

EXAMPLES:

```
sage: J = list(range(6))
sage: A = [ZZ(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: s.dft(lambda x: x^2)                                               #␣
↪needs sage.rings.number_field
Indexed sequence: [6, 0, 0, 6, 0, 0]
 indexed by [0, 1, 2, 3, 4, 5]
sage: s.dft()                                                            #␣
↪needs sage.rings.number_field
Indexed sequence: [6, 0, 0, 0, 0, 0]
 indexed by [0, 1, 2, 3, 4, 5]

sage: # needs sage.combinat sage.groups
sage: G = SymmetricGroup(3)
sage: J = G.conjugacy_classes_representatives()
sage: s = IndexedSequence([1,2,3], J)   # 1,2,3 are the values of a class fcn␣
↪on G
sage: s.dft()   # the "scalar-valued Fourier transform" of this class fcn
Indexed sequence: [8, 2, 2]
 indexed by [(), (1,2), (1,2,3)]

sage: # needs sage.rings.number_field
sage: J = AbelianGroup(2, [2,3], names='ab')
sage: s = IndexedSequence([1,2,3,4,5,6], J)
sage: s.dft()   # the precision of output is somewhat random and architecture␣
↪dependent.
Indexed sequence: [21.0000000000000,
                   -2.99999999999997 - 1.73205080756885*I,
                   -2.99999999999999 + 1.73205080756888*I,
```

(continues on next page)

(continued from previous page)

```
                    -9.00000000000000 + 0.00000000000000485744257349999*I,
                    -0.0000000000000976996261670137 - 0.
↪0000000000000159872115546022*I,
                    -0.0000000000000621724893790087 - 0.
↪0000000000000106581410364015*I]
 indexed by Multiplicative Abelian group isomorphic to C2 x C3

sage: # needs sage.groups sage.rings.number_field
sage: J = CyclicPermutationGroup(6)
sage: s = IndexedSequence([1,2,3,4,5,6], J)
sage: s.dft()    # the precision of output is somewhat random and architecture␣
↪dependent.
Indexed sequence: [21.0000000000000,
                    -2.99999999999997 - 1.73205080756885*I,
                    -2.99999999999999 + 1.73205080756888*I,
                    -9.00000000000000 + 0.00000000000000485744257349999*I,
                    -0.0000000000000976996261670137 - 0.
↪0000000000000159872115546022*I,
                    -0.0000000000000621724893790087 - 0.
↪0000000000000106581410364015*I]
 indexed by Cyclic group of order 6 as a permutation group

sage: # needs sage.rings.number_field
sage: p = 7; J = list(range(p)); A = [kronecker_symbol(j,p) for j in J]
sage: s = IndexedSequence(A, J)
sage: Fs = s.dft()
sage: c = Fs.list()[1]; [x/c for x in Fs.list()]; s.list()
[0, 1, 1, -1, 1, -1, -1]
[0, 1, 1, -1, 1, -1, -1]
```

```
>>> from sage.all import *
>>> J = list(range(Integer(6)))
>>> A = [ZZ(Integer(1)) for i in J]
>>> s = IndexedSequence(A,J)
>>> s.dft(lambda x: x**Integer(2))                                          ␣
↪       # needs sage.rings.number_field
Indexed sequence: [6, 0, 0, 6, 0, 0]
 indexed by [0, 1, 2, 3, 4, 5]
>>> s.dft()                                                                #␣
↪needs sage.rings.number_field
Indexed sequence: [6, 0, 0, 0, 0, 0]
 indexed by [0, 1, 2, 3, 4, 5]

>>> # needs sage.combinat sage.groups
>>> G = SymmetricGroup(Integer(3))
>>> J = G.conjugacy_classes_representatives()
>>> s = IndexedSequence([Integer(1),Integer(2),Integer(3)], J)   # 1,2,3 are␣
↪the values of a class fcn on G
>>> s.dft()    # the "scalar-valued Fourier transform" of this class fcn
Indexed sequence: [8, 2, 2]
 indexed by [(), (1,2), (1,2,3)]
```

(continues on next page)

(continued from previous page)

```
>>> # needs sage.rings.number_field
>>> J = AbelianGroup(Integer(2), [Integer(2),Integer(3)], names='ab')
>>> s = IndexedSequence([Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5),Integer(6)], J)
>>> s.dft()   # the precision of output is somewhat random and architecture␣
↪dependent.
Indexed sequence: [21.0000000000000,
                   -2.99999999999997 - 1.73205080756885*I,
                   -2.99999999999999 + 1.73205080756888*I,
                   -9.00000000000000 + 0.0000000000000485744257349999*I,
                   -0.00000000000000976996261670137 - 0.
↪0000000000000159872115546022*I,
                   -0.00000000000000621724893790087 - 0.
↪0000000000000106581410364015*I]
 indexed by Multiplicative Abelian group isomorphic to C2 x C3

>>> # needs sage.groups sage.rings.number_field
>>> J = CyclicPermutationGroup(Integer(6))
>>> s = IndexedSequence([Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5),Integer(6)], J)
>>> s.dft()   # the precision of output is somewhat random and architecture␣
↪dependent.
Indexed sequence: [21.0000000000000,
                   -2.99999999999997 - 1.73205080756885*I,
                   -2.99999999999999 + 1.73205080756888*I,
                   -9.00000000000000 + 0.0000000000000485744257349999*I,
                   -0.00000000000000976996261670137 - 0.
↪0000000000000159872115546022*I,
                   -0.00000000000000621724893790087 - 0.
↪0000000000000106581410364015*I]
 indexed by Cyclic group of order 6 as a permutation group

>>> # needs sage.rings.number_field
>>> p = Integer(7); J = list(range(p)); A = [kronecker_symbol(j,p) for j in J]
>>> s = IndexedSequence(A, J)
>>> Fs = s.dft()
>>> c = Fs.list()[Integer(1)]; [x/c for x in Fs.list()]; s.list()
[0, 1, 1, -1, 1, -1, -1]
[0, 1, 1, -1, 1, -1, -1]
```

The DFT of the values of the quadratic residue symbol is itself, up to a constant factor (denoted c on the last line above).

Todo

Read the parent of the elements of S; if **Q** or **C** leave as is; if AbelianGroup, use abelian_group_dual; if some other implemented Group (permutation, matrix), call .characters() and test if the index list is the set of conjugacy classes.

dict()

Return a python dict of `self` where the keys are elements in the indexing set.

EXAMPLES:

```
sage: J = list(range(10))
sage: A = [1/10 for j in J]
sage: s = IndexedSequence(A,J)
sage: s.dict()
{0: 1/10, 1: 1/10, 2: 1/10, 3: 1/10, 4: 1/10, 5: 1/10, 6: 1/10, 7: 1/10, 8: 1/
↪10, 9: 1/10}
```

```
>>> from sage.all import *
>>> J = list(range(Integer(10)))
>>> A = [Integer(1)/Integer(10) for j in J]
>>> s = IndexedSequence(A,J)
>>> s.dict()
{0: 1/10, 1: 1/10, 2: 1/10, 3: 1/10, 4: 1/10, 5: 1/10, 6: 1/10, 7: 1/10, 8: 1/
↪10, 9: 1/10}
```

dst()

A discrete Sine transform.

EXAMPLES:

```
sage: J = list(range(5))
sage: I = CC.0; pi = CC.pi()
sage: A = [exp(-2*pi*i*I/5) for i in J]
sage: s = IndexedSequence(A, J)

sage: s.dst()         # discrete sine
Indexed sequence: [0.000000000000000, 1.11022302462516e-16 - 2.
↪50000000000000*I, ...]
indexed by [0, 1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> J = list(range(Integer(5)))
>>> I = CC.gen(0); pi = CC.pi()
>>> A = [exp(-Integer(2)*pi*i*I/Integer(5)) for i in J]
>>> s = IndexedSequence(A, J)

>>> s.dst()         # discrete sine
Indexed sequence: [0.000000000000000, 1.11022302462516e-16 - 2.
↪50000000000000*I, ...]
indexed by [0, 1, 2, 3, 4]
```

dwt(*other='haar'*, *wavelet_k=2*)

Wraps the gsl `WaveletTransform.forward` in *dwt* (written by Joshua Kantor). Assumes the length of the sample is a power of 2. Uses the GSL function `gsl_wavelet_transform_forward()`.

INPUT:

- `other` – the name of the type of wavelet; valid choices are:
  - `'daubechies'`
  - `'daubechies_centered'`
  - `'haar'` (default)

  - 'haar_centered'
  - 'bspline'
  - 'bspline_centered'
- wavelet_k – for daubechies wavelets, wavelet_k specifies a daubechie wavelet with $k/2$ vanishing moments; $k = 4, 6, ..., 20$ for k even are the only ones implemented

  For Haar wavelets, wavelet_k must be 2.

  For bspline wavelets, wavelet_k equal to $103, 105, 202, 204, 206, 208, 301, 305, 307, 309$ will give biorthogonal B-spline wavelets of order (i, j) where wavelet_k equals $100 \cdot i + j$.

The wavelet transform uses $J = \log_2(n)$ levels.

EXAMPLES:

```
sage: J = list(range(8))
sage: A = [RR(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: t = s.dwt()
sage: t        # slightly random output
Indexed sequence: [2.82842712474999, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000]
    indexed by [0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> from sage.all import *
>>> J = list(range(Integer(8)))
>>> A = [RR(Integer(1)) for i in J]
>>> s = IndexedSequence(A,J)
>>> t = s.dwt()
>>> t        # slightly random output
Indexed sequence: [2.82842712474999, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000]
    indexed by [0, 1, 2, 3, 4, 5, 6, 7]
```

fft()

Wraps the gsl FastFourierTransform.forward() in *fft*.

If the length is a power of 2 then this automatically uses the radix2 method. If the number of sample points in the input is a power of 2 then the wrapper for the GSL function gsl_fft_complex_radix2_forward() is automatically called. Otherwise, gsl_fft_complex_forward() is used.

EXAMPLES:

```
sage: J = list(range(5))
sage: A = [RR(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: t = s.fft(); t
Indexed sequence: [5.00000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000]
    indexed by [0, 1, 2, 3, 4]
```

```
>>> from sage.all import *
>>> J = list(range(Integer(5)))
>>> A = [RR(Integer(1)) for i in J]
>>> s = IndexedSequence(A,J)
>>> t = s.fft(); t
Indexed sequence: [5.00000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000]
    indexed by [0, 1, 2, 3, 4]
```

idft()

A discrete inverse Fourier transform. Only works over **Q**.

EXAMPLES:

```
sage: J = list(range(5))
sage: A = [ZZ(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: fs = s.dft(); fs                                                      #␣
↪needs sage.rings.number_field
Indexed sequence: [5, 0, 0, 0, 0]
    indexed by [0, 1, 2, 3, 4]
sage: it = fs.idft(); it                                                    #␣
↪needs sage.rings.number_field
Indexed sequence: [1, 1, 1, 1, 1]
    indexed by [0, 1, 2, 3, 4]
sage: it == s                                                               #␣
↪needs sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> J = list(range(Integer(5)))
>>> A = [ZZ(Integer(1)) for i in J]
>>> s = IndexedSequence(A,J)
>>> fs = s.dft(); fs                                                      #␣
↪needs sage.rings.number_field
Indexed sequence: [5, 0, 0, 0, 0]
    indexed by [0, 1, 2, 3, 4]
>>> it = fs.idft(); it                                                    #␣
↪needs sage.rings.number_field
Indexed sequence: [1, 1, 1, 1, 1]
    indexed by [0, 1, 2, 3, 4]
>>> it == s                                                               #␣
↪needs sage.rings.number_field
True
```

idwt(*other='haar'*, *wavelet_k=2*)

Implement the gsl `WaveletTransform.backward()` in *dwt*.

Assumes the length of the sample is a power of 2. Uses the GSL function `gsl_wavelet_transform_backward()`.

INPUT:

- `other` – must be one of the following:

  - 'haar'
  - 'daubechies'
  - 'daubechies_centered'
  - 'haar_centered'
  - 'bspline'
  - 'bspline_centered'

- wavelet_k – for daubechies wavelets, wavelet_k specifies a daubechie wavelet with $k/2$ vanishing moments; $k = 4, 6, ..., 20$ for k even are the only ones implemented

  For Haar wavelets, wavelet_k must be 2.

  For bspline wavelets, wavelet_k equal to $103, 105, 202, 204, 206, 208, 301, 305, 307, 309$ will give biorthogonal B-spline wavelets of order (i, j) where wavelet_k equals $100 \cdot i + j$.

EXAMPLES:

```
sage: J = list(range(8))
sage: A = [RR(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: t = s.dwt()
sage: t            # random arch dependent output
Indexed sequence: [2.82842712474999, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000]
    indexed by [0, 1, 2, 3, 4, 5, 6, 7]
sage: t.idwt()                  # random arch dependent output
Indexed sequence: [1.00000000000000, 1.00000000000000, 1.00000000000000, 1.
↪00000000000000, 1.00000000000000, 1.00000000000000, 1.00000000000000, 1.
↪00000000000000]
    indexed by [0, 1, 2, 3, 4, 5, 6, 7]
sage: t.idwt() == s
True
sage: J = list(range(16))
sage: A = [RR(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: t = s.dwt("bspline", 103)
sage: t   # random arch dependent output
Indexed sequence: [4.00000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000]
    indexed by [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
sage: t.idwt("bspline", 103) == s
True
```

```
>>> from sage.all import *
>>> J = list(range(Integer(8)))
>>> A = [RR(Integer(1)) for i in J]
>>> s = IndexedSequence(A,J)
>>> t = s.dwt()
>>> t            # random arch dependent output
```

(continues on next page)

(continued from previous page)

```
Indexed sequence: [2.82842712474999, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000]
    indexed by [0, 1, 2, 3, 4, 5, 6, 7]
>>> t.idwt()                  # random arch dependent output
Indexed sequence: [1.00000000000000, 1.00000000000000, 1.00000000000000, 1.
↪00000000000000, 1.00000000000000, 1.00000000000000, 1.00000000000000, 1.
↪00000000000000]
    indexed by [0, 1, 2, 3, 4, 5, 6, 7]
>>> t.idwt() == s
True
>>> J = list(range(Integer(16)))
>>> A = [RR(Integer(1)) for i in J]
>>> s = IndexedSequence(A,J)
>>> t = s.dwt("bspline", Integer(103))
>>> t   # random arch dependent output
Indexed sequence: [4.00000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000]
    indexed by [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> t.idwt("bspline", Integer(103)) == s
True
```

ifft()

Implement the gsl `FastFourierTransform.inverse` in *fft*.

If the number of sample points in the input is a power of 2 then the wrapper for the GSL function `gsl_fft_complex_radix2_inverse()` is automatically called. Otherwise, `gsl_fft_complex_inverse()` is used.

EXAMPLES:

```
sage: J = list(range(5))
sage: A = [RR(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: t = s.fft(); t
Indexed sequence: [5.00000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000]
    indexed by [0, 1, 2, 3, 4]
sage: t.ifft()
Indexed sequence: [1.00000000000000, 1.00000000000000, 1.00000000000000, 1.
↪00000000000000, 1.00000000000000]
    indexed by [0, 1, 2, 3, 4]
sage: t.ifft() == s
1
```

```
>>> from sage.all import *
>>> J = list(range(Integer(5)))
>>> A = [RR(Integer(1)) for i in J]
>>> s = IndexedSequence(A,J)
>>> t = s.fft(); t
```

(continues on next page)

(continued from previous page)

```
Indexed sequence: [5.00000000000000, 0.000000000000000, 0.000000000000000, 0.
↪000000000000000, 0.000000000000000]
    indexed by [0, 1, 2, 3, 4]
>>> t.ifft()
Indexed sequence: [1.00000000000000, 1.00000000000000, 1.00000000000000, 1.
↪00000000000000, 1.00000000000000]
    indexed by [0, 1, 2, 3, 4]
>>> t.ifft() == s
1
```

index_object()

Return the indexing object.

EXAMPLES:

```
sage: J = list(range(10))
sage: A = [1/10 for j in J]
sage: s = IndexedSequence(A,J)
sage: s.index_object()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from sage.all import *
>>> J = list(range(Integer(10)))
>>> A = [Integer(1)/Integer(10) for j in J]
>>> s = IndexedSequence(A,J)
>>> s.index_object()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

list()

Return the list of `self`.

EXAMPLES:

```
sage: J = list(range(10))
sage: A = [1/10 for j in J]
sage: s = IndexedSequence(A,J)
sage: s.list()
[1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10]
```

```
>>> from sage.all import *
>>> J = list(range(Integer(10)))
>>> A = [Integer(1)/Integer(10) for j in J]
>>> s = IndexedSequence(A,J)
>>> s.list()
[1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10, 1/10]
```

plot()

Plot the points of the sequence.

Elements of the sequence are assumed to be real or from a finite field, with a real indexing set `I = range(len(self))`.

EXAMPLES:

```
sage: I = list(range(3))
sage: A = [ZZ(i^2)+1 for i in I]
sage: s = IndexedSequence(A,I)
sage: P = s.plot()                                                    #␣
↪needs sage.plot
sage: show(P)                        # not tested                     #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> I = list(range(Integer(3)))
>>> A = [ZZ(i**Integer(2))+Integer(1) for i in I]
>>> s = IndexedSequence(A,I)
>>> P = s.plot()                                                      #␣
↪needs sage.plot
>>> show(P)                          # not tested                     #␣
↪needs sage.plot
```

plot_histogram(*clr=(0, 0, 1)*, *eps=0.4*)

Plot the histogram plot of the sequence.

The sequence is assumed to be real or from a finite field, with a real indexing set `I` coercible into **R**.

Options are `clr`, which is an RGB value, and `eps`, which is the spacing between the bars.

EXAMPLES:

```
sage: J = list(range(3))
sage: A = [ZZ(i^2)+1 for i in J]
sage: s = IndexedSequence(A,J)
sage: P = s.plot_histogram()                                          #␣
↪needs sage.plot
sage: show(P)                        # not tested                     #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> J = list(range(Integer(3)))
>>> A = [ZZ(i**Integer(2))+Integer(1) for i in J]
>>> s = IndexedSequence(A,J)
>>> P = s.plot_histogram()                                            #␣
↪needs sage.plot
>>> show(P)                          # not tested                     #␣
↪needs sage.plot
```

2.23 Fast Fourier Transforms Using GSL

AUTHORS:

- William Stein (2006-9): initial file (radix2)
- D. Joyner (2006-10): Minor modifications (from radix2 to general case and some documentation).
- M. Hansen (2013-3): Fix radix2 backwards transformation
- L.F. Tabera Alonso (2013-3): Documentation

sage.calculus.transforms.fft.**FFT**(*size*, *base_ring=None*)

Create an array for fast Fourier transform conversion using gsl.

INPUT:

- `size` – the size of the array
- `base_ring` – unused (2013-03)

EXAMPLES:

We create an array of the desired size:

```
sage: a = FastFourierTransform(8)
sage: a
[(0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.
↪0), (0.0, 0.0)]
```

```
>>> from sage.all import *
>>> a = FastFourierTransform(Integer(8))
>>> a
[(0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.
↪0), (0.0, 0.0)]
```

Now, set the values of the array:

```
sage: for i in range(8): a[i] = i + 1
sage: a
[(1.0, 0.0), (2.0, 0.0), (3.0, 0.0), (4.0, 0.0), (5.0, 0.0), (6.0, 0.0), (7.0, 0.
↪0), (8.0, 0.0)]
```

```
>>> from sage.all import *
>>> for i in range(Integer(8)): a[i] = i + Integer(1)
>>> a
[(1.0, 0.0), (2.0, 0.0), (3.0, 0.0), (4.0, 0.0), (5.0, 0.0), (6.0, 0.0), (7.0, 0.
↪0), (8.0, 0.0)]
```

We can perform the forward Fourier transform on the array:

```
sage: a.forward_transform()
sage: a                             #abs tol 1e-2
[(36.0, 0.0), (-4.00, 9.65), (-4.0, 4.0), (-4.0, 1.65), (-4.0, 0.0), (-4.0, -1.
↪65), (-4.0, -4.0), (-4.0, -9.65)]
```

```
>>> from sage.all import *
>>> a.forward_transform()
>>> a                             #abs tol 1e-2
[(36.0, 0.0), (-4.00, 9.65), (-4.0, 4.0), (-4.0, 1.65), (-4.0, 0.0), (-4.0, -1.
↪65), (-4.0, -4.0), (-4.0, -9.65)]
```

And backwards:

```
sage: a.backward_transform()
sage: a                             #abs tol 1e-2
[(8.0, 0.0), (16.0, 0.0), (24.0, 0.0), (32.0, 0.0), (40.0, 0.0), (48.0, 0.0), (56.
↪0, 0.0), (64.0, 0.0)]
```

```
>>> from sage.all import *
>>> a.backward_transform()
>>> a                        #abs tol 1e-2
[(8.0, 0.0), (16.0, 0.0), (24.0, 0.0), (32.0, 0.0), (40.0, 0.0), (48.0, 0.0), (56.
↪0, 0.0), (64.0, 0.0)]
```

Other example:

```
sage: a = FastFourierTransform(128)
sage: for i in range(1, 11):
....:     a[i] = 1
....:     a[128-i] = 1
sage: a[:6:2]
[(0.0, 0.0), (1.0, 0.0), (1.0, 0.0)]
sage: a.plot().show(ymin=0)                                                    #␣
↪needs sage.plot
sage: a.forward_transform()
sage: a.plot().show()                                                          #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> a = FastFourierTransform(Integer(128))
>>> for i in range(Integer(1), Integer(11)):
...     a[i] = Integer(1)
...     a[Integer(128)-i] = Integer(1)
>>> a[:Integer(6):Integer(2)]
[(0.0, 0.0), (1.0, 0.0), (1.0, 0.0)]
>>> a.plot().show(ymin=Integer(0))                                              ␣
↪     # needs sage.plot
>>> a.forward_transform()
>>> a.plot().show()                                                          #␣
↪needs sage.plot
```

sage.calculus.transforms.fft.**FastFourierTransform**(*size*, *base_ring=None*)

Create an array for fast Fourier transform conversion using gsl.

INPUT:

- `size` – the size of the array
- `base_ring` – unused (2013-03)

EXAMPLES:

We create an array of the desired size:

```
sage: a = FastFourierTransform(8)
sage: a
[(0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.
↪0), (0.0, 0.0)]
```

```
>>> from sage.all import *
>>> a = FastFourierTransform(Integer(8))
>>> a
```

(continues on next page)

(continued from previous page)

```
[(0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), (0.0, 0.
↪0), (0.0, 0.0)]
```

Now, set the values of the array:

```
sage: for i in range(8): a[i] = i + 1
sage: a
[(1.0, 0.0), (2.0, 0.0), (3.0, 0.0), (4.0, 0.0), (5.0, 0.0), (6.0, 0.0), (7.0, 0.
↪0), (8.0, 0.0)]
```

```
>>> from sage.all import *
>>> for i in range(Integer(8)): a[i] = i + Integer(1)
>>> a
[(1.0, 0.0), (2.0, 0.0), (3.0, 0.0), (4.0, 0.0), (5.0, 0.0), (6.0, 0.0), (7.0, 0.
↪0), (8.0, 0.0)]
```

We can perform the forward Fourier transform on the array:

```
sage: a.forward_transform()
sage: a                             #abs tol 1e-2
[(36.0, 0.0), (-4.00, 9.65), (-4.0, 4.0), (-4.0, 1.65), (-4.0, 0.0), (-4.0, -1.
↪65), (-4.0, -4.0), (-4.0, -9.65)]
```

```
>>> from sage.all import *
>>> a.forward_transform()
>>> a                             #abs tol 1e-2
[(36.0, 0.0), (-4.00, 9.65), (-4.0, 4.0), (-4.0, 1.65), (-4.0, 0.0), (-4.0, -1.
↪65), (-4.0, -4.0), (-4.0, -9.65)]
```

And backwards:

```
sage: a.backward_transform()
sage: a                             #abs tol 1e-2
[(8.0, 0.0), (16.0, 0.0), (24.0, 0.0), (32.0, 0.0), (40.0, 0.0), (48.0, 0.0), (56.
↪0, 0.0), (64.0, 0.0)]
```

```
>>> from sage.all import *
>>> a.backward_transform()
>>> a                             #abs tol 1e-2
[(8.0, 0.0), (16.0, 0.0), (24.0, 0.0), (32.0, 0.0), (40.0, 0.0), (48.0, 0.0), (56.
↪0, 0.0), (64.0, 0.0)]
```

Other example:

```
sage: a = FastFourierTransform(128)
sage: for i in range(1, 11):
....:     a[i] = 1
....:     a[128-i] = 1
sage: a[:6:2]
[(0.0, 0.0), (1.0, 0.0), (1.0, 0.0)]
sage: a.plot().show(ymin=0)                                                    #␣
↪needs sage.plot
```

(continues on next page)

(continued from previous page)

```
sage: a.forward_transform()
sage: a.plot().show()                                                        #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> a = FastFourierTransform(Integer(128))
>>> for i in range(Integer(1), Integer(11)):
...     a[i] = Integer(1)
...     a[Integer(128)-i] = Integer(1)
>>> a[:Integer(6):Integer(2)]
[(0.0, 0.0), (1.0, 0.0), (1.0, 0.0)]
>>> a.plot().show(ymin=Integer(0))                                            ␣
↪     # needs sage.plot
>>> a.forward_transform()
>>> a.plot().show()                                                        #␣
↪needs sage.plot
```

class sage.calculus.transforms.fft.**FastFourierTransform_base**

Bases: `object`

class sage.calculus.transforms.fft.**FastFourierTransform_complex**

Bases: *FastFourierTransform_base*

Wrapper class for GSL's fast Fourier transform.

backward_transform()

Compute the in-place backwards Fourier transform of this data using the Cooley-Tukey algorithm.

OUTPUT: none, the transformation is done in-place

This is the same as *inverse_transform()* but lacks normalization so that `f.forward_transform().backward_transform() == n*f`. Where `n` is the size of the array.

EXAMPLES:

```
sage: a = FastFourierTransform(125)
sage: b = FastFourierTransform(125)
sage: for i in range(1, 60): a[i]=1
sage: for i in range(1, 60): b[i]=1
sage: a.forward_transform()
sage: a.backward_transform()
sage: (a.plot() + b.plot()).show(ymin=0)     # long time (2s on sage.math,␣
↪2011), needs sage.plot
sage: abs(sum([CDF(a[i])/125-CDF(b[i]) for i in range(125)])) < 2**-16
True
```

```
>>> from sage.all import *
>>> a = FastFourierTransform(Integer(125))
>>> b = FastFourierTransform(Integer(125))
>>> for i in range(Integer(1), Integer(60)): a[i]=Integer(1)
>>> for i in range(Integer(1), Integer(60)): b[i]=Integer(1)
>>> a.forward_transform()
>>> a.backward_transform()
>>> (a.plot() + b.plot()).show(ymin=Integer(0))     # long time (2s on sage.
```

(continues on next page)

(continued from previous page)

```
↪math, 2011), needs sage.plot
>>> abs(sum([CDF(a[i])/Integer(125)-CDF(b[i]) for i in range(Integer(125))]))
↪< Integer(2)**-Integer(16)
True
```

Here we check it with a power of two:

```
sage: a = FastFourierTransform(128)
sage: b = FastFourierTransform(128)
sage: for i in range(1, 60): a[i]=1
sage: for i in range(1, 60): b[i]=1
sage: a.forward_transform()
sage: a.backward_transform()
sage: (a.plot() + b.plot()).show(ymin=0)                                    #
↪needs sage.plot
```

```
>>> from sage.all import *
>>> a = FastFourierTransform(Integer(128))
>>> b = FastFourierTransform(Integer(128))
>>> for i in range(Integer(1), Integer(60)): a[i]=Integer(1)
>>> for i in range(Integer(1), Integer(60)): b[i]=Integer(1)
>>> a.forward_transform()
>>> a.backward_transform()
>>> (a.plot() + b.plot()).show(ymin=Integer(0))
↪     # needs sage.plot
```

forward_transform()

Compute the in-place forward Fourier transform of this data using the Cooley-Tukey algorithm.

OUTPUT: none, the transformation is done in-place

If the number of sample points in the input is a power of 2 then the gsl function `gsl_fft_complex_radix2_forward` is automatically called. Otherwise, `gsl_fft_complex_forward` is called.

EXAMPLES:

```
sage: a = FastFourierTransform(4)
sage: for i in range(4): a[i] = i
sage: a.forward_transform()
sage: a #abs tol 1e-2
[(6.0, 0.0), (-2.0, 2.0), (-2.0, 0.0), (-2.0, -2.0)]
```

```
>>> from sage.all import *
>>> a = FastFourierTransform(Integer(4))
>>> for i in range(Integer(4)): a[i] = i
>>> a.forward_transform()
>>> a #abs tol 1e-2
[(6.0, 0.0), (-2.0, 2.0), (-2.0, 0.0), (-2.0, -2.0)]
```

inverse_transform()

Compute the in-place inverse Fourier transform of this data using the Cooley-Tukey algorithm.

OUTPUT: none, the transformation is done in-place

If the number of sample points in the input is a power of 2 then the function `gsl_fft_complex_radix2_inverse` is automatically called. Otherwise, `gsl_fft_complex_inverse` is called.

This transform is normalized so `f.forward_transform().inverse_transform() == f` modulo round-off errors. See also `backward_transform()`.

EXAMPLES:

```
sage: a = FastFourierTransform(125)
sage: b = FastFourierTransform(125)
sage: for i in range(1, 60): a[i]=1
sage: for i in range(1, 60): b[i]=1
sage: a.forward_transform()
sage: a.inverse_transform()
sage: a.plot() + b.plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 250 graphics primitives
sage: abs(sum([CDF(a[i])-CDF(b[i]) for i in range(125)])) < 2**-16
True
```

```
>>> from sage.all import *
>>> a = FastFourierTransform(Integer(125))
>>> b = FastFourierTransform(Integer(125))
>>> for i in range(Integer(1), Integer(60)): a[i]=Integer(1)
>>> for i in range(Integer(1), Integer(60)): b[i]=Integer(1)
>>> a.forward_transform()
>>> a.inverse_transform()
>>> a.plot() + b.plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 250 graphics primitives
>>> abs(sum([CDF(a[i])-CDF(b[i]) for i in range(Integer(125))])) <␣
↪Integer(2)**-Integer(16)
True
```

Here we check it with a power of two:

```
sage: a = FastFourierTransform(128)
sage: b = FastFourierTransform(128)
sage: for i in range(1, 60): a[i]=1
sage: for i in range(1, 60): b[i]=1
sage: a.forward_transform()
sage: a.inverse_transform()
sage: a.plot() + b.plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 256 graphics primitives
```

```
>>> from sage.all import *
>>> a = FastFourierTransform(Integer(128))
>>> b = FastFourierTransform(Integer(128))
>>> for i in range(Integer(1), Integer(60)): a[i]=Integer(1)
>>> for i in range(Integer(1), Integer(60)): b[i]=Integer(1)
>>> a.forward_transform()
>>> a.inverse_transform()
>>> a.plot() + b.plot()                                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics object consisting of 256 graphics primitives
```

plot (*style='rect'*, *xmin=None*, *xmax=None*, ***args*)

Plot a slice of the array.

- `style` – style of the plot, options are `'rect'` or `'polar'`
  - `'rect'` – height represents real part, color represents imaginary part
  - `'polar'` – height represents absolute value, color represents argument
- `xmin` – the lower bound of the slice to plot; 0 by default
- `xmax` – the upper bound of the slice to plot; `len(self)` by default
- `**args` – passed on to the line plotting function

OUTPUT: a plot of the array

EXAMPLES:

```
sage: # needs sage.plot
sage: a = FastFourierTransform(16)
sage: for i in range(16): a[i] = (random(),random())
sage: A = plot(a)
sage: B = plot(a, style='polar')                                           #␣
↪needs sage.symbolic
sage: type(A)
<class 'sage.plot.graphics.Graphics'>
sage: type(B)                                                              #␣
↪needs sage.symbolic
<class 'sage.plot.graphics.Graphics'>
sage: a = FastFourierTransform(125)
sage: b = FastFourierTransform(125)
sage: for i in range(1, 60): a[i]=1
sage: for i in range(1, 60): b[i]=1
sage: a.forward_transform()
sage: a.inverse_transform()
sage: a.plot() + b.plot()
Graphics object consisting of 250 graphics primitives
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> a = FastFourierTransform(Integer(16))
>>> for i in range(Integer(16)): a[i] = (random(),random())
>>> A = plot(a)
>>> B = plot(a, style='polar')                                           #␣
↪needs sage.symbolic
>>> type(A)
<class 'sage.plot.graphics.Graphics'>
>>> type(B)                                                              #␣
↪needs sage.symbolic
<class 'sage.plot.graphics.Graphics'>
>>> a = FastFourierTransform(Integer(125))
>>> b = FastFourierTransform(Integer(125))
```

(continues on next page)

(continued from previous page)

```
>>> for i in range(Integer(1), Integer(60)): a[i]=Integer(1)
>>> for i in range(Integer(1), Integer(60)): b[i]=Integer(1)
>>> a.forward_transform()
>>> a.inverse_transform()
>>> a.plot() + b.plot()
Graphics object consisting of 250 graphics primitives
```

class sage.calculus.transforms.fft.**FourierTransform_complex**

Bases: object

class sage.calculus.transforms.fft.**FourierTransform_real**

Bases: object

2.24 Solving ODE numerically by GSL

AUTHORS:

- Joshua Kantor (2004-2006)
- Robert Marik (2010 - fixed docstrings)

class sage.calculus.ode.**PyFunctionWrapper**

Bases: object

class sage.calculus.ode.**ode_solver**(*function=None*, *jacobian=None*, *h=0.01*, *error_abs=1e-10*, *error_rel=1e-10*, *a=False*, *a_dydt=False*, *scale_abs=False*, *algorithm='rkf45'*, *y_0=None*, *t_span=None*, *params=[]*)

Bases: object

ode_solver() is a class that wraps the GSL library's ode solver routines.

To use it, instantiate the class:

```
sage: T = ode_solver()
```

```
>>> from sage.all import *
>>> T = ode_solver()
```

To solve a system of the form $dy_i/dt = f_i(t, y)$, you must supply a vector or tuple/list valued function f representing f_i. The functions f and the jacobian should have the form foo(t,y) or foo(t,y,params). params which is optional allows for your function to depend on one or a tuple of parameters. Note if you use it, params must be a tuple even if it only has one component. For example if you wanted to solve $y'' + y = 0$, you would need to write it as a first order system:

```
y_0' = y_1
y_1' = -y_0
```

In code:

```
sage: f = lambda t, y: [y[1], -y[0]]
sage: T.function = f
```

```
>>> from sage.all import *
>>> f = lambda t, y: [y[Integer(1)], -y[Integer(0)]]
>>> T.function = f
```

For some algorithms, the jacobian must be supplied as well, the form of this should be a function returning a list of lists of the form `[ [df_1/dy_1,...,df_1/dy_n], ..., [df_n/dy_1,...,df_n,dy_n], [df_1/dt, ...,df_n/dt] ]`.

There are examples below, if your jacobian was the function `my_jacobian` you would do:

```
sage: T.jacobian = my_jacobian     # not tested, since it doesn't make sense to␣
↪test this
```

```
>>> from sage.all import *
>>> T.jacobian = my_jacobian     # not tested, since it doesn't make sense to␣
↪test this
```

There are a variety of algorithms available for different types of systems. Possible algorithms are

- `'rkf45'` – Runge-Kutta-Fehlberg (4,5)
- `'rk2'` – embedded Runge-Kutta (2,3)
- `'rk4'` – 4-th order classical Runge-Kutta
- `'rk8pd'` – Runge-Kutta Prince-Dormand (8,9)
- `'rk2imp'` – implicit 2nd order Runge-Kutta at gaussian points
- `'rk4imp'` – implicit 4-th order Runge-Kutta at gaussian points
- `'bsimp'` – implicit Burlisch-Stoer (requires jacobian)
- `'gear1'` – M=1 implicit gear
- `'gear2'` – M=2 implicit gear

The default algorithm is `'rkf45'`. If you instead wanted to use `'bsimp'` you would do:

```
sage: T.algorithm = "bsimp"
```

```
>>> from sage.all import *
>>> T.algorithm = "bsimp"
```

The user should supply initial conditions in `y_0`. For example if your initial conditions are $y_0 = 1, y_1 = 1$, do:

```
sage: T.y_0 = [1,1]
```

```
>>> from sage.all import *
>>> T.y_0 = [Integer(1),Integer(1)]
```

The actual solver is invoked by the method *`ode_solve()`*. It has arguments `t_span`, `y_0`, `num_points`, `params`. `y_0` must be supplied either as an argument or above by assignment. Params which are optional and only necessary if your system uses `params` can be supplied to `ode_solve` or by assignment.

`t_span` is the time interval on which to solve the ode. There are two ways to specify `t_span`:

- If `num_points` is not specified, then the sequence `t_span` is used as the time points for the solution. Note that the first element `t_span[0]` is the initial time, where the initial condition `y_0` is the specified solution, and subsequent elements are the ones where the solution is computed.

- If `num_points` is specified and `t_span` is a sequence with just 2 elements, then these are the starting and ending times, and the solution will be computed at `num_points` equally spaced points between `t_span[0]` and `t_span[1]`. The initial condition is also included in the output so that `num_points + 1` total points are returned. E.g. if `t_span = [0.0, 1.0]` and `num_points = 10`, then solution is returned at the 11 time points `[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]`.

(Note that if `num_points` is specified and `t_span` is not length 2 then `t_span` are used as the time points and `num_points` is ignored.)

Error is estimated via the expression `D_i = error_abs*s_i+error_rel*(a|y_i|+a_dydt*h*|y_i'|)`. The user can specify

- `error_abs` (1e-10 by default),
- `error_rel` (1e-10 by default),
- `a` (1 by default),
- `a_dydt` (0 by default) and
- `s_i` (as `scaling_abs` which should be a tuple and is 1 in all components by default).

If you specify one of `a` or `a_dydt` you must specify the other. You may specify `a` and `a_dydt` without `scaling_abs` (which will be taken =1 be default). `h` is the initial step size, which is 1e-2 by default.

`ode_solve` solves the solution as a list of tuples of the form, `[ (t_0,[y_1,...,y_n]),(t_1,[y_1,...,y_n]),...,(t_n,[y_1,...,y_n])]`.

This data is stored in the variable solutions:

```
sage: T.solution                # not tested
```

```
>>> from sage.all import *
>>> T.solution                # not tested
```

EXAMPLES:

Consider solving the Van der Pol oscillator $x''(t)+ux'(t)(x(t)^2-1)+x(t)=0$ between $t=0$ and $t=100$. As a first order system it is $x'=y$, $y'=-x+uy(1-x^2)$. Let us take $u=10$ and use initial conditions $(x,y)=(1,0)$ and use the Runge-Kutta Prince-Dormand algorithm.

```
sage: def f_1(t, y, params):
....:    return [y[1], -y[0] - params[0]*y[1]*(y[0]**2-1.0)]

sage: def j_1(t, y, params):
....:    return [[0.0, 1.0],
....:            [-2.0*params[0]*y[0]*y[1] - 1.0, -params[0]*(y[0]*y[0]-1.0)],
....:            [0.0, 0.0]]

sage: T = ode_solver()
sage: T.algorithm = "rk8pd"
sage: T.function = f_1
sage: T.jacobian = j_1
sage: T.ode_solve(y_0=[1,0], t_span=[0,100], params=[10.0], num_points=1000)
sage: import tempfile
sage: with tempfile.NamedTemporaryFile(suffix='.png') as f:                    #␣
↪needs sage.plot
....:     T.plot_solution(filename=f.name)
```

```
>>> from sage.all import *
>>> def f_1(t, y, params):
...     return [y[Integer(1)], -y[Integer(0)] -␣
↪params[Integer(0)]*y[Integer(1)]*(y[Integer(0)]**Integer(2)-RealNumber('1.0'))]

>>> def j_1(t, y, params):
...     return [[RealNumber('0.0'), RealNumber('1.0')],
...             [-RealNumber('2.0')*params[Integer(0)]*y[Integer(0)]*y[Integer(1)]␣
↪- RealNumber('1.0'), -params[Integer(0)]*(y[Integer(0)]*y[Integer(0)]-
↪RealNumber('1.0'))],
...             [RealNumber('0.0'), RealNumber('0.0')]]

>>> T = ode_solver()
>>> T.algorithm = "rk8pd"
>>> T.function = f_1
>>> T.jacobian = j_1
>>> T.ode_solve(y_0=[Integer(1),Integer(0)], t_span=[Integer(0),Integer(100)],␣
↪params=[RealNumber('10.0')], num_points=Integer(1000))
>>> import tempfile
>>> with tempfile.NamedTemporaryFile(suffix='.png') as f:                  #␣
↪needs sage.plot
...     T.plot_solution(filename=f.name)
```

The solver line is equivalent to:

```
sage: T.ode_solve(y_0=[1,0], t_span=[x/10.0 for x in range(1000)], params=[10.0])
```

```
>>> from sage.all import *
>>> T.ode_solve(y_0=[Integer(1),Integer(0)], t_span=[x/RealNumber('10.0') for x␣
↪in range(Integer(1000))], params=[RealNumber('10.0')])
```

Let's try a system:

```
y_0'=y_1*y_2
y_1'=-y_0*y_2
y_2'=-.51*y_0*y_1
```

We will not use the jacobian this time and will change the error tolerances.

```
sage: g_1 = lambda t,y: [y[1]*y[2], -y[0]*y[2], -0.51*y[0]*y[1]]
sage: T.function = g_1
sage: T.y_0 = [0,1,1]
sage: T.scale_abs = [1e-4, 1e-4, 1e-5]
sage: T.error_rel = 1e-4
sage: T.ode_solve(t_span=[0,12], num_points=100)
```

```
>>> from sage.all import *
>>> g_1 = lambda t,y: [y[Integer(1)]*y[Integer(2)], -y[Integer(0)]*y[Integer(2)],␣
↪-RealNumber('0.51')*y[Integer(0)]*y[Integer(1)]]
>>> T.function = g_1
>>> T.y_0 = [Integer(0),Integer(1),Integer(1)]
>>> T.scale_abs = [RealNumber('1e-4'), RealNumber('1e-4'), RealNumber('1e-5')]
```

(continues on next page)

(continued from previous page)

```
>>> T.error_rel = RealNumber('1e-4')
>>> T.ode_solve(t_span=[Integer(0),Integer(12)], num_points=Integer(100))
```

By default `T.plot_solution()` plots the y_0; to plot general y_i, use:

```
sage: with tempfile.NamedTemporaryFile(suffix='.png') as f:                    #␣
↪needs sage.plot
....:     T.plot_solution(i=0, filename=f.name)
....:     T.plot_solution(i=1, filename=f.name)
....:     T.plot_solution(i=2, filename=f.name)
```

```
>>> from sage.all import *
>>> with tempfile.NamedTemporaryFile(suffix='.png') as f:                     #␣
↪needs sage.plot
...     T.plot_solution(i=Integer(0), filename=f.name)
...     T.plot_solution(i=Integer(1), filename=f.name)
...     T.plot_solution(i=Integer(2), filename=f.name)
```

The method interpolate_solution will return a spline interpolation through the points found by the solver. By default, y_0 is interpolated. You can interpolate y_i through the keyword argument `i`.

```
sage: f = T.interpolate_solution()
sage: plot(f,0,12).show()                                                      #␣
↪needs sage.plot
sage: f = T.interpolate_solution(i=1)
sage: plot(f,0,12).show()                                                      #␣
↪needs sage.plot
sage: f = T.interpolate_solution(i=2)
sage: plot(f,0,12).show()                                                      #␣
↪needs sage.plot
sage: f = T.interpolate_solution()
sage: from math import pi
sage: f(pi)
0.5379...
```

```
>>> from sage.all import *
>>> f = T.interpolate_solution()
>>> plot(f,Integer(0),Integer(12)).show()                                      ␣
↪               # needs sage.plot
>>> f = T.interpolate_solution(i=Integer(1))
>>> plot(f,Integer(0),Integer(12)).show()                                      ␣
↪               # needs sage.plot
>>> f = T.interpolate_solution(i=Integer(2))
>>> plot(f,Integer(0),Integer(12)).show()                                      ␣
↪               # needs sage.plot
>>> f = T.interpolate_solution()
>>> from math import pi
>>> f(pi)
0.5379...
```

The solver attributes may also be set up using arguments to ode_solver. The previous example can be rewritten as:

```
sage: T = ode_solver(g_1, y_0=[0,1,1], scale_abs=[1e-4,1e-4,1e-5],
....:                error_rel=1e-4, algorithm='rk8pd')
sage: T.ode_solve(t_span=[0,12], num_points=100)
sage: f = T.interpolate_solution()
sage: f(pi)
0.5379...
```

```
>>> from sage.all import *
>>> T = ode_solver(g_1, y_0=[Integer(0),Integer(1),Integer(1)], scale_
↪abs=[RealNumber('1e-4'),RealNumber('1e-4'),RealNumber('1e-5')],
...                error_rel=RealNumber('1e-4'), algorithm='rk8pd')
>>> T.ode_solve(t_span=[Integer(0),Integer(12)], num_points=Integer(100))
>>> f = T.interpolate_solution()
>>> f(pi)
0.5379...
```

Unfortunately because Python functions are used, this solver is slow on systems that require many function evaluations. It is possible to pass a compiled function by deriving from the class *ode_system* and overloading `c_f` and `c_j` with C functions that specify the system. The following will work in the notebook:

```
%cython
cimport sage.calculus.ode
import sage.calculus.ode
from sage.libs.gsl.all cimport *

cdef class van_der_pol(sage.calculus.ode.ode_system):
    cdef int c_f(self, double t, double *y, double *dydt):
        dydt[0]=y[1]
        dydt[1]=-y[0]-1000*y[1]*(y[0]*y[0]-1)
        return GSL_SUCCESS
    cdef int c_j(self, double t, double *y, double *dfdy, double *dfdt):
        dfdy[0]=0
        dfdy[1]=1.0
        dfdy[2]=-2.0*1000*y[0]*y[1]-1.0
        dfdy[3]=-1000*(y[0]*y[0]-1.0)
        dfdt[0]=0
        dfdt[1]=0
        return GSL_SUCCESS
```

After executing the above block of code you can do the following (WARNING: the following is *not* automatically doctested):

```
sage: # not tested
sage: T = ode_solver()
sage: T.algorithm = "bsimp"
sage: vander = van_der_pol()
sage: T.function = vander
sage: T.ode_solve(y_0=[1, 0], t_span=[0, 2000],
....:             num_points=1000)
sage: from tempfile import NamedTemporaryFile
sage: with NamedTemporaryFile(suffix='.png') as f:
....:     T.plot_solution(i=0, filename=f.name)
```

```
>>> from sage.all import *
>>> # not tested
>>> T = ode_solver()
>>> T.algorithm = "bsimp"
>>> vander = van_der_pol()
>>> T.function = vander
>>> T.ode_solve(y_0=[Integer(1), Integer(0)], t_span=[Integer(0), Integer(2000)],
...             num_points=Integer(1000))
>>> from tempfile import NamedTemporaryFile
>>> with NamedTemporaryFile(suffix='.png') as f:
...     T.plot_solution(i=Integer(0), filename=f.name)
```

interpolate_solution(*i=0*)

ode_solve(*t_span=False*, *y_0=False*, *num_points=False*, *params=[]*)

plot_solution(*i=0*, *filename=None*, *interpolate=False*, ***kwds*)

Plot a one dimensional projection of the solution.

INPUT:

- `i` – nonnegative integer; composant of the projection
- `filename` – string or `None`; whether to plot the picture or save it in a file
- `interpolate` – whether to interpolate between the points of the discretized solution
- additional keywords are passed to the graphics primitive

EXAMPLES:

```
sage: T = ode_solver()
sage: T.function = lambda t,y: [cos(y[0]) * sin(t)]
sage: T.jacobian = lambda t,y: [[-sin(y[0]) * sin(t)]]
sage: T.ode_solve(y_0=[1],t_span=[0,20],num_points=1000)
sage: T.plot_solution()                                                  #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> T = ode_solver()
>>> T.function = lambda t,y: [cos(y[Integer(0)]) * sin(t)]
>>> T.jacobian = lambda t,y: [[-sin(y[Integer(0)]) * sin(t)]]
>>> T.ode_solve(y_0=[Integer(1)],t_span=[Integer(0),Integer(20)],num_
↪points=Integer(1000))
>>> T.plot_solution()                                                   #␣
↪needs sage.plot
```

And with some options:

```
sage: T.plot_solution(color='red', axes_labels=["t", "x(t)"])            #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> T.plot_solution(color='red', axes_labels=["t", "x(t)"])             #␣
↪needs sage.plot
```

class sage.calculus.ode.**ode_system**

Bases: object

2.25 Numerical Integration

AUTHORS:

- Josh Kantor (2007-02): first version
- William Stein (2007-02): rewrite of docs, conventions, etc.
- Robert Bradshaw (2008-08): fast float integration
- Jeroen Demeyer (2011-11-23): Issue #12047: return 0 when the integration interval is a point; reformat documentation and add to the reference manual.

class sage.calculus.integration.**PyFunctionWrapper**

Bases: object

class sage.calculus.integration.**compiled_integrand**

Bases: object

sage.calculus.integration.**monte_carlo_integral**(*func*, *xl*, *xu*, *calls*, *algorithm='plain'*, *params=None*)

Integrate func by Monte-Carlo method.

Integrate func over the dim-dimensional hypercubic region defined by the lower and upper limits in the arrays xl and xu, each of size dim.

The integration uses a fixed number of function calls and obtains random sampling points using the default gsl's random number generator.

ALGORITHM: Uses calls to the GSL (GNU Scientific Library) C library. Documentation can be found in [GSL] chapter "Monte Carlo Integration".

INPUT:

- func – the function to integrate
- params – used to pass parameters to your function
- xl – list of lower limits
- xu – list of upper limits
- calls – number of functions calls used
- algorithm – valid choices are:
  - 'plain' – The plain Monte Carlo algorithm samples points randomly from the integration region to estimate the integral and its error.
  - 'miser' – The MISER algorithm of Press and Farrar is based on recursive stratified sampling
  - 'vegas' – The VEGAS algorithm of Lepage is based on importance sampling.

OUTPUT:

A tuple whose first component is the approximated integral and whose second component is an error estimate.

EXAMPLES:

```
sage: x, y = SR.var('x,y')
sage: monte_carlo_integral(x*y, [0,0], [2,2], 10000)    # abs tol 0.1
(4.0, 0.0)
sage: integral(integral(x*y, (x,0,2)), (y,0,2))
4
```

```
>>> from sage.all import *
>>> x, y = SR.var('x,y')
>>> monte_carlo_integral(x*y, [Integer(0),Integer(0)], [Integer(2),Integer(2)],␣
↪Integer(10000))    # abs tol 0.1
(4.0, 0.0)
>>> integral(integral(x*y, (x,Integer(0),Integer(2))), (y,Integer(0),Integer(2)))
4
```

An example with a parameter:

```
sage: x, y, z = SR.var('x,y,z')
sage: monte_carlo_integral(x*y*z, [0,0], [2,2], 10000, params=[1.2])    # abs tol␣
↪0.1
(4.8, 0.0)
```

```
>>> from sage.all import *
>>> x, y, z = SR.var('x,y,z')
>>> monte_carlo_integral(x*y*z, [Integer(0),Integer(0)], [Integer(2),Integer(2)],␣
↪Integer(10000), params=[RealNumber('1.2')])    # abs tol 0.1
(4.8, 0.0)
```

Integral of a constant:

```
sage: monte_carlo_integral(3, [0,0], [2,2], 10000)    # abs tol 0.1
(12, 0.0)
```

```
>>> from sage.all import *
>>> monte_carlo_integral(Integer(3), [Integer(0),Integer(0)], [Integer(2),
↪Integer(2)], Integer(10000))    # abs tol 0.1
(12, 0.0)
```

Test different algorithms:

```
sage: x, y, z = SR.var('x,y,z')
sage: f(x,y,z) = exp(z) * cos(x + sin(y))
sage: for algo in ['plain', 'miser', 'vegas']:  # abs tol 0.01
....:   monte_carlo_integral(f, [0,0,-1], [2,2,1], 10^6, algorithm=algo)
(-1.06, 0.01)
(-1.06, 0.01)
(-1.06, 0.01)
```

```
>>> from sage.all import *
>>> x, y, z = SR.var('x,y,z')
>>> __tmp__=var("x,y,z"); f = symbolic_expression(exp(z) * cos(x + sin(y))).
↪function(x,y,z)
>>> for algo in ['plain', 'miser', 'vegas']:  # abs tol 0.01
```

(continues on next page)

(continued from previous page)

```
...   monte_carlo_integral(f, [Integer(0),Integer(0),-Integer(1)], [Integer(2),
↪Integer(2),Integer(1)], Integer(10)**Integer(6), algorithm=algo)
(-1.06, 0.01)
(-1.06, 0.01)
(-1.06, 0.01)
```

Tests with Python functions:

```
sage: def f(u, v): return u * v
sage: monte_carlo_integral(f, [0,0], [2,2], 10000)  # abs tol 0.1
(4.0, 0.0)
sage: monte_carlo_integral(lambda u,v: u*v, [0,0], [2,2], 10000)  # abs tol 0.1
(4.0, 0.0)
sage: def f(x1, x2, x3, x4): return x1*x2*x3*x4
sage: monte_carlo_integral(f, [0,0], [2,2], 1000, params=[0.6,2])  # abs tol 0.2
(4.8, 0.0)
```

```
>>> from sage.all import *
>>> def f(u, v): return u * v
>>> monte_carlo_integral(f, [Integer(0),Integer(0)], [Integer(2),Integer(2)],␣
↪Integer(10000))  # abs tol 0.1
(4.0, 0.0)
>>> monte_carlo_integral(lambda u,v: u*v, [Integer(0),Integer(0)], [Integer(2),
↪Integer(2)], Integer(10000))  # abs tol 0.1
(4.0, 0.0)
>>> def f(x1, x2, x3, x4): return x1*x2*x3*x4
>>> monte_carlo_integral(f, [Integer(0),Integer(0)], [Integer(2),Integer(2)],␣
↪Integer(1000), params=[RealNumber('0.6'),Integer(2)])  # abs tol 0.2
(4.8, 0.0)
```

AUTHORS:

- Vincent Delecroix
- Vincent Klein

`sage.calculus.integration.`**`numerical_integral`**(*func*, *a*, *b=None*, *algorithm='qag'*, *max_points=87*, *params=[]*, *eps_abs=1e-06*, *eps_rel=1e-06*, *rule=6*)

Return the numerical integral of the function on the interval from a to b and an error bound.

INPUT:

- `a`, `b` – the interval of integration, specified as two numbers or as a tuple/list with the first element the lower bound and the second element the upper bound. Use `+Infinity` and `-Infinity` for plus or minus infinity.
- `algorithm` – valid choices are:
  - 'qag' – for an adaptive integration
  - 'qags' – for an adaptive integration with (integrable) singularities
  - 'qng' – for a non-adaptive Gauss-Kronrod (samples at a maximum of 87pts)
- `max_points` – sets the maximum number of sample points
- `params` – used to pass parameters to your function

- `eps_abs`, `eps_rel` – sets the absolute and relative error tolerances which satisfies the relation `|RESULT - I| <= max(eps_abs, eps_rel * |I|)`, where `I = \int_a^b f(x) d x`.
- `rule` – this controls the Gauss-Kronrod rule used in the adaptive integration:
  - rule=1 – 15 point rule
  - rule=2 – 21 point rule
  - rule=3 – 31 point rule
  - rule=4 – 41 point rule
  - rule=5 – 51 point rule
  - rule=6 – 61 point rule

  Higher key values are more accurate for smooth functions but lower key values deal better with discontinuities.

OUTPUT:

A tuple whose first component is the answer and whose second component is an error estimate.

REMARK:

There is also a method `nintegral` on symbolic expressions that implements numerical integration using Maxima. It is potentially very useful for symbolic expressions.

EXAMPLES:

To integrate the function x^2 from 0 to 1, we do

```
sage: numerical_integral(x^2, 0, 1, max_points=100)
(0.3333333333333333, 3.700743415417188e-15)
```

```
>>> from sage.all import *
>>> numerical_integral(x**Integer(2), Integer(0), Integer(1), max_
↪points=Integer(100))
(0.3333333333333333, 3.700743415417188e-15)
```

To integrate the function $\sin(x)^3 + \sin(x)$ we do

```
sage: numerical_integral(sin(x)^3 + sin(x),  0, pi)
(3.333333333333333, 3.700743415417188e-14)
```

```
>>> from sage.all import *
>>> numerical_integral(sin(x)**Integer(3) + sin(x),  Integer(0), pi)
(3.333333333333333, 3.700743415417188e-14)
```

The input can be any callable:

```
sage: numerical_integral(lambda x: sin(x)^3 + sin(x),  0, pi)
(3.333333333333333, 3.700743415417188e-14)
```

```
>>> from sage.all import *
>>> numerical_integral(lambda x: sin(x)**Integer(3) + sin(x),  Integer(0), pi)
(3.333333333333333, 3.700743415417188e-14)
```

We check this with a symbolic integration:

```
sage: (sin(x)^3+sin(x)).integral(x,0,pi)
10/3
```

```
>>> from sage.all import *
>>> (sin(x)**Integer(3)+sin(x)).integral(x,Integer(0),pi)
10/3
```

If we want to change the error tolerances and Gauss rule used:

```
sage: f = x^2
sage: numerical_integral(f, 0, 1, max_points=200, eps_abs=1e-7, eps_rel=1e-7,␣
↪rule=4)
(0.3333333333333333, 3.700743415417188e-15)
```

```
>>> from sage.all import *
>>> f = x**Integer(2)
>>> numerical_integral(f, Integer(0), Integer(1), max_points=Integer(200), eps_
↪abs=RealNumber('1e-7'), eps_rel=RealNumber('1e-7'), rule=Integer(4))
(0.3333333333333333, 3.700743415417188e-15)
```

For a Python function with parameters:

```
sage: f(x,a) = 1/(a+x^2)
sage: [numerical_integral(f, 1, 2, max_points=100, params=[n])[0]  # abs tol 1.0e-
↪6
....:            for n in range(10)]
[0.5000000000000000,
 0.3217505543966422,
 0.24030098317248832,
 0.19253082576711372,
 0.1608752771983211,
 0.138275456763492,
 0.1212997593570257,
 0.10806674191683492,
 0.09745444625553161,
 0.08875068305030848]

sage: y = var('y')
sage: numerical_integral(x*y, 0, 1)
Traceback (most recent call last):
...
ValueError: The function to be integrated depends on 2 variables (x, y),
and so cannot be integrated in one dimension. Please fix additional
variables with the 'params' argument
```

```
>>> from sage.all import *
>>> __tmp__=var("x,a"); f = symbolic_expression(Integer(1)/(a+x**Integer(2))).
↪function(x,a)
>>> [numerical_integral(f, Integer(1), Integer(2), max_points=Integer(100),␣
↪params=[n])[Integer(0)]  # abs tol 1.0e-6
...            for n in range(Integer(10))]
[0.5000000000000000,
```

(continues on next page)

(continued from previous page)

```
 0.3217505543966422,
 0.24030098317248832,
 0.19253082576711372,
 0.1608752771983211,
 0.138275456763492,
 0.1212997593570257,
 0.10806674191683492,
 0.09745444625553161,
 0.08875068305030848]

>>> y = var('y')
>>> numerical_integral(x*y, Integer(0), Integer(1))
Traceback (most recent call last):
...
ValueError: The function to be integrated depends on 2 variables (x, y),
and so cannot be integrated in one dimension. Please fix additional
variables with the 'params' argument
```

Note the parameters are always a tuple even if they have one component.

It is possible to integrate on infinite intervals as well by using +Infinity or -Infinity in the interval argument. For example:

```
sage: f = exp(-x)
sage: numerical_integral(f, 0, +Infinity)  # random output
(0.99999999999957279, 1.8429811298996553e-07)
```

```
>>> from sage.all import *
>>> f = exp(-x)
>>> numerical_integral(f, Integer(0), +Infinity)  # random output
(0.99999999999957279, 1.8429811298996553e-07)
```

Note the coercion to the real field RR, which prevents underflow:

```
sage: f = exp(-x**2)
sage: numerical_integral(f, -Infinity, +Infinity)  # random output
(1.7724538509060035, 3.4295192165889879e-08)
```

```
>>> from sage.all import *
>>> f = exp(-x**Integer(2))
>>> numerical_integral(f, -Infinity, +Infinity)  # random output
(1.7724538509060035, 3.4295192165889879e-08)
```

One can integrate any real-valued callable function:

```
sage: numerical_integral(lambda x: abs(zeta(x)), [1.1,1.5])  # random output
(1.8488570602160455, 2.052643677492633e-14)
```

```
>>> from sage.all import *
>>> numerical_integral(lambda x: abs(zeta(x)), [RealNumber('1.1'),RealNumber('1.5
↪')])  # random output
(1.8488570602160455, 2.052643677492633e-14)
```

We can also numerically integrate symbolic expressions using either this function (which uses GSL) or the native integration (which uses Maxima):

```
sage: exp(-1/x).nintegral(x, 1, 2)  # via maxima
(0.50479221787318..., 5.60431942934407...e-15, 21, 0)
sage: numerical_integral(exp(-1/x), 1, 2)
(0.50479221787318..., 5.60431942934407...e-15)
```

```
>>> from sage.all import *
>>> exp(-Integer(1)/x).nintegral(x, Integer(1), Integer(2))  # via maxima
(0.50479221787318..., 5.60431942934407...e-15, 21, 0)
>>> numerical_integral(exp(-Integer(1)/x), Integer(1), Integer(2))
(0.50479221787318..., 5.60431942934407...e-15)
```

We can also integrate constant expressions:

```
sage: numerical_integral(2, 1, 7)
(12.0, 0.0)
```

```
>>> from sage.all import *
>>> numerical_integral(Integer(2), Integer(1), Integer(7))
(12.0, 0.0)
```

If the interval of integration is a point, then the result is always zero (this makes sense within the Lebesgue theory of integration), see Issue #12047:

```
sage: numerical_integral(log, 0, 0)
(0.0, 0.0)
sage: numerical_integral(lambda x: sqrt(x), (-2.0, -2.0) )
(0.0, 0.0)
```

```
>>> from sage.all import *
>>> numerical_integral(log, Integer(0), Integer(0))
(0.0, 0.0)
>>> numerical_integral(lambda x: sqrt(x), (-RealNumber('2.0'), -RealNumber('2.0
↪')) )
(0.0, 0.0)
```

In the presence of integrable singularity, the default adaptive method might fail and it is advised to use `'qags'`:

```
sage: b = 1.81759643554688
sage: F(x) = sqrt((-x + b)/((x - 1.0)*x))
sage: numerical_integral(F, 1, b)
(inf, nan)
sage: numerical_integral(F, 1, b, algorithm='qags')    # abs tol 1e-10
(1.1817104238446596, 3.387268288079781e-07)
```

```
>>> from sage.all import *
>>> b = RealNumber('1.81759643554688')
>>> __tmp__=var("x"); F = symbolic_expression(sqrt((-x + b)/((x - RealNumber('1.0
↪'))*x))).function(x)
>>> numerical_integral(F, Integer(1), b)
(inf, nan)
```

(continues on next page)

(continued from previous page)

```
>>> numerical_integral(F, Integer(1), b, algorithm='qags')    # abs tol 1e-10
(1.1817104238446596, 3.387268288079781e-07)
```

AUTHORS:

- Josh Kantor
- William Stein
- Robert Bradshaw
- Jeroen Demeyer

ALGORITHM: Uses calls to the GSL (GNU Scientific Library) C library. Documentation can be found in [GSL] chapter "Numerical Integration".

2.26 Riemann Mapping

AUTHORS:

- Ethan Van Andel (2009-2011): initial version and upgrades
- Robert Bradshaw (2009): his "complex_plot" was adapted for plot_colored

Development supported by NSF award No. 0702939.

class sage.calculus.riemann.**Riemann_Map**

Bases: object

The Riemann_Map class computes an interior or exterior Riemann map, or an Ahlfors map of a region given by the supplied boundary curve(s) and center point. The class also provides various methods to evaluate, visualize, or extract data from the map.

A Riemann map conformally maps a simply connected region in the complex plane to the unit disc. The Ahlfors map does the same thing for multiply connected regions.

Note that all the methods are numerical. As a result all answers have some imprecision. Moreover, maps computed with small number of collocation points, or for unusually shaped regions, may be very inaccurate. Error computations for the ellipse can be found in the documentation for *analytic_boundary()* and *analytic_interior()*.

[BSV2010] provides an overview of the Riemann map and discusses the research that lead to the creation of this module.

INPUT:

- fs – list of the boundaries of the region, given as complex-valued functions with domain 0 to $2 * pi$. Note that the outer boundary must be parameterized counter clockwise (i.e. e^(I*t)) while the inner boundaries must be clockwise (i.e. e^(-I*t)).
- fprimes – list of the derivatives of the boundary functions (Must be in the same order as fs)
- a – complex, the center of the Riemann map. Will be mapped to the origin of the unit disc. Note that a MUST be within the region in order for the results to be mathematically valid.

The following inputs may be passed in as named parameters:

- N – integer (default: 500); the number of collocation points used to compute the map. More points will give more accurate results, especially near the boundaries, but will take longer to compute.
- exterior – boolean (default: False); if set to True, the exterior map will be computed, mapping the exterior of the region to the exterior of the unit circle.

The following inputs may be passed as named parameters in unusual circumstances:

- `ncorners` – integer (default: 4); if mapping a figure with (equally t-spaced) corners – corners that make a significant change in the direction of the boundary – better results may be sometimes obtained by accurately giving this parameter. Used to add the proper constant to the theta correspondence function.
- `opp` – boolean (default: `False`); set to `True` in very rare cases where the theta correspondence function is off by `pi`, that is, if red is mapped left of the origin in the color plot.

EXAMPLES:

The unit circle identity map:

```
sage: f(t) = e^(I*t)
sage: fprime(t) = I*e^(I*t)
sage: m = Riemann_Map([f], [fprime], 0)  # long time (4 sec)
sage: m.plot_colored() + m.plot_spiderweb()  # long time
Graphics object consisting of 22 graphics primitives
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))  # long time (4 sec)
>>> m.plot_colored() + m.plot_spiderweb()  # long time
Graphics object consisting of 22 graphics primitives
```

The exterior map for the unit circle:

```
sage: m = Riemann_Map([f], [fprime], 0, exterior=True)  # long time (4 sec)
sage: #spiderwebs are not supported for exterior maps
sage: m.plot_colored() # long time
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> m = Riemann_Map([f], [fprime], Integer(0), exterior=True)  # long time (4 sec)
>>> #spiderwebs are not supported for exterior maps
>>> m.plot_colored() # long time
Graphics object consisting of 1 graphics primitive
```

The unit circle with a small hole:

```
sage: f(t) = e^(I*t)
sage: fprime(t) = I*e^(I*t)
sage: hf(t) = 0.5*e^(-I*t)
sage: hfprime(t) = 0.5*-I*e^(-I*t)
sage: m = Riemann_Map([f, hf], [fprime, hfprime], 0.5 + 0.5*I)
sage: #spiderweb and color plots cannot be added for multiply
sage: #connected regions. Instead we do this.
sage: m.plot_spiderweb(withcolor = True)  # long time
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t)).function(t)
>>> __tmp__=var("t"); hf = symbolic_expression(RealNumber('0.5')*e**(-I*t)).
```

(continues on next page)

(continued from previous page)

```
↪function(t)
>>> __tmp__=var("t"); hfprime = symbolic_expression(RealNumber('0.5')*-I*e**(-
↪I*t)).function(t)
>>> m = Riemann_Map([f, hf], [fprime, hfprime], RealNumber('0.5') + RealNumber('0.
↪5')*I)
>>> #spiderweb and color plots cannot be added for multiply
>>> #connected regions. Instead we do this.
>>> m.plot_spiderweb(withcolor = True)  # long time
Graphics object consisting of 3 graphics primitives
```

A square:

```
sage: ps = polygon_spline([(-1, -1), (1, -1), (1, 1), (-1, 1)])
sage: f = lambda t: ps.value(real(t))
sage: fprime = lambda t: ps.derivative(real(t))
sage: m = Riemann_Map([f], [fprime], 0.25, ncorners=4)
sage: m.plot_colored() + m.plot_spiderweb()  # long time
Graphics object consisting of 22 graphics primitives
```

```
>>> from sage.all import *
>>> ps = polygon_spline([(-Integer(1), -Integer(1)), (Integer(1), -Integer(1)),␣
↪(Integer(1), Integer(1)), (-Integer(1), Integer(1))])
>>> f = lambda t: ps.value(real(t))
>>> fprime = lambda t: ps.derivative(real(t))
>>> m = Riemann_Map([f], [fprime], RealNumber('0.25'), ncorners=Integer(4))
>>> m.plot_colored() + m.plot_spiderweb()  # long time
Graphics object consisting of 22 graphics primitives
```

Compute rough error for this map:

```
sage: x = 0.75  # long time
sage: print("error = {}".format(m.inverse_riemann_map(m.riemann_map(x)) - x))  #␣
↪long time
error = (-0.000...+0.0016...j)
```

```
>>> from sage.all import *
>>> x = RealNumber('0.75')  # long time
>>> print("error = {}".format(m.inverse_riemann_map(m.riemann_map(x)) - x))  #␣
↪long time
error = (-0.000...+0.0016...j)
```

A fun, complex region for demonstration purposes:

```
sage: f(t) = e^(I*t)
sage: fp(t) = I*e^(I*t)
sage: ef1(t) = .2*e^(-I*t) +.4+.4*I
sage: ef1p(t) = -I*.2*e^(-I*t)
sage: ef2(t) = .2*e^(-I*t) -.4+.4*I
sage: ef2p(t) = -I*.2*e^(-I*t)
sage: pts = [(-.5,-.15-20/1000),(-.6,-.27-10/1000),(-.45,-.45),(0,-.65+10/1000),(.
↪45,-.45),(.6,-.27-10/1000),(.5,-.15-10/1000),(0,-.43+10/1000)]
sage: pts.reverse()
```

(continues on next page)

(continued from previous page)

```
sage: cs = complex_cubic_spline(pts)
sage: mf = lambda x:cs.value(x)
sage: mfprime = lambda x: cs.derivative(x)
sage: m = Riemann_Map([f,ef1,ef2,mf],[fp,ef1p,ef2p,mfprime],0,N = 400) # long time
sage: p = m.plot_colored(plot_points = 400) # long time
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t)).function(t)
>>> __tmp__=var("t"); fp = symbolic_expression(I*e**(I*t)).function(t)
>>> __tmp__=var("t"); ef1 = symbolic_expression(RealNumber('.2')*e**(-I*t)␣
↪+RealNumber('.4')+RealNumber('.4')*I).function(t)
>>> __tmp__=var("t"); ef1p = symbolic_expression(-I*RealNumber('.2')*e**(-I*t)).
↪function(t)
>>> __tmp__=var("t"); ef2 = symbolic_expression(RealNumber('.2')*e**(-I*t) -
↪RealNumber('.4')+RealNumber('.4')*I).function(t)
>>> __tmp__=var("t"); ef2p = symbolic_expression(-I*RealNumber('.2')*e**(-I*t)).
↪function(t)
>>> pts = [(-RealNumber('.5'),-RealNumber('.15')-Integer(20)/Integer(1000)),(-
↪RealNumber('.6'),-RealNumber('.27')-Integer(10)/Integer(1000)),(-RealNumber('.45
↪'),-RealNumber('.45')),(Integer(0),-RealNumber('.65')+Integer(10)/
↪Integer(1000)),(RealNumber('.45'),-RealNumber('.45')),(RealNumber('.6'),-
↪RealNumber('.27')-Integer(10)/Integer(1000)),(RealNumber('.5'),-RealNumber('.15
↪')-Integer(10)/Integer(1000)),(Integer(0),-RealNumber('.43')+Integer(10)/
↪Integer(1000))]
>>> pts.reverse()
>>> cs = complex_cubic_spline(pts)
>>> mf = lambda x:cs.value(x)
>>> mfprime = lambda x: cs.derivative(x)
>>> m = Riemann_Map([f,ef1,ef2,mf],[fp,ef1p,ef2p,mfprime],Integer(0),N =␣
↪Integer(400)) # long time
>>> p = m.plot_colored(plot_points = Integer(400)) # long time
```

ALGORITHM:

This class computes the Riemann Map via the Szego kernel using an adaptation of the method described by [KT1986].

compute_on_grid(*plot_range*, *x_points*)

Compute the Riemann map on a grid of points.

Note that these points are complex of the form z = x + y*i.

INPUT:

- `plot_range` – tuple of the form `[xmin, xmax, ymin, ymax]`; if the value is `[]`, the default plotting window of the map will be used
- `x_points` – integer; the size of the grid in the x direction; the number of points in the y direction is scaled accordingly

OUTPUT:

- a tuple containing `[z_values, xmin, xmax, ymin, ymax]` where `z_values` is the evaluation of the map on the specified grid.

EXAMPLES:

General usage:

```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: data = m.compute_on_grid([],5)
sage: data[0][8,1]
(-0.0879...+0.9709...j)
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t) - RealNumber('0.5
↪')*e**(-I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t) + RealNumber('0.
↪5')*I*e**(-I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))
>>> data = m.compute_on_grid([],Integer(5))
>>> data[Integer(0)][Integer(8),Integer(1)]
(-0.0879...+0.9709...j)
```

get_szego(*boundary=-1*, *absolute_value=False*)

Return a discretized version of the Szego kernel for each boundary function.

INPUT:

The following inputs may be passed in as named parameters:

- `boundary` – integer (default: -1); if < 0, *get_theta_points()* will return the points for all boundaries. If >= 0, *get_theta_points()* will return only the points for the boundary specified.
- `absolute_value` – boolean (default: `False`); if `True`, will return the absolute value of the (complex valued) Szego kernel instead of the kernel itself. Useful for plotting.

OUTPUT:

A list of points of the form `[t value, value of the Szego kernel at that t]`.

EXAMPLES:

Generic use:

```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: sz = m.get_szego(boundary=0)
sage: points = m.get_szego(absolute_value=True)
sage: list_plot(points)                                                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t) - RealNumber('0.5
↪')*e**(-I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t) + RealNumber('0.
↪5')*I*e**(-I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))
>>> sz = m.get_szego(boundary=Integer(0))
>>> points = m.get_szego(absolute_value=True)
```

(continues on next page)

(continued from previous page)

```
>>> list_plot(points)                                                        #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

Extending the points by a spline:

```
sage: s = spline(points)
sage: s(3*pi / 4)
0.0012158...
sage: plot(s,0,2*pi)  # plot the kernel                                      #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> s = spline(points)
>>> s(Integer(3)*pi / Integer(4))
0.0012158...
>>> plot(s,Integer(0),Integer(2)*pi)  # plot the kernel                        ␣
↪              # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

The unit circle with a small hole:

```
sage: f(t) = e^(I*t)
sage: fprime(t) = I*e^(I*t)
sage: hf(t) = 0.5*e^(-I*t)
sage: hfprime(t) = 0.5*-I*e^(-I*t)
sage: m = Riemann_Map([f, hf], [fprime, hfprime], 0.5 + 0.5*I)
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t)).function(t)
>>> __tmp__=var("t"); hf = symbolic_expression(RealNumber('0.5')*e**(-I*t)).
↪function(t)
>>> __tmp__=var("t"); hfprime = symbolic_expression(RealNumber('0.5')*-I*e**(-
↪I*t)).function(t)
>>> m = Riemann_Map([f, hf], [fprime, hfprime], RealNumber('0.5') +␣
↪RealNumber('0.5')*I)
```

Getting the szego for a specific boundary:

```
sage: sz0 = m.get_szego(boundary=0)
sage: sz1 = m.get_szego(boundary=1)
```

```
>>> from sage.all import *
>>> sz0 = m.get_szego(boundary=Integer(0))
>>> sz1 = m.get_szego(boundary=Integer(1))
```

get_theta_points(*boundary=-1*)

Return an array of points of the form `[t value, theta in e^(I*theta)]`, that is, a discretized version of the theta/boundary correspondence function. In other words, a point in this array [t1, t2] represents that the boundary point given by f(t1) is mapped to a point on the boundary of the unit circle given by e^(I*t2).

For multiply connected domains, `get_theta_points` will list the points for each boundary in the order that they were supplied.

INPUT:

The following input must all be passed in as named parameters:

- `boundary` – integer (default: -1); if < 0, `get_theta_points()` will return the points for all boundaries. If >= 0, `get_theta_points()` will return only the points for the boundary specified.

OUTPUT:

A list of points of the form `[t value, theta in e^(I*theta)]`.

EXAMPLES:

Getting the list of points, extending it via a spline, getting the points for only the outside of a multiply connected domain:

```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: points = m.get_theta_points()
sage: list_plot(points)                                                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t) - RealNumber('0.5
↪')*e**(-I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t) + RealNumber('0.
↪5')*I*e**(-I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))
>>> points = m.get_theta_points()
>>> list_plot(points)                                                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

Extending the points by a spline:

```
sage: s = spline(points)
sage: s(3*pi / 4)
1.627660...
```

```
>>> from sage.all import *
>>> s = spline(points)
>>> s(Integer(3)*pi / Integer(4))
1.627660...
```

The unit circle with a small hole:

```
sage: f(t) = e^(I*t)
sage: fprime(t) = I*e^(I*t)
sage: hf(t) = 0.5*e^(-I*t)
sage: hfprime(t) = 0.5*-I*e^(-I*t)
sage: m = Riemann_Map([f, hf], [hf, hfprime], 0.5 + 0.5*I)
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t)).function(t)
>>> __tmp__=var("t"); hf = symbolic_expression(RealNumber('0.5')*e**(-I*t)).
↪function(t)
>>> __tmp__=var("t"); hfprime = symbolic_expression(RealNumber('0.5')*-I*e**(-
↪I*t)).function(t)
>>> m = Riemann_Map([f, hf], [hf, hfprime], RealNumber('0.5') + RealNumber('0.
↪5')*I)
```

Getting the boundary correspondence for a specific boundary:

```
sage: tp0 = m.get_theta_points(boundary=0)
sage: tp1 = m.get_theta_points(boundary=1)
```

```
>>> from sage.all import *
>>> tp0 = m.get_theta_points(boundary=Integer(0))
>>> tp1 = m.get_theta_points(boundary=Integer(1))
```

inverse_riemann_map(*pt*)

Return the inverse Riemann mapping of a point.

That is, given `pt` on the interior of the unit disc, `inverse_riemann_map()` will return the point on the original region that would be Riemann mapped to `pt`. Note that this method does not work for multiply connected domains.

INPUT:

- `pt` – a complex number (usually with absolute value <= 1) representing the point to be inverse mapped

OUTPUT: the point on the region that Riemann maps to the input point

EXAMPLES:

Can work for different types of complex numbers:

```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: m.inverse_riemann_map(0.5 + sqrt(-0.5))
(0.406880...+0.3614702...j)
sage: m.inverse_riemann_map(0.95)
(0.486319...-4.90019052...j)
sage: m.inverse_riemann_map(0.25 - 0.3*I)
(0.1653244...-0.180936...j)
sage: m.inverse_riemann_map(complex(-0.2, 0.5))
(-0.156280...+0.321819...j)
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t) - RealNumber('0.5
↪')*e**(-I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t) + RealNumber('0.
↪5')*I*e**(-I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))
>>> m.inverse_riemann_map(RealNumber('0.5') + sqrt(-RealNumber('0.5')))
```

(continues on next page)

(continued from previous page)

```
(0.406880...+0.3614702...j)
>>> m.inverse_riemann_map(RealNumber('0.95'))
(0.486319...-4.90019052...j)
>>> m.inverse_riemann_map(RealNumber('0.25') - RealNumber('0.3')*I)
(0.1653244...-0.180936...j)
>>> m.inverse_riemann_map(complex(-RealNumber('0.2'), RealNumber('0.5')))
(-0.156280...+0.321819...j)
```

plot_boundaries(*plotjoined=True*, *rgbcolor=[0, 0, 0]*, *thickness=1*)

Plots the boundaries of the region for the Riemann map. Note that this method DOES work for multiply connected domains.

INPUT:

The following inputs may be passed in as named parameters:

- `plotjoined` – boolean (default: `True`); if `False`, discrete points will be drawn; otherwise they will be connected by lines. In this case, if `plotjoined=False`, the points shown will be the original collocation points used to generate the Riemann map.
- `rgbcolor` – float array (default: `[0,0,0]`) the red-green-blue color of the boundary
- `thickness` – positive float (default: `1`) the thickness of the lines or points in the boundary

EXAMPLES:

General usage:

```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t) - RealNumber('0.5
↪')*e**(-I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t) + RealNumber('0.
↪5')*I*e**(-I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))
```

Default plot:

```
sage: m.plot_boundaries()                                                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> m.plot_boundaries()                                                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

Big blue collocation points:

```
sage: m.plot_boundaries(plotjoined=False, rgbcolor=[0,0,1], thickness=6)    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> m.plot_boundaries(plotjoined=False, rgbcolor=[Integer(0),Integer(0),
↪Integer(1)], thickness=Integer(6))     # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

plot_colored(*plot_range=[]*, *plot_points=100*, *interpolation='catrom'*, ***options*)

Generates a colored plot of the Riemann map. A red point on the colored plot corresponds to a red point on the unit disc.

INPUT:

The following inputs may be passed in as named parameters:

- `plot_range` – (default: `[]`) list of 4 values `(xmin, xmax, ymin, ymax)`. Declare if you do not want the plot to use the default range for the figure.
- `plot_points` – integer (default: 100); number of points to plot in the x direction. Points in the y direction are scaled accordingly. Note that very large values can cause this function to run slowly.

EXAMPLES:

Given a Riemann map m, general usage:

```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: m.plot_colored()                                                  #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t) - RealNumber('0.5
↪')*e**(-I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t) + RealNumber('0.
↪5')*I*e**(-I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))
>>> m.plot_colored()                                                  #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

Plot zoomed in on a specific spot:

```
sage: m.plot_colored(plot_range=[0,1,.25,.75])                          #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> m.plot_colored(plot_range=[Integer(0),Integer(1),RealNumber('.25'),
↪RealNumber('.75')])                           # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

High resolution plot:

```
sage: m.plot_colored(plot_points=1000)       # long time (29s on sage.math,␣
↪2012), needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> m.plot_colored(plot_points=Integer(1000))      # long time (29s on sage.
↪math, 2012), needs sage.plot
Graphics object consisting of 1 graphics primitive
```

To generate the unit circle map, it's helpful to see what the colors correspond to:

```
sage: f(t) = e^(I*t)
sage: fprime(t) = I*e^(I*t)
sage: m = Riemann_Map([f], [fprime], 0, 1000)
sage: m.plot_colored()                                                    #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0), Integer(1000))
>>> m.plot_colored()                                                      #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

plot_spiderweb(*spokes=16*, *circles=4*, *pts=32*, *linescale=0.99*, *rgbcolor=[0, 0, 0]*, *thickness=1*, *plotjoined=True*, *withcolor=False*, *plot_points=200*, *min_mag=0.001*, *interpolation='catrom'*, ***options*)

Generate a traditional "spiderweb plot" of the Riemann map.

This shows what concentric circles and radial lines map to. The radial lines may exhibit erratic behavior near the boundary; if this occurs, decreasing `linescale` may mitigate the problem.

For multiply connected domains the spiderweb is by necessity generated using the forward mapping. This method is more computationally intensive. In addition, these spiderwebs cannot be `added` to color plots. Instead the `withcolor` option must be used.

In addition, spiderweb plots are not currently supported for exterior maps.

INPUT:

The following inputs may be passed in as named parameters:

- `spokes` – integer (default: 16); the number of equally spaced radial lines to plot
- `circles` – integer (default: 4); the number of equally spaced circles about the center to plot
- `pts` – integer (default: 32); the number of points to plot. Each radial line is made by `1*pts` points, each circle has `2*pts` points. Note that high values may cause erratic behavior of the radial lines near the boundaries. - only for simply connected domains
- `linescale` – float between 0 and 1; shrinks the radial lines away from the boundary to reduce erratic behavior - only for simply connected domains
- `rgbcolor` – float array (default: `[0,0,0]`); the red-green-blue color of the spiderweb
- `thickness` – positive float (default: 1); the thickness of the lines or points in the spiderweb
- `plotjoined` – boolean (default: `True`); if `False`, discrete points will be drawn; otherwise they will be connected by lines - only for simply connected domains
- `withcolor` – boolean (default: `False`); if `True`, the spiderweb will be overlaid on the basic color plot

- `plot_points` – integer (default: 200); the size of the grid in the x direction. The number of points in the y_direction is scaled accordingly. Note that very large values can cause this function to run slowly. - only for multiply connected domains
- `min_mag` – float (default: 0.001); the magnitude cutoff below which spiderweb points are not drawn. This only applies to multiply connected domains and is designed to prevent "fuzz" at the edge of the domain. Some complicated multiply connected domains (particularly those with corners) may require a larger value to look clean outside.

EXAMPLES:

General usage:

```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t) - RealNumber('0.5
↪')*e**(-I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t) + RealNumber('0.
↪5')*I*e**(-I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))
```

Default plot:

```
sage: m.plot_spiderweb()                                                    #␣
↪needs sage.plot
Graphics object consisting of 21 graphics primitives
```

```
>>> from sage.all import *
>>> m.plot_spiderweb()                                                    #␣
↪needs sage.plot
Graphics object consisting of 21 graphics primitives
```

Simplified plot with many discrete points:

```
sage: m.plot_spiderweb(spokes=4, circles=1, pts=400,                        #␣
↪needs sage.plot
....:                   linescale=0.95, plotjoined=False)
Graphics object consisting of 6 graphics primitives
```

```
>>> from sage.all import *
>>> m.plot_spiderweb(spokes=Integer(4), circles=Integer(1), pts=Integer(400),␣
↪                       # needs sage.plot
...                   linescale=RealNumber('0.95'), plotjoined=False)
Graphics object consisting of 6 graphics primitives
```

Plot with thick, red lines:

```
sage: m.plot_spiderweb(rgbcolor=[1,0,0], thickness=3)                       #␣
↪needs sage.plot
Graphics object consisting of 21 graphics primitives
```

```
>>> from sage.all import *
>>> m.plot_spiderweb(rgbcolor=[Integer(1),Integer(0),Integer(0)],␣
↪thickness=Integer(3))                        # needs sage.plot
Graphics object consisting of 21 graphics primitives
```

To generate the unit circle map, it's helpful to see what the original spiderweb looks like:

```
sage: f(t) = e^(I*t)
sage: fprime(t) = I*e^(I*t)
sage: m = Riemann_Map([f], [fprime], 0, 1000)
sage: m.plot_spiderweb()                                                    #␣
↪needs sage.plot
Graphics object consisting of 21 graphics primitives
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0), Integer(1000))
>>> m.plot_spiderweb()                                                    #␣
↪needs sage.plot
Graphics object consisting of 21 graphics primitives
```

A multiply connected region with corners. We set `min_mag` higher to remove "fuzz" outside the domain:

```
sage: ps = polygon_spline([(-4,-2),(4,-2),(4,2),(-4,2)])
sage: z1 = lambda t: ps.value(t); z1p = lambda t: ps.derivative(t)
sage: z2(t) = -2+exp(-I*t); z2p(t) = -I*exp(-I*t)
sage: z3(t) = 2+exp(-I*t); z3p(t) = -I*exp(-I*t)
sage: m = Riemann_Map([z1,z2,z3], [z1p,z2p,z3p], 0,           # long time
....:                 ncorners=4)
sage: p = m.plot_spiderweb(withcolor=True, plot_points=500,          # long␣
↪time, needs sage.plot
....:                      thickness=2.0, min_mag=0.1)
```

```
>>> from sage.all import *
>>> ps = polygon_spline([(-Integer(4),-Integer(2)),(Integer(4),-Integer(2)),
↪(Integer(4),Integer(2)),(-Integer(4),Integer(2))])
>>> z1 = lambda t: ps.value(t); z1p = lambda t: ps.derivative(t)
>>> __tmp__=var("t"); z2 = symbolic_expression(-Integer(2)+exp(-I*t)).
↪function(t); __tmp__=var("t"); z2p = symbolic_expression(-I*exp(-I*t)).
↪function(t)
>>> __tmp__=var("t"); z3 = symbolic_expression(Integer(2)+exp(-I*t)).
↪function(t); __tmp__=var("t"); z3p = symbolic_expression(-I*exp(-I*t)).
↪function(t)
>>> m = Riemann_Map([z1,z2,z3], [z1p,z2p,z3p], Integer(0),              # long␣
↪time
...                 ncorners=Integer(4))
>>> p = m.plot_spiderweb(withcolor=True, plot_points=Integer(500),          #␣
↪long time, needs sage.plot
...                      thickness=RealNumber('2.0'), min_mag=RealNumber('0.1
↪'))
```

riemann_map(*pt*)

Return the Riemann mapping of a point.

That is, given `pt` on the interior of the mapped region, `riemann_map` will return the point on the unit disk that `pt` maps to. Note that this method only works for interior points; accuracy breaks down very close to the boundary. To get boundary correspondence, use *get_theta_points()*.

INPUT:

- `pt` – a complex number representing the point to be inverse mapped

OUTPUT:

A complex number representing the point on the unit circle that the input point maps to.

EXAMPLES:

Can work for different types of complex numbers:

```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: m.riemann_map(0.25 + sqrt(-0.5))
(0.137514...+0.876696...j)
sage: I = CDF.gen()
sage: m.riemann_map(1.3*I)
(-1.56...e-05+0.989694...j)
sage: m.riemann_map(0.4)
(0.73324...+3.2...e-06j)
sage: m.riemann_map(complex(-3, 0.0001))
(1.405757...e-05+8.06...e-10j)
```

```
>>> from sage.all import *
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t) - RealNumber('0.5
↪')*e**(-I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t) + RealNumber('0.
↪5')*I*e**(-I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))
>>> m.riemann_map(RealNumber('0.25') + sqrt(-RealNumber('0.5')))
(0.137514...+0.876696...j)
>>> I = CDF.gen()
>>> m.riemann_map(RealNumber('1.3')*I)
(-1.56...e-05+0.989694...j)
>>> m.riemann_map(RealNumber('0.4'))
(0.73324...+3.2...e-06j)
>>> m.riemann_map(complex(-Integer(3), RealNumber('0.0001')))
(1.405757...e-05+8.06...e-10j)
```

`sage.calculus.riemann.`**`analytic_boundary`**(*t*, *n*, *epsilon*)

Provides an exact (for $n = \infty$) Riemann boundary correspondence for the ellipse with axes $1 + \epsilon$ and $1 - \epsilon$. The boundary is therefore given by $\exp(It) + \epsilon \exp(-It)$. It is primarily useful for testing the accuracy of the numerical *Riemann_Map*.

INPUT:

- `t` – the boundary parameter, from 0 to 2π
- `n` – integer; the number of terms to include (10 is fairly accurate, 20 is very accurate)

- epsilon – float; the skew of the ellipse (0 is circular)

OUTPUT:

A theta value from 0 to 2*pi, corresponding to the point on the circle e^(I*theta)

sage.calculus.riemann.**analytic_interior**(*z*, *n*, *epsilon*)

Provides a nearly exact computation of the Riemann Map of an interior point of the ellipse with axes 1 + epsilon and 1 - epsilon. It is primarily useful for testing the accuracy of the numerical Riemann Map.

INPUT:

- z – complex; the point to be mapped
- n – integer; the number of terms to include (10 is fairly accurate, 20 is very accurate)

sage.calculus.riemann.**cauchy_kernel**(*t*, *args*)

Intermediate function for the integration in analytic_interior().

INPUT:

- t – the boundary parameter, meant to be integrated over
- args – tuple containing:
  - epsilon – float; the skew of the ellipse (0 is circular)
  - z – complex; the point to be mapped
  - n – integer; the number of terms to include (10 is fairly accurate, 20 is very accurate)
  - part – will return the real ('r'), imaginary ('i') or complex ('c') value of the kernel

sage.calculus.riemann.**complex_to_rgb**(*z_values*)

Convert from a (Numpy) array of complex numbers to its corresponding matrix of RGB values. For internal use of *plot_colored()* only.

INPUT:

- z_values – numpy array of complex numbers

OUTPUT:

An $N \times M \times 3$ floating point Numpy array X, where X[i,j] is an (r,g,b) tuple.

EXAMPLES:

```
sage: from sage.calculus.riemann import complex_to_rgb
sage: import numpy
sage: complex_to_rgb(numpy.array([[0, 1, 1000]], dtype=numpy.complex128))
array([[[1.        , 1.        , 1.        ],
        [1.        , 0.05558355, 0.05558355],
        [0.17301243, 0.        , 0.        ]]])

sage: complex_to_rgb(numpy.array([[0, 1j, 1000j]], dtype=numpy.complex128))
array([[[1.        , 1.        , 1.        ],
        [0.52779177, 1.        , 0.05558355],
        [0.08650622, 0.17301243, 0.        ]]])
```

```
>>> from sage.all import *
>>> from sage.calculus.riemann import complex_to_rgb
>>> import numpy
```

(continues on next page)

(continued from previous page)

```
>>> complex_to_rgb(numpy.array([[Integer(0), Integer(1), Integer(1000)]],␣
↪dtype=numpy.complex128))
array([[[1.        , 1.        , 1.        ],
        [1.        , 0.05558355, 0.05558355],
        [0.17301243, 0.        , 0.        ]]])

>>> complex_to_rgb(numpy.array([[Integer(0), ComplexNumber(0, '1'),␣
↪ComplexNumber(0, '1000')]], dtype=numpy.complex128))
array([[[1.        , 1.        , 1.        ],
        [0.52779177, 1.        , 0.05558355],
        [0.08650622, 0.17301243, 0.        ]]])
```

`sage.calculus.riemann.`**`complex_to_spiderweb`**(*z_values*, *dr*, *dtheta*, *spokes*, *circles*, *rgbcolor*, *thickness*, *withcolor*, *min_mag*)

Convert a grid of complex numbers into a matrix containing rgb data for the Riemann spiderweb plot.

INPUT:

- `z_values` – a grid of complex numbers, as a list of lists
- `dr` – grid of floats, the r derivative of `z_values` Used to determine precision
- `dtheta` – grid of floats, the theta derivative of `z_values` Used to determine precision
- `spokes` – integer; the number of equally spaced radial lines to plot
- `circles` – integer; the number of equally spaced circles about the center to plot
- `rgbcolor` – float array; the red-green-blue color of the lines of the spiderweb
- `thickness` – positive float; the thickness of the lines or points in the spiderweb
- `withcolor` – boolean; if `True` the spiderweb will be overlaid on the basic color plot
- `min_mag` – float; the magnitude cutoff below which spiderweb points are not drawn. This only applies to multiply connected domains and is designed to prevent "fuzz" at the edge of the domain. Some complicated multiply connected domains (particularly those with corners) may require a larger value to look clean outside.

OUTPUT:

An $NxMx3$ floating point Numpy array `X`, where `X[i,j]` is an (r,g,b) tuple.

EXAMPLES:

```
sage: from sage.calculus.riemann import complex_to_spiderweb
sage: import numpy
sage: zval = numpy.array([[0,1,1000], [.2+.3j,1,-.3j], [0,0,0]],
....:                    dtype=numpy.complex128)
sage: deriv = numpy.array([[.1]],dtype = numpy.float64)
sage: complex_to_spiderweb(zval, deriv, deriv, 4, 4, [0,0,0], 1, False, 0.001)
array([[[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],


       [[1., 1., 1.],
        [0., 0., 0.],
        [1., 1., 1.]],
```

(continues on next page)

(continued from previous page)

```
       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]])

sage: complex_to_spiderweb(zval, deriv, deriv, 4, 4, [0,0,0], 1, True, 0.001)
array([[[1.        , 1.        , 1.        ],
        [1.        , 0.05558355, 0.05558355],
        [0.17301243, 0.        , 0.        ]],

       [[1.        , 0.96804683, 0.48044583],
        [0.        , 0.        , 0.        ],
        [0.77351965, 0.5470393 , 1.        ]],

       [[1.        , 1.        , 1.        ],
        [1.        , 1.        , 1.        ],
        [1.        , 1.        , 1.        ]]])
```

```
>>> from sage.all import *
>>> from sage.calculus.riemann import complex_to_spiderweb
>>> import numpy
>>> zval = numpy.array([[Integer(0),Integer(1),Integer(1000)], [RealNumber('.2
↪')+ComplexNumber(0, '.3'),Integer(1),-ComplexNumber(0, '.3')], [Integer(0),
↪Integer(0),Integer(0)]],
...                     dtype=numpy.complex128)
>>> deriv = numpy.array([[RealNumber('.1')]],dtype = numpy.float64)
>>> complex_to_spiderweb(zval, deriv, deriv, Integer(4), Integer(4), [Integer(0),
↪Integer(0),Integer(0)], Integer(1), False, RealNumber('0.001'))
array([[[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],
<BLANKLINE>
       [[1., 1., 1.],
        [0., 0., 0.],
        [1., 1., 1.]],
<BLANKLINE>
       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]])

>>> complex_to_spiderweb(zval, deriv, deriv, Integer(4), Integer(4), [Integer(0),
↪Integer(0),Integer(0)], Integer(1), True, RealNumber('0.001'))
array([[[1.        , 1.        , 1.        ],
        [1.        , 0.05558355, 0.05558355],
        [0.17301243, 0.        , 0.        ]],
<BLANKLINE>
       [[1.        , 0.96804683, 0.48044583],
        [0.        , 0.        , 0.        ],
        [0.77351965, 0.5470393 , 1.        ]],
<BLANKLINE>
       [[1.        , 1.        , 1.        ],
        [1.        , 1.        , 1.        ],
        [1.        , 1.        , 1.        ]]])
```

sage.calculus.riemann.**get_derivatives**(*z_values*, *xstep*, *ystep*)

Compute the r*e^(I*theta) form of derivatives from the grid of points. The derivatives are computed using quick-and-dirty taylor expansion and assuming analyticity. As such `get_derivatives` is primarily intended to be used for comparisons in `plot_spiderweb` and not for applications that require great precision.

INPUT:

- `z_values` – the values for a complex function evaluated on a grid in the complex plane, usually from `compute_on_grid`
- `xstep` – float, the spacing of the grid points in the real direction

OUTPUT:

- A tuple of arrays, [`dr`, `dtheta`], with each array 2 less in both dimensions than `z_values`
  - `dr` – the abs of the derivative of the function in the +r direction
  - `dtheta` – the rate of accumulation of angle in the +theta direction

EXAMPLES:

Standard usage with compute_on_grid:

```
sage: from sage.calculus.riemann import get_derivatives
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: data = m.compute_on_grid([],19)
sage: xstep = (data[2]-data[1])/19
sage: ystep = (data[4]-data[3])/19
sage: dr, dtheta = get_derivatives(data[0],xstep,ystep)
sage: dr[8,8]
0.241...
sage: dtheta[5,5]
5.907...
```

```
>>> from sage.all import *
>>> from sage.calculus.riemann import get_derivatives
>>> __tmp__=var("t"); f = symbolic_expression(e**(I*t) - RealNumber('0.5')*e**(-
↪I*t)).function(t)
>>> __tmp__=var("t"); fprime = symbolic_expression(I*e**(I*t) + RealNumber('0.5
↪')*I*e**(-I*t)).function(t)
>>> m = Riemann_Map([f], [fprime], Integer(0))
>>> data = m.compute_on_grid([],Integer(19))
>>> xstep = (data[Integer(2)]-data[Integer(1)])/Integer(19)
>>> ystep = (data[Integer(4)]-data[Integer(3)])/Integer(19)
>>> dr, dtheta = get_derivatives(data[Integer(0)],xstep,ystep)
>>> dr[Integer(8),Integer(8)]
0.241...
>>> dtheta[Integer(5),Integer(5)]
5.907...
```

2.27 Real Interpolation using GSL

class sage.calculus.interpolation.**Spline**

Bases: object

Create a spline interpolation object.

Given a list v of pairs, s = spline(v) is an object s such that $s(x)$ is the value of the spline interpolation through the points in v at the point x.

The values in v do not have to be sorted. Moreover, one can append values to v, delete values from v, or change values in v, and the spline is recomputed.

EXAMPLES:

```
sage: S = spline([(0, 1), (1, 2), (4, 5), (5, 3)]); S
[(0, 1), (1, 2), (4, 5), (5, 3)]
sage: S(1.5)
2.76136363636...
```

```
>>> from sage.all import *
>>> S = spline([(Integer(0), Integer(1)), (Integer(1), Integer(2)), (Integer(4),␣
↪Integer(5)), (Integer(5), Integer(3))]); S
[(0, 1), (1, 2), (4, 5), (5, 3)]
>>> S(RealNumber('1.5'))
2.76136363636...
```

Changing the points of the spline causes the spline to be recomputed:

```
sage: S[0] = (0, 2); S
[(0, 2), (1, 2), (4, 5), (5, 3)]
sage: S(1.5)
2.507575757575...
```

```
>>> from sage.all import *
>>> S[Integer(0)] = (Integer(0), Integer(2)); S
[(0, 2), (1, 2), (4, 5), (5, 3)]
>>> S(RealNumber('1.5'))
2.507575757575...
```

We may delete interpolation points of the spline:

```
sage: del S[2]; S
[(0, 2), (1, 2), (5, 3)]
sage: S(1.5)
2.04296875
```

```
>>> from sage.all import *
>>> del S[Integer(2)]; S
[(0, 2), (1, 2), (5, 3)]
>>> S(RealNumber('1.5'))
2.04296875
```

We may append to the list of interpolation points:

```
sage: S.append((4, 5)); S
[(0, 2), (1, 2), (5, 3), (4, 5)]
sage: S(1.5)
2.507575757575...
```

```
>>> from sage.all import *
>>> S.append((Integer(4), Integer(5))); S
[(0, 2), (1, 2), (5, 3), (4, 5)]
>>> S(RealNumber('1.5'))
2.507575757575...
```

If we set the n-th interpolation point, where n is larger than `len(S)`, then points $(0, 0)$ will be inserted between the interpolation points and the point to be added:

```
sage: S[6] = (6, 3); S
[(0, 2), (1, 2), (5, 3), (4, 5), (0, 0), (0, 0), (6, 3)]
```

```
>>> from sage.all import *
>>> S[Integer(6)] = (Integer(6), Integer(3)); S
[(0, 2), (1, 2), (5, 3), (4, 5), (0, 0), (0, 0), (6, 3)]
```

This example is in the GSL documentation:

```
sage: v = [(i + RDF(i).sin()/2, i + RDF(i^2).cos()) for i in range(10)]
sage: s = spline(v)
sage: show(point(v) + plot(s,0,9, hue=.8))                                  #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> v = [(i + RDF(i).sin()/Integer(2), i + RDF(i**Integer(2)).cos()) for i in␣
↪range(Integer(10))]
>>> s = spline(v)
>>> show(point(v) + plot(s,Integer(0),Integer(9), hue=RealNumber('.8')))         ␣
↪                            # needs sage.plot
```

We compute the area underneath the spline:

```
sage: s.definite_integral(0, 9)
41.196516041067...
```

```
>>> from sage.all import *
>>> s.definite_integral(Integer(0), Integer(9))
41.196516041067...
```

The definite integral is additive:

```
sage: s.definite_integral(0, 4) + s.definite_integral(4, 9)
41.196516041067...
```

```
>>> from sage.all import *
>>> s.definite_integral(Integer(0), Integer(4)) + s.definite_integral(Integer(4),␣
↪Integer(9))
41.196516041067...
```

Switching the order of the bounds changes the sign of the integral:

```
sage: s.definite_integral(9, 0)
-41.196516041067...
```

```
>>> from sage.all import *
>>> s.definite_integral(Integer(9), Integer(0))
-41.196516041067...
```

We compute the first and second-order derivatives at a few points:

```
sage: s.derivative(5)
-0.1623008526180...
sage: s.derivative(6)
0.2099798628571...
sage: s.derivative(5, order=2)
-3.0874707456138...
sage: s.derivative(6, order=2)
2.6187684827485...
```

```
>>> from sage.all import *
>>> s.derivative(Integer(5))
-0.1623008526180...
>>> s.derivative(Integer(6))
0.2099798628571...
>>> s.derivative(Integer(5), order=Integer(2))
-3.0874707456138...
>>> s.derivative(Integer(6), order=Integer(2))
2.6187684827485...
```

Only the first two derivatives are supported:

```
sage: s.derivative(4, order=3)
Traceback (most recent call last):
...
ValueError: Order of derivative must be 1 or 2.
```

```
>>> from sage.all import *
>>> s.derivative(Integer(4), order=Integer(3))
Traceback (most recent call last):
...
ValueError: Order of derivative must be 1 or 2.
```

append(*xy*)

EXAMPLES:

```
sage: S = spline([(1,1), (2,3), (4,5)]); S.append((5,7)); S
[(1, 1), (2, 3), (4, 5), (5, 7)]
```

```
>>> from sage.all import *
>>> S = spline([(Integer(1),Integer(1)), (Integer(2),Integer(3)), (Integer(4),
↪Integer(5))]); S.append((Integer(5),Integer(7))); S
[(1, 1), (2, 3), (4, 5), (5, 7)]
```

The spline is recomputed when points are appended (Issue #13519):

```
sage: S = spline([(1,1), (2,3), (4,5)]); S
[(1, 1), (2, 3), (4, 5)]
sage: S(3)
4.25
sage: S.append((5, 5)); S
[(1, 1), (2, 3), (4, 5), (5, 5)]
sage: S(3)
4.375
```

```
>>> from sage.all import *
>>> S = spline([(Integer(1),Integer(1)), (Integer(2),Integer(3)), (Integer(4),
↪Integer(5))]); S
[(1, 1), (2, 3), (4, 5)]
>>> S(Integer(3))
4.25
>>> S.append((Integer(5), Integer(5))); S
[(1, 1), (2, 3), (4, 5), (5, 5)]
>>> S(Integer(3))
4.375
```

definite_integral(*a*, *b*)

Value of the definite integral between a and b.

INPUT:

- a – lower bound for the integral
- b – upper bound for the integral

EXAMPLES:

We draw a cubic spline through three points and compute the area underneath the curve:

```
sage: s = spline([(0, 0), (1, 3), (2, 0)])
sage: s.definite_integral(0, 2)
3.75
sage: s.definite_integral(0, 1)
1.875
sage: s.definite_integral(0, 1) + s.definite_integral(1, 2)
3.75
sage: s.definite_integral(2, 0)
-3.75
```

```
>>> from sage.all import *
>>> s = spline([(Integer(0), Integer(0)), (Integer(1), Integer(3)),␣
↪(Integer(2), Integer(0))])
>>> s.definite_integral(Integer(0), Integer(2))
3.75
>>> s.definite_integral(Integer(0), Integer(1))
1.875
>>> s.definite_integral(Integer(0), Integer(1)) + s.definite_
↪integral(Integer(1), Integer(2))
3.75
```

(continues on next page)

(continued from previous page)

```
>>> s.definite_integral(Integer(2), Integer(0))
-3.75
```

derivative(*x*, *order=1*)

Value of the first or second derivative of the spline at x.

INPUT:

- `x` – value at which to evaluate the derivative
- `order` – (default: 1) order of the derivative; must be 1 or 2

EXAMPLES:

We draw a cubic spline through three points and compute the derivatives:

```
sage: s = spline([(0, 0), (2, 3), (4, 0)])
sage: s.derivative(0)
2.25
sage: s.derivative(2)
0.0
sage: s.derivative(4)
-2.25
sage: s.derivative(1, order=2)
-1.125
sage: s.derivative(3, order=2)
-1.125
```

```
>>> from sage.all import *
>>> s = spline([(Integer(0), Integer(0)), (Integer(2), Integer(3)),␣
↪(Integer(4), Integer(0))])
>>> s.derivative(Integer(0))
2.25
>>> s.derivative(Integer(2))
0.0
>>> s.derivative(Integer(4))
-2.25
>>> s.derivative(Integer(1), order=Integer(2))
-1.125
>>> s.derivative(Integer(3), order=Integer(2))
-1.125
```

list()

Underlying list of points that this spline goes through.

EXAMPLES:

```
sage: S = spline([(1,1), (2,3), (4,5)]); S.list()
[(1, 1), (2, 3), (4, 5)]
```

```
>>> from sage.all import *
>>> S = spline([(Integer(1),Integer(1)), (Integer(2),Integer(3)), (Integer(4),
↪Integer(5))]); S.list()
[(1, 1), (2, 3), (4, 5)]
```

This is a copy of the list, not a reference (Issue #13530):

```
sage: S = spline([(1,1), (2,3), (4,5)])
sage: L = S.list(); L
[(1, 1), (2, 3), (4, 5)]
sage: L[2] = (3, 2)
sage: L
[(1, 1), (2, 3), (3, 2)]
sage: S.list()
[(1, 1), (2, 3), (4, 5)]
```

```
>>> from sage.all import *
>>> S = spline([(Integer(1),Integer(1)), (Integer(2),Integer(3)), (Integer(4),
↪Integer(5))])
>>> L = S.list(); L
[(1, 1), (2, 3), (4, 5)]
>>> L[Integer(2)] = (Integer(3), Integer(2))
>>> L
[(1, 1), (2, 3), (3, 2)]
>>> S.list()
[(1, 1), (2, 3), (4, 5)]
```

sage.calculus.interpolation.**spline**

alias of Spline

2.28 Complex Interpolation

AUTHORS:

- Ethan Van Andel (2009): initial version

Development supported by NSF award No. 0702939.

class sage.calculus.interpolators.**CCSpline**

Bases: object

A CCSpline object contains a cubic interpolation of a figure in the complex plane.

EXAMPLES:

A simple square:

```
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: cs = complex_cubic_spline(pts)
sage: cs.value(0)
(-1-1j)
sage: cs.derivative(0)
(0.9549296...-0.9549296...j)
```

```
>>> from sage.all import *
>>> pts = [(-Integer(1), -Integer(1)), (Integer(1), -Integer(1)), (Integer(1),␣
↪Integer(1)), (-Integer(1), Integer(1))]
>>> cs = complex_cubic_spline(pts)
>>> cs.value(Integer(0))
(-1-1j)
```

(continues on next page)

(continued from previous page)

```
>>> cs.derivative(Integer(0))
(0.9549296...-0.9549296...j)
```

derivative(*t*)

Return the derivative (speed and direction of the curve) of a given point from the parameter `t`.

INPUT:

- `t` – double; the parameter value for the parameterized curve, between 0 and 2*pi

OUTPUT:

A complex number representing the derivative at the point on the figure corresponding to the input `t`.

EXAMPLES:

```
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: cs = complex_cubic_spline(pts)
sage: cs.derivative(3 / 5)
(1.40578892327...-0.225417136326...j)
sage: from math import pi
sage: cs.derivative(0) - cs.derivative(2 * pi)
0j
sage: cs.derivative(-6)
(2.52047692949...-1.89392588310...j)
```

```
>>> from sage.all import *
>>> pts = [(-Integer(1), -Integer(1)), (Integer(1), -Integer(1)), (Integer(1),
↪ Integer(1)), (-Integer(1), Integer(1))]
>>> cs = complex_cubic_spline(pts)
>>> cs.derivative(Integer(3) / Integer(5))
(1.40578892327...-0.225417136326...j)
>>> from math import pi
>>> cs.derivative(Integer(0)) - cs.derivative(Integer(2) * pi)
0j
>>> cs.derivative(-Integer(6))
(2.52047692949...-1.89392588310...j)
```

value(*t*)

Return the location of a given point from the parameter `t`.

INPUT:

- `t` – double; the parameter value for the parameterized curve, between 0 and 2*pi

OUTPUT:

A complex number representing the point on the figure corresponding to the input `t`.

EXAMPLES:

```
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: cs = complex_cubic_spline(pts)
sage: cs.value(4 / 7)
(-0.303961332787...-1.34716728183...j)
sage: from math import pi
sage: cs.value(0) - cs.value(2*pi)
```

(continues on next page)

(continued from previous page)

```
0j
sage: cs.value(-2.73452)
(0.934561222231...+0.881366116402...j)
```

```
>>> from sage.all import *
>>> pts = [(-Integer(1), -Integer(1)), (Integer(1), -Integer(1)), (Integer(1),
↪ Integer(1)), (-Integer(1), Integer(1))]
>>> cs = complex_cubic_spline(pts)
>>> cs.value(Integer(4) / Integer(7))
(-0.303961332787...-1.34716728183...j)
>>> from math import pi
>>> cs.value(Integer(0)) - cs.value(Integer(2)*pi)
0j
>>> cs.value(-RealNumber('2.73452'))
(0.934561222231...+0.881366116402...j)
```

class sage.calculus.interpolators.**PSpline**

Bases: `object`

A `CCSpline` object contains a polygon interpolation of a figure in the complex plane.

EXAMPLES:

A simple `square`:

```
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: ps = polygon_spline(pts)
sage: ps.value(0)
(-1-1j)
sage: ps.derivative(0)
(1.27323954...+0j)
```

```
>>> from sage.all import *
>>> pts = [(-Integer(1), -Integer(1)), (Integer(1), -Integer(1)), (Integer(1),␣
↪Integer(1)), (-Integer(1), Integer(1))]
>>> ps = polygon_spline(pts)
>>> ps.value(Integer(0))
(-1-1j)
>>> ps.derivative(Integer(0))
(1.27323954...+0j)
```

derivative(*t*)

Return the derivative (speed and direction of the curve) of a given point from the parameter `t`.

INPUT:

- `t` – double; the parameter value for the parameterized curve, between 0 and 2*pi

OUTPUT:

A complex number representing the derivative at the point on the polygon corresponding to the input `t`.

EXAMPLES:

```
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: ps = polygon_spline(pts)
sage: ps.derivative(1 / 3)
(1.27323954473...+0j)
sage: from math import pi
sage: ps.derivative(0) - ps.derivative(2*pi)
0j
sage: ps.derivative(10)
(-1.27323954473...+0j)
```

```
>>> from sage.all import *
>>> pts = [(-Integer(1), -Integer(1)), (Integer(1), -Integer(1)), (Integer(1),
↪ Integer(1)), (-Integer(1), Integer(1))]
>>> ps = polygon_spline(pts)
>>> ps.derivative(Integer(1) / Integer(3))
(1.27323954473...+0j)
>>> from math import pi
>>> ps.derivative(Integer(0)) - ps.derivative(Integer(2)*pi)
0j
>>> ps.derivative(Integer(10))
(-1.27323954473...+0j)
```

value(*t*)

Return the derivative (speed and direction of the curve) of a given point from the parameter `t`.

INPUT:

- `t` – double; the parameter value for the parameterized curve, between 0 and 2*pi

OUTPUT:

A complex number representing the point on the polygon corresponding to the input `t`.

EXAMPLES:

```
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: ps = polygon_spline(pts)
sage: ps.value(.5)
(-0.363380227632...-1j)
sage: ps.value(0) - ps.value(2*RDF.pi())
0j
sage: ps.value(10)
(0.26760455264...+1j)
```

```
>>> from sage.all import *
>>> pts = [(-Integer(1), -Integer(1)), (Integer(1), -Integer(1)), (Integer(1),
↪ Integer(1)), (-Integer(1), Integer(1))]
>>> ps = polygon_spline(pts)
>>> ps.value(RealNumber('.5'))
(-0.363380227632...-1j)
>>> ps.value(Integer(0)) - ps.value(Integer(2)*RDF.pi())
0j
>>> ps.value(Integer(10))
(0.26760455264...+1j)
```

sage.calculus.interpolators.**complex_cubic_spline**(*pts*)

Create a cubic spline interpolated figure from a set of complex or (x, y) points. The figure will be a parametric curve from 0 to 2*pi. The returned values will be complex, not (x, y).

INPUT:

- `pts` – list or array of complex numbers, or tuples of the form (x, y)

EXAMPLES:

A simple `square`:

```
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: cs = complex_cubic_spline(pts)
sage: fx = lambda x: cs.value(x).real
sage: fy = lambda x: cs.value(x).imag
sage: from math import pi
sage: show(parametric_plot((fx, fy), (0, 2*pi)))                       #␣
↪needs sage.plot
sage: m = Riemann_Map([lambda x: cs.value(real(x))],
....:                 [lambda x: cs.derivative(real(x))], 0)
sage: show(m.plot_colored() + m.plot_spiderweb())                       #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> pts = [(-Integer(1), -Integer(1)), (Integer(1), -Integer(1)), (Integer(1),␣
↪Integer(1)), (-Integer(1), Integer(1))]
>>> cs = complex_cubic_spline(pts)
>>> fx = lambda x: cs.value(x).real
>>> fy = lambda x: cs.value(x).imag
>>> from math import pi
>>> show(parametric_plot((fx, fy), (Integer(0), Integer(2)*pi)))         ␣
↪               # needs sage.plot
>>> m = Riemann_Map([lambda x: cs.value(real(x))],
...                 [lambda x: cs.derivative(real(x))], Integer(0))
>>> show(m.plot_colored() + m.plot_spiderweb())                         #␣
↪needs sage.plot
```

Polygon approximation of a circle:

```
sage: from cmath import exp
sage: pts = [exp(1j * t / 25) for t in range(25)]
sage: cs = complex_cubic_spline(pts)
sage: cs.derivative(2)
(-0.0497765406583...+0.151095006434...j)
```

```
>>> from sage.all import *
>>> from cmath import exp
>>> pts = [exp(ComplexNumber(0, '1') * t / Integer(25)) for t in␣
↪range(Integer(25))]
>>> cs = complex_cubic_spline(pts)
>>> cs.derivative(Integer(2))
(-0.0497765406583...+0.151095006434...j)
```

sage.calculus.interpolators.**polygon_spline**(*pts*)

Create a polygon from a set of complex or (x, y) points. The polygon will be a parametric curve from 0 to 2*pi. The returned values will be complex, not (x, y).

INPUT:

- `pts` – list or array of complex numbers of tuples of the form (x, y)

EXAMPLES:

A simple square:

```
sage: pts = [(-1, -1), (1, -1), (1, 1), (-1, 1)]
sage: ps = polygon_spline(pts)
sage: fx = lambda x: ps.value(x).real
sage: fy = lambda x: ps.value(x).imag
sage: show(parametric_plot((fx, fy), (0, 2*pi)))                               #␣
↪needs sage.plot sage.symbolic
sage: m = Riemann_Map([lambda x: ps.value(real(x))],
....:                 [lambda x: ps.derivative(real(x))], 0)
sage: show(m.plot_colored() + m.plot_spiderweb())                              #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> pts = [(-Integer(1), -Integer(1)), (Integer(1), -Integer(1)), (Integer(1),␣
↪Integer(1)), (-Integer(1), Integer(1))]
>>> ps = polygon_spline(pts)
>>> fx = lambda x: ps.value(x).real
>>> fy = lambda x: ps.value(x).imag
>>> show(parametric_plot((fx, fy), (Integer(0), Integer(2)*pi)))               ␣
↪              # needs sage.plot sage.symbolic
>>> m = Riemann_Map([lambda x: ps.value(real(x))],
...                 [lambda x: ps.derivative(real(x))], Integer(0))
>>> show(m.plot_colored() + m.plot_spiderweb())                              #␣
↪needs sage.plot
```

Polygon approximation of a circle:

```
sage: # needs sage.symbolic
sage: pts = [e^(I*t / 25) for t in range(25)]
sage: ps = polygon_spline(pts)
sage: ps.derivative(2)
(-0.0470303661...+0.1520363883...j)
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> pts = [e**(I*t / Integer(25)) for t in range(Integer(25))]
>>> ps = polygon_spline(pts)
>>> ps.derivative(Integer(2))
(-0.0470303661...+0.1520363883...j)
```

2.29 Calculus functions

`sage.calculus.functions.`**`jacobian`**(*functions*, *variables*)

Return the Jacobian matrix, which is the matrix of partial derivatives in which the i,j entry of the Jacobian matrix is the partial derivative diff(functions[i], variables[j]).

EXAMPLES:

```
sage: x,y = var('x,y')
sage: g=x^2-2*x*y
sage: jacobian(g, (x,y))
[2*x - 2*y      -2*x]
```

```
>>> from sage.all import *
>>> x,y = var('x,y')
>>> g=x**Integer(2)-Integer(2)*x*y
>>> jacobian(g, (x,y))
[2*x - 2*y      -2*x]
```

The Jacobian of the Jacobian should give us the "second derivative", which is the Hessian matrix:

```
sage: jacobian(jacobian(g, (x,y)), (x,y))
[ 2 -2]
[-2  0]
sage: g.hessian()
[ 2 -2]
[-2  0]

sage: f=(x^3*sin(y), cos(x)*sin(y), exp(x))
sage: jacobian(f, (x,y))
[   3*x^2*sin(y)       x^3*cos(y)]
[-sin(x)*sin(y)   cos(x)*cos(y)]
[            e^x               0]
sage: jacobian(f, (y,x))
[    x^3*cos(y)    3*x^2*sin(y)]
[ cos(x)*cos(y)  -sin(x)*sin(y)]
[              0             e^x]
```

```
>>> from sage.all import *
>>> jacobian(jacobian(g, (x,y)), (x,y))
[ 2 -2]
[-2  0]
>>> g.hessian()
[ 2 -2]
[-2  0]

>>> f=(x**Integer(3)*sin(y), cos(x)*sin(y), exp(x))
>>> jacobian(f, (x,y))
[   3*x^2*sin(y)       x^3*cos(y)]
[-sin(x)*sin(y)   cos(x)*cos(y)]
[            e^x               0]
>>> jacobian(f, (y,x))
[    x^3*cos(y)    3*x^2*sin(y)]
```

(continues on next page)

(continued from previous page)

```
[ cos(x)*cos(y) -sin(x)*sin(y)]
[             0            e^x]
```

sage.calculus.functions.**wronskian**(**args*)

Return the Wronskian of the provided functions, differentiating with respect to the given variable.

If no variable is provided, diff(f) is called for each function f.

wronskian(f1,…,fn, x) returns the Wronskian of f1,…,fn, with derivatives taken with respect to x.

wronskian(f1,…,fn) returns the Wronskian of f1,…,fn where k'th derivatives are computed by doing . `derivative(k)` on each function.

The Wronskian of a list of functions is a determinant of derivatives. The n-th row (starting from 0) is a list of the n-th derivatives of the given functions.

For two functions:

```
             | f   g  |
W(f, g) = det|        | = f*g' - g*f'.
             | f'  g' |
```

EXAMPLES:

```
sage: wronskian(e^x, x^2)
-x^2*e^x + 2*x*e^x

sage: x,y = var('x, y')
sage: wronskian(x*y, log(x), x)
-y*log(x) + y
```

```
>>> from sage.all import *
>>> wronskian(e**x, x**Integer(2))
-x^2*e^x + 2*x*e^x

>>> x,y = var('x, y')
>>> wronskian(x*y, log(x), x)
-y*log(x) + y
```

If your functions are in a list, you can use $*'toturnthemintoargumentsto:func:`wronskian$:

```
sage: wronskian(*[x^k for k in range(1, 5)])
12*x^4
```

```
>>> from sage.all import *
>>> wronskian(*[x**k for k in range(Integer(1), Integer(5))])
12*x^4
```

If you want to use 'x' as one of the functions in the Wronskian, you can't put it last or it will be interpreted as the variable with respect to which we differentiate. There are several ways to get around this.

Two-by-two Wronskian of sin(x) and e^x:

```
sage: wronskian(sin(x), e^x, x)
-cos(x)*e^x + e^x*sin(x)
```

```
>>> from sage.all import *
>>> wronskian(sin(x), e**x, x)
-cos(x)*e^x + e^x*sin(x)
```

Or don't put x last:

```
sage: wronskian(x, sin(x), e^x)
(cos(x)*e^x + e^x*sin(x))*x - 2*e^x*sin(x)
```

```
>>> from sage.all import *
>>> wronskian(x, sin(x), e**x)
(cos(x)*e^x + e^x*sin(x))*x - 2*e^x*sin(x)
```

Example where one of the functions is constant:

```
sage: wronskian(1, e^(-x), e^(2*x))
-6*e^x
```

```
>>> from sage.all import *
>>> wronskian(Integer(1), e**(-x), e**(Integer(2)*x))
-6*e^x
```

REFERENCES:

- Wikipedia article Wronskian
- http://planetmath.org/encyclopedia/WronskianDeterminant.html

AUTHORS:

- Dan Drake (2008-03-12)

2.30 Symbolic variables

sage.calculus.var.**clear_vars**()

Delete all 1-letter symbolic variables that are predefined at startup of Sage.

Any one-letter global variables that are not symbolic variables are not cleared.

EXAMPLES:

```
sage: var('x y z')
(x, y, z)
sage: (x+y)^z
(x + y)^z
sage: k = 15
sage: clear_vars()
sage: (x+y)^z
Traceback (most recent call last):
...
NameError: name 'x' is not defined
sage: expand((e + i)^2)
e^2 + 2*I*e - 1
sage: k
15
```

```
>>> from sage.all import *
>>> var('x y z')
(x, y, z)
>>> (x+y)**z
(x + y)^z
>>> k = Integer(15)
>>> clear_vars()
>>> (x+y)**z
Traceback (most recent call last):
...
NameError: name 'x' is not defined
>>> expand((e + i)**Integer(2))
e^2 + 2*I*e - 1
>>> k
15
```

sage.calculus.var.**function**(*s*, ***kwds*)

Create a formal symbolic function with the name *s*.

INPUT:

- `nargs=0` – number of arguments the function accepts, defaults to variable number of arguments, or 0
- `latex_name` – name used when printing in latex mode
- `conversions` – dictionary specifying names of this function in other systems, this is used by the interfaces internally during conversion
- `eval_func` – method used for automatic evaluation
- `evalf_func` – method used for numeric evaluation
- `evalf_params_first` – boolean to indicate if parameters should be evaluated numerically before calling the custom evalf function
- `conjugate_func` – method used for complex conjugation
- `real_part_func` – method used when taking real parts
- `imag_part_func` – method used when taking imaginary parts
- `derivative_func` – method to be used for (partial) derivation This method should take a keyword argument deriv_param specifying the index of the argument to differentiate w.r.t
- `tderivative_func` – method to be used for derivatives
- `power_func` – method used when taking powers This method should take a keyword argument power_param specifying the exponent
- `series_func` – method used for series expansion This method should expect keyword arguments - `order` – order for the expansion to be computed - `var` – variable to expand w.r.t. - `at` – expand at this value
- `print_func` – method for custom printing
- `print_latex_func` – method for custom printing in latex mode

Note that custom methods must be instance methods, i.e., expect the instance of the symbolic function as the first argument.

Note

The new function is both returned and automatically injected into the global namespace. If you use this function in library code, it is better to use sage.symbolic.function_factory.function, since it will not touch the global namespace.

EXAMPLES:

We create a formal function called supersin

```
sage: function('supersin')
supersin
```

```
>>> from sage.all import *
>>> function('supersin')
supersin
```

We can immediately use supersin in symbolic expressions:

```
sage: y, z, A = var('y z A')
sage: supersin(y+z) + A^3
A^3 + supersin(y + z)
```

```
>>> from sage.all import *
>>> y, z, A = var('y z A')
>>> supersin(y+z) + A**Integer(3)
A^3 + supersin(y + z)
```

We can define other functions in terms of supersin:

```
sage: g(x,y) = supersin(x)^2 + sin(y/2)
sage: g
(x, y) |--> supersin(x)^2 + sin(1/2*y)
sage: g.diff(y)
(x, y) |--> 1/2*cos(1/2*y)
sage: k = g.diff(x); k
(x, y) |--> 2*supersin(x)*diff(supersin(x), x)
```

```
>>> from sage.all import *
>>> __tmp__=var("x,y"); g = symbolic_expression(supersin(x)**Integer(2) + sin(y/
↪Integer(2))).function(x,y)
>>> g
(x, y) |--> supersin(x)^2 + sin(1/2*y)
>>> g.diff(y)
(x, y) |--> 1/2*cos(1/2*y)
>>> k = g.diff(x); k
(x, y) |--> 2*supersin(x)*diff(supersin(x), x)
```

We create a formal function of one variable, write down an expression that involves first and second derivatives, and extract off coefficients:

```
sage: r, kappa = var('r,kappa')
sage: psi = function('psi', nargs=1)(r); psi
```

(continues on next page)

(continued from previous page)

```
psi(r)
sage: g = 1/r^2*(2*r*psi.derivative(r,1) + r^2*psi.derivative(r,2)); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
sage: g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
sage: g.coefficient(psi.derivative(r,2))
1
sage: g.coefficient(psi.derivative(r,1))
2/r
```

```
>>> from sage.all import *
>>> r, kappa = var('r,kappa')
>>> psi = function('psi', nargs=Integer(1))(r); psi
psi(r)
>>> g = Integer(1)/r**Integer(2)*(Integer(2)*r*psi.derivative(r,Integer(1)) +␣
↪r**Integer(2)*psi.derivative(r,Integer(2))); g
(r^2*diff(psi(r), r, r) + 2*r*diff(psi(r), r))/r^2
>>> g.expand()
2*diff(psi(r), r)/r + diff(psi(r), r, r)
>>> g.coefficient(psi.derivative(r,Integer(2)))
1
>>> g.coefficient(psi.derivative(r,Integer(1)))
2/r
```

Custom typesetting of symbolic functions in LaTeX, either using latex_name keyword:

```
sage: function('riemann', latex_name='\\mathcal{R}')
riemann
sage: latex(riemann(x))
\mathcal{R}\left(x\right)
```

```
>>> from sage.all import *
>>> function('riemann', latex_name='\\mathcal{R}')
riemann
>>> latex(riemann(x))
\mathcal{R}\left(x\right)
```

or passing a custom callable function that returns a latex expression:

```
sage: mu,nu = var('mu,nu')
sage: def my_latex_print(self, *args): return "\\psi_{%s}"%(', '.join(map(latex,␣
↪args)))
sage: function('psi', print_latex_func=my_latex_print)
psi
sage: latex(psi(mu,nu))
\psi_{\mu, \nu}
```

```
>>> from sage.all import *
>>> mu,nu = var('mu,nu')
>>> def my_latex_print(self, *args): return "\\psi_{%s}"%(', '.join(map(latex,␣
↪args)))
>>> function('psi', print_latex_func=my_latex_print)
```

(continues on next page)

(continued from previous page)

```
psi
>>> latex(psi(mu,nu))
\psi_{\mu, \nu}
```

Defining custom methods for automatic or numeric evaluation, derivation, conjugation, etc. is supported:

```
sage: def ev(self, x): return 2*x
sage: foo = function("foo", nargs=1, eval_func=ev)
sage: foo(x)
2*x
sage: foo = function("foo", nargs=1, eval_func=lambda self, x: 5)
sage: foo(x)
5
sage: def ef(self, x): pass
sage: bar = function("bar", nargs=1, eval_func=ef)
sage: bar(x)
bar(x)

sage: def evalf_f(self, x, parent=None, algorithm=None): return 6
sage: foo = function("foo", nargs=1, evalf_func=evalf_f)
sage: foo(x)
foo(x)
sage: foo(x).n()
6

sage: foo = function("foo", nargs=1, conjugate_func=ev)
sage: foo(x).conjugate()
2*x

sage: def deriv(self, *args, **kwds): print("{} {}".format(args, kwds)); return␣
↪args[kwds['diff_param']]^2
sage: foo = function("foo", nargs=2, derivative_func=deriv)
sage: foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2

sage: def pow(self, x, power_param=None): print("{} {}".format(x, power_param));␣
↪return x*power_param
sage: foo = function("foo", nargs=1, power_func=pow)
sage: foo(y)^(x+y)
y x + y
(x + y)*y

sage: from pprint import pformat
sage: def expand(self, *args, **kwds):
....:     print("{} {}".format(args, pformat(kwds)))
....:     return sum(args[0]^i for i in range(kwds['order']))
sage: foo = function("foo", nargs=1, series_func=expand)
sage: foo(y).series(y, 5)
(y,) {'at': 0, 'options': 0, 'order': 5, 'var': y}
y^4 + y^3 + y^2 + y + 1
```

(continues on next page)

(continued from previous page)

```
sage: def my_print(self, *args):
....:     return "my args are: " + ', '.join(map(repr, args))
sage: foo = function('t', nargs=2, print_func=my_print)
sage: foo(x,y^z)
my args are: x, y^z

sage: latex(foo(x,y^z))
t\left(x, y^{z}\right)
sage: foo = function('t', nargs=2, print_latex_func=my_print)
sage: foo(x,y^z)
t(x, y^z)
sage: latex(foo(x,y^z))
my args are: x, y^z
sage: foo = function('t', nargs=2, latex_name='foo')
sage: latex(foo(x,y^z))
foo\left(x, y^{z}\right)
```

```
>>> from sage.all import *
>>> def ev(self, x): return Integer(2)*x
>>> foo = function("foo", nargs=Integer(1), eval_func=ev)
>>> foo(x)
2*x
>>> foo = function("foo", nargs=Integer(1), eval_func=lambda self, x: Integer(5))
>>> foo(x)
5
>>> def ef(self, x): pass
>>> bar = function("bar", nargs=Integer(1), eval_func=ef)
>>> bar(x)
bar(x)

>>> def evalf_f(self, x, parent=None, algorithm=None): return Integer(6)
>>> foo = function("foo", nargs=Integer(1), evalf_func=evalf_f)
>>> foo(x)
foo(x)
>>> foo(x).n()
6

>>> foo = function("foo", nargs=Integer(1), conjugate_func=ev)
>>> foo(x).conjugate()
2*x

>>> def deriv(self, *args, **kwds): print("{} {}".format(args, kwds)); return␣
↪args[kwds['diff_param']]**Integer(2)
>>> foo = function("foo", nargs=Integer(2), derivative_func=deriv)
>>> foo(x,y).derivative(y)
(x, y) {'diff_param': 1}
y^2

>>> def pow(self, x, power_param=None): print("{} {}".format(x, power_param));␣
↪return x*power_param
>>> foo = function("foo", nargs=Integer(1), power_func=pow)
>>> foo(y)**(x+y)
```

(continues on next page)

(continued from previous page)

```
y x + y
(x + y)*y

>>> from pprint import pformat
>>> def expand(self, *args, **kwds):
...     print("{} {}".format(args, pformat(kwds)))
...     return sum(args[Integer(0)]**i for i in range(kwds['order']))
>>> foo = function("foo", nargs=Integer(1), series_func=expand)
>>> foo(y).series(y, Integer(5))
(y,) {'at': 0, 'options': 0, 'order': 5, 'var': y}
y^4 + y^3 + y^2 + y + 1

>>> def my_print(self, *args):
...     return "my args are: " + ', '.join(map(repr, args))
>>> foo = function('t', nargs=Integer(2), print_func=my_print)
>>> foo(x,y**z)
my args are: x, y^z

>>> latex(foo(x,y**z))
t\left(x, y^{z}\right)
>>> foo = function('t', nargs=Integer(2), print_latex_func=my_print)
>>> foo(x,y**z)
t(x, y^z)
>>> latex(foo(x,y**z))
my args are: x, y^z
>>> foo = function('t', nargs=Integer(2), latex_name='foo')
>>> latex(foo(x,y**z))
foo\left(x, y^{z}\right)
```

Chain rule:

```
sage: def print_args(self, *args, **kwds): print("args: {}".format(args)); print(
↪"kwds: {}".format(kwds)); return args[0]
sage: foo = function('t', nargs=2, tderivative_func=print_args)
sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': x}
x
sage: foo = function('t', nargs=2, derivative_func=print_args)
sage: foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
args: (x, x)
kwds: {'diff_param': 1}
2*x
```

```
>>> from sage.all import *
>>> def print_args(self, *args, **kwds): print("args: {}".format(args)); print(
↪"kwds: {}".format(kwds)); return args[Integer(0)]
>>> foo = function('t', nargs=Integer(2), tderivative_func=print_args)
>>> foo(x,x).derivative(x)
args: (x, x)
```

(continues on next page)

(continued from previous page)

```
kwds: {'diff_param': x}
x
>>> foo = function('t', nargs=Integer(2), derivative_func=print_args)
>>> foo(x,x).derivative(x)
args: (x, x)
kwds: {'diff_param': 0}
args: (x, x)
kwds: {'diff_param': 1}
2*x
```

Since Sage 4.0, basic arithmetic with unevaluated functions is no longer supported:

```
sage: x = var('x')
sage: f = function('f')
sage: 2*f
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring' and '<class 'sage.
↪symbolic.function_factory...NewSymbolicFunction'>'
```

```
>>> from sage.all import *
>>> x = var('x')
>>> f = function('f')
>>> Integer(2)*f
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Integer Ring' and '<class 'sage.
↪symbolic.function_factory...NewSymbolicFunction'>'
```

You now need to evaluate the function in order to do the arithmetic:

```
sage: 2*f(x)
2*f(x)
```

```
>>> from sage.all import *
>>> Integer(2)*f(x)
2*f(x)
```

Since Sage 4.0, you need to use `substitute_function()` to replace all occurrences of a function with another:

```
sage: var('a, b')
(a, b)
sage: cr = function('cr')
sage: f = cr(a)
sage: g = f.diff(a).integral(b)
sage: g
b*diff(cr(a), a)
sage: g.substitute_function(cr, cos)
-b*sin(a)

sage: g.substitute_function(cr, (sin(x) + cos(x)).function(x))
b*(cos(a) - sin(a))
```

```
>>> from sage.all import *
>>> var('a, b')
(a, b)
>>> cr = function('cr')
>>> f = cr(a)
>>> g = f.diff(a).integral(b)
>>> g
b*diff(cr(a), a)
>>> g.substitute_function(cr, cos)
-b*sin(a)

>>> g.substitute_function(cr, (sin(x) + cos(x)).function(x))
b*(cos(a) - sin(a))
```

`sage.calculus.var.var(*args, **kwds)`

Create a symbolic variable with the name *s*.

INPUT:

- `args` – a single string `var('x y')`, a list of strings `var(['x','y'])`, or multiple strings `var('x', 'y')`. A single string can be either a single variable name, or a space or comma separated list of variable names. In a list or tuple of strings, each entry is one variable. If multiple arguments are specified, each argument is taken to be one variable. Spaces before or after variable names are ignored.
- `kwds` – keyword arguments can be given to specify domain and custom latex_name for variables; see EXAMPLES for usage

Note

The new variable is both returned and automatically injected into the global namespace. If you need a symbolic variable in library code, you must use either `SR.var()` or `SR.symbol()`.

OUTPUT:

If a single symbolic variable was created, the variable itself. Otherwise, a tuple of symbolic variables. The variable names are checked to be valid Python identifiers and a `ValueError` is raised otherwise.

EXAMPLES:

Here are the different ways to define three variables `x`, `y`, and `z` in a single line:

```
sage: var('x y z')
(x, y, z)
sage: var('x, y, z')
(x, y, z)
sage: var(['x', 'y', 'z'])
(x, y, z)
sage: var('x', 'y', 'z')
(x, y, z)
sage: var('x'), var('y'), var(z)
(x, y, z)
```

```
>>> from sage.all import *
>>> var('x y z')
```

(continues on next page)

(continued from previous page)

```
(x, y, z)
>>> var('x, y, z')
(x, y, z)
>>> var(['x', 'y', 'z'])
(x, y, z)
>>> var('x', 'y', 'z')
(x, y, z)
>>> var('x'), var('y'), var(z)
(x, y, z)
```

We define some symbolic variables:

```
sage: var('n xx yy zz')
(n, xx, yy, zz)
```

```
>>> from sage.all import *
>>> var('n xx yy zz')
(n, xx, yy, zz)
```

Then we make an algebraic expression out of them:

```
sage: f = xx^n + yy^n + zz^n; f
xx^n + yy^n + zz^n
```

```
>>> from sage.all import *
>>> f = xx**n + yy**n + zz**n; f
xx^n + yy^n + zz^n
```

By default, var returns a complex variable. To define real or positive variables we can specify the domain as:

```
sage: x = var('x', domain=RR); x; x.conjugate()
x
x
sage: y = var('y', domain='real'); y.conjugate()
y
sage: y = var('y', domain='positive'); y.abs()
y
```

```
>>> from sage.all import *
>>> x = var('x', domain=RR); x; x.conjugate()
x
x
>>> y = var('y', domain='real'); y.conjugate()
y
>>> y = var('y', domain='positive'); y.abs()
y
```

Custom latex expression can be assigned to variable:

```
sage: x = var('sui', latex_name='s_{u,i}'); x._latex_()
'{s_{u,i}}'
```

```
>>> from sage.all import *
>>> x = var('sui', latex_name='s_{u,i}'); x._latex_()
'{s_{u,i}}'
```

In notebook, we can also colorize latex expression:

```
sage: x = var('sui', latex_name='\\color{red}{s_{u,i}}'); x._latex_()
'{\\color{red}{s_{u,i}}}'
```

```
>>> from sage.all import *
>>> x = var('sui', latex_name='\\color{red}{s_{u,i}}'); x._latex_()
'{\\color{red}{s_{u,i}}}'
```

We can substitute a new variable name for n:

```
sage: f(n = var('sigma'))
xx^sigma + yy^sigma + zz^sigma
```

```
>>> from sage.all import *
>>> f(n = var('sigma'))
xx^sigma + yy^sigma + zz^sigma
```

If you make an important built-in variable into a symbolic variable, you can get back the original value using restore:

```
sage: var('QQ RR')
(QQ, RR)
sage: QQ
QQ
sage: restore('QQ')
sage: QQ
Rational Field
```

```
>>> from sage.all import *
>>> var('QQ RR')
(QQ, RR)
>>> QQ
QQ
>>> restore('QQ')
>>> QQ
Rational Field
```

We make two new variables separated by commas:

```
sage: var('theta, gamma')
(theta, gamma)
sage: theta^2 + gamma^3
gamma^3 + theta^2
```

```
>>> from sage.all import *
>>> var('theta, gamma')
(theta, gamma)
>>> theta**Integer(2) + gamma**Integer(3)
gamma^3 + theta^2
```

The new variables are of type Expression, and belong to the symbolic expression ring:

```
sage: type(theta)
<class 'sage.symbolic.expression.Expression'>
sage: parent(theta)
Symbolic Ring
```

```
>>> from sage.all import *
>>> type(theta)
<class 'sage.symbolic.expression.Expression'>
>>> parent(theta)
Symbolic Ring
```

2.31 Access to Maxima methods

class sage.symbolic.maxima_wrapper.**MaximaFunctionElementWrapper**(*obj*, *name*)

Bases: InterfaceFunctionElement

class sage.symbolic.maxima_wrapper.**MaximaWrapper**(*exp*)

Bases: SageObject

Wrapper around Sage expressions to give access to Maxima methods.

We convert the given expression to Maxima and convert the return value back to a Sage expression. Tab completion and help strings of Maxima methods also work as expected.

EXAMPLES:

```
sage: t = log(sqrt(2) - 1) + log(sqrt(2) + 1); t
log(sqrt(2) + 1) + log(sqrt(2) - 1)
sage: u = t.maxima_methods(); u
MaximaWrapper(log(sqrt(2) + 1) + log(sqrt(2) - 1))
sage: type(u)
<class 'sage.symbolic.maxima_wrapper.MaximaWrapper'>
sage: u.logcontract()
log((sqrt(2) + 1)*(sqrt(2) - 1))
sage: u.logcontract().parent()
Symbolic Ring
```

```
>>> from sage.all import *
>>> t = log(sqrt(Integer(2)) - Integer(1)) + log(sqrt(Integer(2)) + Integer(1)); t
log(sqrt(2) + 1) + log(sqrt(2) - 1)
>>> u = t.maxima_methods(); u
MaximaWrapper(log(sqrt(2) + 1) + log(sqrt(2) - 1))
>>> type(u)
<class 'sage.symbolic.maxima_wrapper.MaximaWrapper'>
>>> u.logcontract()
log((sqrt(2) + 1)*(sqrt(2) - 1))
>>> u.logcontract().parent()
Symbolic Ring
```

sage()

Return the Sage expression this wrapper corresponds to.

EXAMPLES:

```
sage: t = log(sqrt(2) - 1) + log(sqrt(2) + 1); t
log(sqrt(2) + 1) + log(sqrt(2) - 1)
sage: u = t.maxima_methods().sage()
sage: u is t
True
```

```
>>> from sage.all import *
>>> t = log(sqrt(Integer(2)) - Integer(1)) + log(sqrt(Integer(2)) +
↪Integer(1)); t
log(sqrt(2) + 1) + log(sqrt(2) - 1)
>>> u = t.maxima_methods().sage()
>>> u is t
True
```

2.32 Operators

class sage.symbolic.operators.**DerivativeOperator**

Bases: object

Derivative operator.

Acting with this operator onto a function gives a new operator (of type *FDerivativeOperator*) representing the function differentiated with respect to one or multiple of its arguments.

This operator takes a list of indices specifying the position of the arguments to differentiate. For example, D[0, 0, 1] is an operator that differentiates a function twice with respect to its first argument and once with respect to its second argument.

EXAMPLES:

```
sage: x, y = var('x,y'); f = function('f')
sage: D[0](f)(x)
diff(f(x), x)
sage: D[0](f)(x, y)
diff(f(x, y), x)
sage: D[0, 1](f)(x, y)
diff(f(x, y), x, y)
sage: D[0, 1](f)(x, x^2)
D[0, 1](f)(x, x^2)
```

```
>>> from sage.all import *
>>> x, y = var('x,y'); f = function('f')
>>> D[Integer(0)](f)(x)
diff(f(x), x)
>>> D[Integer(0)](f)(x, y)
diff(f(x, y), x)
>>> D[Integer(0), Integer(1)](f)(x, y)
diff(f(x, y), x, y)
>>> D[Integer(0), Integer(1)](f)(x, x**Integer(2))
D[0, 1](f)(x, x^2)
```

class DerivativeOperatorWithParameters(*parameter_set*)

Bases: object

class sage.symbolic.operators.**FDerivativeOperator**(*function*, *parameter_set*)

Bases: object

Function derivative operators.

A function derivative operator represents a partial derivative of a function with respect to some variables.

The underlying data are the function, and the parameter set, a list recording the indices of the variables with respect to which the partial derivative is taken.

change_function(*new*)

Return a new function derivative operator with the same parameter set but for a new function.

EXAMPLES:

```
sage: from sage.symbolic.operators import FDerivativeOperator
sage: f = function('foo')
sage: b = function('bar')
sage: op = FDerivativeOperator(f, [0, 1])
sage: op.change_function(bar)
D[0, 1](bar)
```

```
>>> from sage.all import *
>>> from sage.symbolic.operators import FDerivativeOperator
>>> f = function('foo')
>>> b = function('bar')
>>> op = FDerivativeOperator(f, [Integer(0), Integer(1)])
>>> op.change_function(bar)
D[0, 1](bar)
```

function()

Return the function associated to this function derivative operator.

EXAMPLES:

```
sage: from sage.symbolic.operators import FDerivativeOperator
sage: f = function('foo')
sage: op = FDerivativeOperator(f, [0, 1])
sage: op.function()
foo
```

```
>>> from sage.all import *
>>> from sage.symbolic.operators import FDerivativeOperator
>>> f = function('foo')
>>> op = FDerivativeOperator(f, [Integer(0), Integer(1)])
>>> op.function()
foo
```

parameter_set()

Return the parameter set of this function derivative operator.

This is the list of indices of variables with respect to which the derivative is taken.

EXAMPLES:

```
sage: from sage.symbolic.operators import FDerivativeOperator
sage: f = function('foo')
sage: op = FDerivativeOperator(f, [0, 1])
sage: op.parameter_set()
[0, 1]
```

```
>>> from sage.all import *
>>> from sage.symbolic.operators import FDerivativeOperator
>>> f = function('foo')
>>> op = FDerivativeOperator(f, [Integer(0), Integer(1)])
>>> op.parameter_set()
[0, 1]
```

sage.symbolic.operators.**add_vararg**(*first*, **rest*)

Return the sum of all the arguments.

INPUT:

- `first`, `*rest` – arguments to add

OUTPUT: sum of the arguments

EXAMPLES:

```
sage: from sage.symbolic.operators import add_vararg
sage: add_vararg(1, 2, 3, 4, 5, 6, 7)
28
sage: x = SR.var('x')
sage: s = 1 + x + x^2   # symbolic sum
sage: bool(s.operator()(*s.operands()) == s)
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.operators import add_vararg
>>> add_vararg(Integer(1), Integer(2), Integer(3), Integer(4), Integer(5),␣
↪Integer(6), Integer(7))
28
>>> x = SR.var('x')
>>> s = Integer(1) + x + x**Integer(2)   # symbolic sum
>>> bool(s.operator()(*s.operands()) == s)
True
```

sage.symbolic.operators.**mul_vararg**(*first*, **rest*)

Return the product of all the arguments.

INPUT:

- `first`, `*rest` – arguments to multiply

OUTPUT: product of the arguments

EXAMPLES:

```
sage: from sage.symbolic.operators import mul_vararg
sage: mul_vararg(9, 8, 7, 6, 5, 4)
60480
```

(continues on next page)

(continued from previous page)

```
sage: x = SR.var('x')
sage: p = x * cos(x) * sin(x)   # symbolic product
sage: bool(p.operator()(*p.operands()) == p)
True
```

```
>>> from sage.all import *
>>> from sage.symbolic.operators import mul_vararg
>>> mul_vararg(Integer(9), Integer(8), Integer(7), Integer(6), Integer(5),␣
↪Integer(4))
60480
>>> x = SR.var('x')
>>> p = x * cos(x) * sin(x)   # symbolic product
>>> bool(p.operator()(*p.operands()) == p)
True
```

2.33 Benchmarks

Tests that will take a long time if something is wrong, but be very quick otherwise. See https://wiki.sagemath.org/symbench. The parameters chosen below are such that with pynac most of these take well less than a second, but would not even be feasible using Sage's Maxima-based symbolics.

Problem R1

Important note. Below we do s.expand().real() because s.real() takes forever (TODO?).

```
sage: f(z) = sqrt(1/3)*z^2 + i/3
sage: s = f(f(f(f(f(f(f(f(f(f(i/2))))))))))
sage: s.expand().real()
-15323490199844318074242473679071410934833494247466385771803570370858961112774390851791
↪816665679690269559944266275450221158422610550864829860001809051017043021688197776127
↪950364280100817827198253104272072717813588170292459504467263431341723932730457665263
↪332109587572477188748659485208352600164821731771879468537939194614366329290793454584
↪293141198226478876661981255999951540881379628744878434385498068679878257595225816399
↪223611375235323770508845148116869115805950516180796108216231522505729939434820353900
↪258269288473574537739141663854052032336322493116368032469002580200976130713750496330
↪464083589158892588313507899639861636157106594196462804321489035645414503946405543014
↪3/16095998759224694773994485937577374404341600184191042304646688040286318700912682244
↪197817113985332500162377034494593973193701004762164451231301473229400198399276285994
↪792946785996899286435702379837369663054238319473663324668784869926768232153033121399
↪850155929745377211409322439068321250497769340729275766668493319563518628285676685057
↪773881333128424887017517863405443082317192363998756921166842647773997457240285324889
↪512613663992842579081771571790990411154313355878887276292978004143353025122721401971
↪498976738820995466462367907399031469705780010920183465244647991463312258221428804592
↪028002290130820330287220777033623601598272361630412995009921776276570141031383772870
↪73792*sqrt(1/3)
```

```
>>> from sage.all import *
>>> __tmp__=var("z"); f = symbolic_expression(sqrt(Integer(1)/
↪Integer(3))*z**Integer(2) + i/Integer(3)).function(z)
>>> s = f(f(f(f(f(f(f(f(f(f(i/Integer(2)))))))))))
>>> s.expand().real()
```

(continues on next page)

(continued from previous page)

```
-15323490199844318074242473679071410934833494247466385771803570370858961112774390851 79
↪81666567969026955994426627545022115842261055086482986000180905101704302168819777612 7
↪95036428010081782719825310427207271781358817029245950446726343134172393273045766526 3
↪33210958757247718874865948520835260016482173177187946853793919461436632929079345458 4
↪29314119822647887666198125599995154088137962874487843438549806867987825759522581639 9
↪22361137523532377050884514811686911580595051618079610821623152250572993943482035390 0
↪25826928847357453773914166385405203233632249311636803246900258020097613071375049633 0
↪46408358915889258831350789963986163615710659419646280432148903564541450394640554301 4
↪3/1609599875922469477399448593757737440434160018419104230464668804028631870091268244
↪19781711398533250016237703449459397319370100476216445123130147322940019839927628599 4
↪79294678599689928643570237983736966305423831947366332466878486992676823215303312139 9
↪85015592974537721140932243906832125049776934072927576666849331956351862828567668505 7
↪77388133331284248870175178634054430823171923639987569211668426477739974572402853248 9
↪51261366399284257908177157179099041115431335587887276292978004143353025122721401971 5
↪49897673882099546646236790739903146970578001092018346524464799146331225822142880459 2
↪02800229013082033028722077703362360159827236163041299500992177627657014103138377287 0
↪73792*sqrt(1/3)
```

Problem R2:

```
sage: def hermite(n, y):
....:     if n == 1: return 2*y
....:     if n == 0: return 1
....:     return expand(2*y*hermite(n-1,y) - 2*(n-1)*hermite(n-2,y))
sage: hermite(15,var('y'))
32768*y^15 - 1720320*y^13 + 33546240*y^11 - 307507200*y^9 + 1383782400*y^7 -␣
↪2905943040*y^5 + 2421619200*y^3 - 518918400*y
```

```
>>> from sage.all import *
>>> def hermite(n, y):
...     if n == Integer(1): return Integer(2)*y
...     if n == Integer(0): return Integer(1)
...     return expand(Integer(2)*y*hermite(n-Integer(1),y) - Integer(2)*(n-
↪Integer(1))*hermite(n-Integer(2),y))
>>> hermite(Integer(15),var('y'))
32768*y^15 - 1720320*y^13 + 33546240*y^11 - 307507200*y^9 + 1383782400*y^7 -␣
↪2905943040*y^5 + 2421619200*y^3 - 518918400*y
```

Problem R3:

```
sage: f = sum(var('x,y,z')); a = [bool(f==f) for _ in range(100000)]
```

```
>>> from sage.all import *
>>> f = sum(var('x,y,z')); a = [bool(f==f) for _ in range(Integer(100000))]
```

Problem R4:

```
sage: u = [e,pi,sqrt(2)]; Tuples(u,3).cardinality()
27
```

```
>>> from sage.all import *
>>> u = [e,pi,sqrt(Integer(2))]; Tuples(u,Integer(3)).cardinality()
```

(continues on next page)

(continued from previous page)

```
27
```

Problem R5:

```
sage: def blowup(L, n):
....:     for i in [0..n]:
....:         L.append( (L[i] + L[i+1]) * L[i+2] )
sage: L = list(var('x,y,z'))
sage: blowup(L,15)
sage: len(set(L))
19
```

```
>>> from sage.all import *
>>> def blowup(L, n):
...     for i in (ellipsis_range(Integer(0),Ellipsis,n)):
...         L.append( (L[i] + L[i+Integer(1)]) * L[i+Integer(2)] )
>>> L = list(var('x,y,z'))
>>> blowup(L,Integer(15))
>>> len(set(L))
19
```

Problem R6:

```
sage: sum(((x+sin(i))/x+(x-sin(i))/x) for i in range(100)).expand()
200
```

```
>>> from sage.all import *
>>> sum(((x+sin(i))/x+(x-sin(i))/x) for i in range(Integer(100))).expand()
200
```

Problem R7:

```
sage: f = x^24+34*x^12+45*x^3+9*x^18 +34*x^10+ 32*x^21
sage: a = [f(x=random()) for _ in range(10^4)]
```

```
>>> from sage.all import *
>>> f = x**Integer(24)+Integer(34)*x**Integer(12)+Integer(45)*x**Integer(3)+Integer(9)
↪*x**Integer(18) +Integer(34)*x**Integer(10)+ Integer(32)*x**Integer(21)
>>> a = [f(x=random()) for _ in range(Integer(10)**Integer(4))]
```

Problem R10:

```
sage: v = [float(z) for z in [-pi,-pi+1/100..,pi]]
```

```
>>> from sage.all import *
>>> v = [float(z) for z in (ellipsis_range(-pi,-pi+Integer(1)/Integer(100),Ellipsis,
↪pi))]
```

Problem R11:

```
sage: a = [random() + random()*I for w in [0..100]]
sage: a.sort()
```

```
>>> from sage.all import *
>>> a = [random() + random()*I for w in (ellipsis_range(Integer(0),Ellipsis,
↪Integer(100)))]
>>> a.sort()
```

Problem W3:

```
sage: acos(cos(x))
arccos(cos(x))
```

```
>>> from sage.all import *
>>> acos(cos(x))
arccos(cos(x))
```

PROBLEM S1:

```
sage: _ = var('x,y,z')
sage: f = (x+y+z+1)^10
sage: g = expand(f*(f+1))
```

```
>>> from sage.all import *
>>> _ = var('x,y,z')
>>> f = (x+y+z+Integer(1))**Integer(10)
>>> g = expand(f*(f+Integer(1)))
```

PROBLEM S2:

```
sage: _ = var('x,y')
sage: a = expand((x^sin(x) + y^cos(y) - z^(x+y))^100)
```

```
>>> from sage.all import *
>>> _ = var('x,y')
>>> a = expand((x**sin(x) + y**cos(y) - z**(x+y))**Integer(100))
```

PROBLEM S3:

```
sage: _ = var('x,y,z')
sage: f = expand((x^y + y^z + z^x)^50)
sage: g = f.diff(x)
```

```
>>> from sage.all import *
>>> _ = var('x,y,z')
>>> f = expand((x**y + y**z + z**x)**Integer(50))
>>> g = f.diff(x)
```

PROBLEM S4:

```
sage: w = (sin(x)*cos(x)).series(x,400)
```

```
>>> from sage.all import *
>>> w = (sin(x)*cos(x)).series(x,Integer(400))
```

2.34 Randomized tests of GiNaC / PyNaC

`sage.symbolic.random_tests.`**`assert_strict_weak_order`**(*a*, *b*, *c*, *cmp_func*)

Check that `cmp_func` is a strict weak order on the elements a,b,c.

A strict weak order is a binary relation < such that

- For all x, it is not the case that $x < x$ (irreflexivity).
- For all $x \neq y$, if $x < y$ then it is not the case that $y < x$ (asymmetry).
- For all x, y, and z, if $x < y$ and $y < z$ then $x < z$ (transitivity).
- For all x, y, and z, if x is incomparable with y, and y is incomparable with z, then x is incomparable with z (transitivity of incomparability).

INPUT:

- `a`, `b`, `c` – anything that can be compared by `cmp_func`
- `cmp_func` – function of two arguments that returns their comparison (i.e. either `True` or `False`)

OUTPUT:

Nothing. Raises a `ValueError` if `cmp_func` is not a strict weak order on the three given elements.

REFERENCES:

Wikipedia article Strict_weak_ordering

EXAMPLES:

The usual ordering of integers is a strict weak order:

```
sage: from sage.symbolic.random_tests import assert_strict_weak_order
sage: a, b, c = [randint(-10, 10) for i in range(3)]
sage: assert_strict_weak_order(a, b, c, lambda x, y: x < y)

sage: x = [-SR(oo), SR(0), SR(oo)]
sage: cmp_M = matrix(3, 3, 0)
sage: for i in range(3):
....:     for j in range(3):
....:         if x[i] < x[j]:
....:             cmp_M[i, j] = -1
....:         elif x[i] > x[j]:
....:             cmp_M[i, j] = 1
sage: cmp_M
[ 0 -1 -1]
[ 1  0 -1]
[ 1  1  0]
```

```
>>> from sage.all import *
>>> from sage.symbolic.random_tests import assert_strict_weak_order
>>> a, b, c = [randint(-Integer(10), Integer(10)) for i in range(Integer(3))]
>>> assert_strict_weak_order(a, b, c, lambda x, y: x < y)

>>> x = [-SR(oo), SR(Integer(0)), SR(oo)]
>>> cmp_M = matrix(Integer(3), Integer(3), Integer(0))
>>> for i in range(Integer(3)):
```

(continues on next page)

(continued from previous page)

```
...     for j in range(Integer(3)):
...         if x[i] < x[j]:
...             cmp_M[i, j] = -Integer(1)
...         elif x[i] > x[j]:
...             cmp_M[i, j] = Integer(1)
>>> cmp_M
[ 0 -1 -1]
[ 1  0 -1]
[ 1  1  0]
```

sage.symbolic.random_tests.**choose_from_prob_list**(*lst*)

INPUT:

- lst – list of tuples, where the first element of each tuple is a nonnegative float (a probability), and the probabilities sum to one

OUTPUT:

A tuple randomly selected from the list according to the given probabilities.

EXAMPLES:

```
sage: from sage.symbolic.random_tests import *
sage: v = [(0.1, False), (0.9, True)]
sage: choose_from_prob_list(v)  # random
(0.900000000000000, True)
sage: true_count = 0
sage: total_count = 0
sage: def more_samples():
....:     global true_count, total_count
....:     for _ in range(10000):
....:         total_count += 1.0
....:         if choose_from_prob_list(v)[1]:
....:             true_count += 1.0
sage: more_samples()
sage: while abs(true_count/total_count - 0.9) > 0.01:
....:     more_samples()
```

```
>>> from sage.all import *
>>> from sage.symbolic.random_tests import *
>>> v = [(RealNumber('0.1'), False), (RealNumber('0.9'), True)]
>>> choose_from_prob_list(v)  # random
(0.900000000000000, True)
>>> true_count = Integer(0)
>>> total_count = Integer(0)
>>> def more_samples():
...     global true_count, total_count
...     for _ in range(Integer(10000)):
...         total_count += RealNumber('1.0')
...         if choose_from_prob_list(v)[Integer(1)]:
...             true_count += RealNumber('1.0')
>>> more_samples()
>>> while abs(true_count/total_count - RealNumber('0.9')) > RealNumber('0.01'):
...     more_samples()
```

sage.symbolic.random_tests.**normalize_prob_list**(*pl*, *extra=()*)

INPUT:

- pl – list of tuples, where the first element of each tuple is a floating-point number (representing a relative probability). The second element of each tuple may be a list or any other kind of object.
- extra – tuple which is to be appended to every tuple in pl

This function takes such a list of tuples (a "probability list") and normalizes the probabilities so that they sum to one. If any of the values are lists, then those lists are first normalized; then the probabilities in the list are multiplied by the main probability and the sublist is merged with the main list.

For example, suppose we want to select between group A and group B with 50% probability each. Then within group A, we select A1 or A2 with 50% probability each (so the overall probability of selecting A1 is 25%); and within group B, we select B1, B2, or B3 with probabilities in a 1:2:2 ratio.

EXAMPLES:

```
sage: from sage.symbolic.random_tests import *
sage: A = [(0.5, 'A1'), (0.5, 'A2')]
sage: B = [(1, 'B1'), (2, 'B2'), (2, 'B3')]
sage: top = [(50, A, 'Group A'), (50, B, 'Group B')]
sage: normalize_prob_list(top)
[(0.250000000000000, 'A1', 'Group A'), (0.250000000000000, 'A2', 'Group A'), (0.1,
↪ 'B1', 'Group B'), (0.2, 'B2', 'Group B'), (0.2, 'B3', 'Group B')]
```

```
>>> from sage.all import *
>>> from sage.symbolic.random_tests import *
>>> A = [(RealNumber('0.5'), 'A1'), (RealNumber('0.5'), 'A2')]
>>> B = [(Integer(1), 'B1'), (Integer(2), 'B2'), (Integer(2), 'B3')]
>>> top = [(Integer(50), A, 'Group A'), (Integer(50), B, 'Group B')]
>>> normalize_prob_list(top)
[(0.250000000000000, 'A1', 'Group A'), (0.250000000000000, 'A2', 'Group A'), (0.1,
↪ 'B1', 'Group B'), (0.2, 'B2', 'Group B'), (0.2, 'B3', 'Group B')]
```

sage.symbolic.random_tests.**random_expr**(*size, nvars=1, ncoeffs=None, var_frac=0.5, internal=[(0.6, [(0.3, <built-in function add>), (0.1, <built-in function sub>), (0.3, <built-in function mul>), (0.2, <built-in function truediv>), (0.1, <built-in function pow>)], 2), (0.2, [(0.8, <built-in function neg>), (0.2, <built-in function inv>)], 1), (0.2, [(1.0, Ei, 1), (1.0, Order, 1), (1.0, _swap_harmonic, 2), (1.0, abs, 1), (1.0, airy_ai, 1), (1.0, airy_ai_prime, 1), (1.0, airy_bi, 1), (1.0, airy_bi_prime, 1), (1.0, arccos, 1), (1.0, arccosh, 1), (1.0, arccot, 1), (1.0, arccoth, 1), (1.0, arccsc, 1), (1.0, arccsch, 1), (1.0, arcsec, 1), (1.0, arcsech, 1), (1.0, arcsin, 1), (1.0, arcsinh, 1), (1.0, arctan, 1), (1.0, arctan2, 2), (1.0, arctanh, 1), (1.0, arg, 1), (1.0, bessel_I, 2), (1.0, bessel_J, 2), (1.0, bessel_K, 2), (1.0, bessel_Y, 2), (1.0, beta, 2), (1.0, binomial, 2), (1.0, ceil, 1), (1.0, chebyshev_T, 2), (1.0, chebyshev_U, 2), (1.0, complex_root_of, 2), (1.0, conjugate, 1), (1.0, cos, 1), (1.0, cos_integral, 1), (1.0, cosh, 1), (1.0, cosh_integral, 1), (1.0, cot, 1), (1.0, coth, 1), (1.0, csc, 1), (1.0, csch, 1), (1.0, dickman_rho, 1), (1.0, dilog, 1), (1.0, dirac_delta, 1), (1.0, elliptic_e, 2), (1.0, elliptic_ec, 1), (1.0, elliptic_eu, 2), (1.0, elliptic_f, 2), (1.0, elliptic_kc, 1), (1.0, elliptic_pi, 3), (1.0, erf, 1), (1.0, erfc, 1), (1.0, erfi, 1), (1.0, erfinv, 1), (1.0, exp, 1), (1.0, exp_integral_e, 2), (1.0, exp_integral_e1, 1), (1.0, exp_polar, 1), (1.0, factorial, 1), (1.0, floor, 1), (1.0, frac, 1), (1.0, fresnel_cos, 1), (1.0, fresnel_sin, 1), (1.0, gamma_inc_lower, 2), (1.0, gegenbauer, 3), (1.0, gen_laguerre, 3), (1.0, gen_legendre_P, 3), (1.0, gen_legendre_Q, 3), (1.0, hahn, 5), (1.0, hankel1, 2), (1.0, hankel2, 2), (1.0, harmonic_number, 1), (1.0, heaviside, 1), (1.0, hermite, 2), (1.0, hurwitz_zeta, 2), (1.0, hypergeometric_M, 3), (1.0, hypergeometric_U, 3), (1.0, imag_part, 1), (1.0, integrate, 4), (1.0, inverse_jacobi_cd, 2), (1.0, inverse_jacobi_cn, 2), (1.0, inverse_jacobi_cs, 2), (1.0, inverse_jacobi_dc, 2), (1.0, inverse_jacobi_dn, 2), (1.0, inverse_jacobi_ds, 2), (1.0, inverse_jacobi_nc, 2), (1.0, inverse_jacobi_nd, 2), (1.0, inverse_jacobi_ns, 2), (1.0, inverse_jacobi_sc, 2), (1.0, inverse_jacobi_sd, 2), (1.0, inverse_jacobi_sn, 2), (1.0, jacobi_P, 4), (1.0, jacobi_am, 2), (1.0, jacobi_cd, 2), (1.0, jacobi_cn, 2), (1.0, jacobi_cs, 2), (1.0, jacobi_dc, 2), (1.0, jacobi_dn, 2), (1.0, jacobi_ds, 2), (1.0, jacobi_nc, 2), (1.0, jacobi_nd, 2), (1.0, jacobi_ns, 2), (1.0, jacobi_sc, 2), (1.0, jacobi_sd, 2), (1.0, jacobi_sn, 2), (1.0, krawtchouk, 4), (1.0, kronecker_delta, 2), (1.0, laguerre, 2), (1.0, lambert_w, 2), (1.0, legendre_P, 2), (1.0, legendre_Q, 2), (1.0, log, 2), (1.0, log_gamma, 1), (1.0, log_integral, 1), (1.0, log_integral_offset, 1), (1.0, meixner, 4), (1.0, polylog, 2), (1.0, prime_pi, 1), (1.0, product, 4), (1.0, real_nth_root, 2), (1.0, real_part, 1), (1.0, sec, 1), (1.0, sech, 1), (1.0, sgn, 1), (1.0, sin, 1), (1.0, sin_integral, 1), (1.0, sinh, 1), (1.0, sinh_integral, 1), (1.0, spherical_bessel_J, 2), (1.0, spherical_bessel_Y, 2), (1.0, spherical_hankel1, 2), (1.0, spherical_hankel2, 2), (1.0, spherical_harmonic, 4), (1.0, stieltjes, 1), (1.0, struve_H, 2), (1.0, struve_L, 2), (1.0, sum, 4), (1.0, tan, 1), (1.0, tanh, 1), (1.0, unit_step, 1), (1.0, zeta, 1), (1.0, zetaderiv, 2)])], nullary=[(1.0, pi), (1.0, e), (0.05, golden_ratio), (0.05, log2), (0.05, euler_gamma), (0.05, catalan), (0.05, khinchin), (0.05, twinprime), (0.05, mertens)], nullary_frac=0.2, coeff_generator=<bound method RationalField.random_element of Rational Field>, verbose=False*)

Produce a random symbolic expression of the given size. By default, the expression involves (at most) one variable, an arbitrary number of coefficients, and all of the symbolic functions and constants (from the probability lists full_internal and full_nullary). It is possible to adjust the ratio of leaves between symbolic constants, variables, and coefficients (var_frac gives the fraction of variables, and nullary_frac the fraction of symbolic constants; the remaining leaves are coefficients).

The actual mix of symbolic constants and internal nodes can be modified by specifying different probability lists.

To use a different type for coefficients, you can specify coeff_generator, which should be a function that will return a random coefficient every time it is called.

This function will often raise an error because it tries to create an erroneous expression (such as a division by zero).

EXAMPLES:

```
sage: from sage.symbolic.random_tests import *
sage: some_functions = [arcsinh, arctan, arctan2, arctanh,
....: arg, beta, binomial, ceil, conjugate, cos, cosh, cot, coth,
....: elliptic_pi, erf, exp, factorial, floor, heaviside, imag_part,
....: sech, sgn, sin, sinh, tan, tanh, unit_step, zeta, zetaderiv]
sage: my_internal = [(0.6, full_binary, 2), (0.2, full_unary, 1),
....: (0.2, [(1.0,f,f.number_of_arguments()) for f in some_functions])]
sage: set_random_seed(1)
sage: random_expr(50, nvars=3, internal=my_internal,  # not tested  # known bug
....:             coeff_generator=CDF.random_element)
(v1^(0.9713408427702117 + 0.195868299334218*I)/cot(-pi + v1^2 + v3) +
↪tan(arctan(v2 + arctan2(-0.35859061674557324 + 0.9407509502498164*I, v3) - 0.
↪8419115504372718 + 0.30375717982404615*I) + arctan2((0.2275357305882964 - 0.
↪8258002386106038*I)/factorial(v2), -v3 - 0.7604559947718565 - 0.
↪5543672548552057*I) + ceil(1/arctan2(v1, v1))))/v2
sage: random_expr(5, verbose=True)  # not tested  # known bug
About to apply <built-in function inv> to [31]
About to apply sgn to [v1]
About to apply <built-in function add> to [1/31, sgn(v1)]
sgn(v1) + 1/31
```

```
>>> from sage.all import *
>>> from sage.symbolic.random_tests import *
>>> some_functions = [arcsinh, arctan, arctan2, arctanh,
... arg, beta, binomial, ceil, conjugate, cos, cosh, cot, coth,
... elliptic_pi, erf, exp, factorial, floor, heaviside, imag_part,
... sech, sgn, sin, sinh, tan, tanh, unit_step, zeta, zetaderiv]
>>> my_internal = [(RealNumber('0.6'), full_binary, Integer(2)), (RealNumber('0.2
↪'), full_unary, Integer(1)),
... (RealNumber('0.2'), [(RealNumber('1.0'),f,f.number_of_arguments()) for f in
↪some_functions])]
>>> set_random_seed(Integer(1))
>>> random_expr(Integer(50), nvars=Integer(3), internal=my_internal,  # not
↪tested  # known bug
...             coeff_generator=CDF.random_element)
(v1^(0.9713408427702117 + 0.195868299334218*I)/cot(-pi + v1^2 + v3) +
↪tan(arctan(v2 + arctan2(-0.35859061674557324 + 0.9407509502498164*I, v3) - 0.
↪8419115504372718 + 0.30375717982404615*I) + arctan2((0.2275357305882964 - 0.
↪8258002386106038*I)/factorial(v2), -v3 - 0.7604559947718565 - 0.
```

(continues on next page)

(continued from previous page)

```
↪5543672548552057*I) + ceil(1/arctan2(v1, v1))))/v2
>>> random_expr(Integer(5), verbose=True)  # not tested  # known bug
About to apply <built-in function inv> to [31]
About to apply sgn to [v1]
About to apply <built-in function add> to [1/31, sgn(v1)]
sgn(v1) + 1/31
```

`sage.symbolic.random_tests.`**`random_expr_helper`**(*n_nodes*, *internal*, *leaves*, *verbose*)

Produce a random symbolic expression of size *n_nodes* (or slightly larger). Internal nodes are selected from the *internal* probability list; leaves are selected from *leaves*. If *verbose* is True, then a message is printed before creating an internal node.

EXAMPLES:

```
sage: from sage.symbolic.random_tests import *
sage: a = random_expr_helper(9, [(0.5, operator.add, 2),
....:     (0.5, operator.neg, 1)], [(0.5, 1), (0.5, x)], True)
About to apply <built-in function ...
```

```
>>> from sage.all import *
>>> from sage.symbolic.random_tests import *
>>> a = random_expr_helper(Integer(9), [(RealNumber('0.5'), operator.add,␣
↪Integer(2)),
...     (RealNumber('0.5'), operator.neg, Integer(1))], [(RealNumber('0.5'),␣
↪Integer(1)), (RealNumber('0.5'), x)], True)
About to apply <built-in function ...
```

In small cases we will see all cases quickly:

```
sage: def next_expr():
....:     return random_expr_helper(
....:         6, [(0.5, operator.add, 2), (0.5, operator.neg, 1)],
....:         [(0.5, 1), (0.5, x)], False)
sage: all_exprs = set()
sage: for a in range(-4, 5):
....:     for b in range(-4+abs(a), 5-abs(a)):
....:         if a % 2 and abs(a) + abs(b) == 4 and sign(a) != sign(b):
....:             continue
....:         all_exprs.add(a*x + b)
sage: our_exprs = set()
sage: while our_exprs != all_exprs:
....:    our_exprs.add(next_expr())
```

```
>>> from sage.all import *
>>> def next_expr():
...     return random_expr_helper(
...         Integer(6), [(RealNumber('0.5'), operator.add, Integer(2)),␣
↪(RealNumber('0.5'), operator.neg, Integer(1))],
...         [(RealNumber('0.5'), Integer(1)), (RealNumber('0.5'), x)], False)
>>> all_exprs = set()
>>> for a in range(-Integer(4), Integer(5)):
...     for b in range(-Integer(4)+abs(a), Integer(5)-abs(a)):
```

(continues on next page)

(continued from previous page)

```
...         if a % Integer(2) and abs(a) + abs(b) == Integer(4) and sign(a) !=␣
↪sign(b):
...             continue
...         all_exprs.add(a*x + b)
>>> our_exprs = set()
>>> while our_exprs != all_exprs:
...     our_exprs.add(next_expr())
```

sage.symbolic.random_tests.**random_integer_vector**(*n*, *length*)

Give a random list of length *length*, consisting of nonnegative integers that sum to *n*.

This is an approximation to IntegerVectors(n, length).random_element(). That gives values uniformly at random, but might be slow; this routine is not uniform, but should always be fast.

(This routine is uniform if `length` is 1 or 2; for longer vectors, we prefer approximately balanced vectors, where all the values are around $n/length$.)

EXAMPLES:

```
sage: from sage.symbolic.random_tests import *
sage: a = random_integer_vector(100, 2); a   # random
[11, 89]
sage: len(a)
2
sage: sum(a)
100

sage: b = random_integer_vector(10000, 20)
sage: len(b)
20
sage: sum(b)
10000
```

```
>>> from sage.all import *
>>> from sage.symbolic.random_tests import *
>>> a = random_integer_vector(Integer(100), Integer(2)); a   # random
[11, 89]
>>> len(a)
2
>>> sum(a)
100

>>> b = random_integer_vector(Integer(10000), Integer(20))
>>> len(b)
20
>>> sum(b)
10000
```

The routine is uniform if `length` is 2:

```
sage: true_count = 0
sage: total_count = 0
sage: def more_samples():
```

(continues on next page)

(continued from previous page)

```
....:     global true_count, total_count
....:     for _ in range(1000):
....:         total_count += 1.0
....:         if a == random_integer_vector(100, 2):
....:             true_count += 1.0
sage: more_samples()
sage: while abs(true_count/total_count - 0.01) > 0.01:
....:     more_samples()
```

```
>>> from sage.all import *
>>> true_count = Integer(0)
>>> total_count = Integer(0)
>>> def more_samples():
...     global true_count, total_count
...     for _ in range(Integer(1000)):
...         total_count += RealNumber('1.0')
...         if a == random_integer_vector(Integer(100), Integer(2)):
...             true_count += RealNumber('1.0')
>>> more_samples()
>>> while abs(true_count/total_count - RealNumber('0.01')) > RealNumber('0.01'):
...     more_samples()
```

sage.symbolic.random_tests.**test_symbolic_expression_order**(*repetitions=100*)

Test whether the comparison of random symbolic expressions satisfies the strict weak order axioms.

This is important because the C++ extension class uses `std::sort()` which requires a strict weak order. See also Issue #9880.

EXAMPLES:

```
sage: from sage.symbolic.random_tests import test_symbolic_expression_order
sage: test_symbolic_expression_order(200)
sage: test_symbolic_expression_order(10000)  # long time
```

```
>>> from sage.all import *
>>> from sage.symbolic.random_tests import test_symbolic_expression_order
>>> test_symbolic_expression_order(Integer(200))
>>> test_symbolic_expression_order(Integer(10000))  # long time
```

CHAPTER

THREE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

I

J

L

M

N

R

S

T

U

V

W

X

Z