Sage Reference Manual: Symbolic Calculus

Release 6.6

The Sage Development Team

April 18, 2015

CONTENTS

1 **Symbolic Expressions** 1

2 **Callable Symbolic Expressions** 125

3 **Assumptions** 131

4 **Symbolic Equations and Inequalities** 137
- 4.1 The operator, left hand side, and right hand side . . . 137
- 4.2 Arithmetic . . . 138
- 4.3 Substitution . . . 138
- 4.4 Solving . . . 139
- 4.5 Assumptions . . . 140
- 4.6 Miscellaneous . . . 140
- 4.7 More Examples . . . 141

5 **Symbolic Computation** 153

6 **Units of measurement** 181

7 **The symbolic ring** 189

8 **Classes for symbolic functions** 195

9 **Factory for symbolic functions** 199

10 **Functional notation support for common calculus methods** 205

11 **Symbolic Integration** 215

12 **A Sample Session using SymPy** 231

13 **Calculus Tests and Examples** 235

14 **Conversion of symbolic expressions to other types** 239

15 **Complexity Measures** 257

16 **Further examples from Wester's paper** 259

17 **Solving ordinary differential equations** 269

18 **Discrete Wavelet Transform** 287

19 Discrete Fourier Transforms 291

20 Fast Fourier Transforms Using GSL 299

21 Solving ODE numerically by GSL 305

22 Numerical Integration 309

23 Riemann Mapping 313

24 Real Interpolation using GSL 325

25 Complex Interpolation 329

26 Indices and Tables 333

Bibliography 335

CHAPTER ONE

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
```

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
```

The can transform a true relational 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
```

We can do arithmetic with relationals:

```
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)
```

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

```
sage: (x^3 + x <= x - 17)  + (-x <= x - 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
```

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)
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
```

TESTS:

Test Jacobian on Pynac expressions. (trac ticket #5546)

```
sage: var('x,y')
(x, y)
sage: f = x + y
sage: jacobian(f, [x,y])
[1 1]
```

Test if matrices work (trac ticket #5546)

```
sage: var('x,y,z')
(x, y, z)
sage: M = matrix(2,2,[x,y,z,x])
sage: v = vector([x,y])
sage: M * v
(x^2 + y^2, x*y + x*z)
sage: v*M
(x^2 + y*z, 2*x*y)
```

Test if comparison bugs from trac ticket #6256 are fixed:

```
sage: t = exp(sqrt(x)); u = 1/t
sage: t*u
1
sage: t + u
e^(-sqrt(x)) + e^sqrt(x)
sage: t
e^sqrt(x)
```

Test if trac ticket #9947 is fixed:

```
sage: real_part(1+2*(sqrt(2)+1)*(sqrt(2)-1))
3
sage: a=(sqrt(4*(sqrt(3) - 5)*(sqrt(3) + 5) + 48) + 4*sqrt(3))/ (sqrt(3) + 5)
sage: a.real_part()
4*sqrt(3)/(sqrt(3) + 5)
sage: a.imag_part()
sqrt(abs(4*(sqrt(3) + 5)*(sqrt(3) - 5) + 48))/(sqrt(3) + 5)
```

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

Bases: `sage.structure.element.CommutativeRingElement`

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.

TESTS:

```
sage: sage.symbolic.expression.Expression(SR)
0
sage: sage.symbolic.expression.Expression(SR, 5)
5
```

We test subclassing `Expression`:

```
sage: from sage.symbolic.expression import Expression
sage: class exp_sub(Expression): pass
sage: f = function('f')
sage: t = f(x)
sage: u = exp_sub(SR, t)
sage: u.operator()
f
```

N(*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation this symbolic expression as either a real or complex number with at least the requested number of bits or digits of precision.

EXAMPLES:

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

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

TESTS:

We test the evaluation of different infinities available in Pynac:

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

Some expressions cannot be evaluated numerically:

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

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

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

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
Order(n^3)
sage: 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)
```

abs (*hold=False*)

Return the absolute value of this expression.

EXAMPLES:

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

```
sage: (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)
```

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

From trac ticket #7557:

```
sage: var('y', domain='real')
y
sage: abs(exp(1.1*y*I)).simplify()
1
sage: var('y', domain='complex') # reset the domain for other tests
y
```

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
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = x.add(x, hold=True); a.simplify()
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)
```

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)
Graphics object consisting of 1 graphics primitive
```

To prevent automatic evaluation use the `hold` argument:

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

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

```
sage: SR(oo).arccos()
Traceback (most recent call last):
...
RuntimeError: arccos_eval(): arccos(infinity) encountered
sage: SR(-oo).arccos()
Traceback (most recent call last):
...
RuntimeError: arccos_eval(): arccos(infinity) encountered
sage: SR(unsigned_infinity).arccos()
Infinity
```

arccosh(*hold=False*)

Return the inverse hyperbolic cosine of self.

EXAMPLES:

```
sage: x.arccosh()
arccosh(x)
sage: SR(0).arccosh()
1/2*I*pi
sage: SR(1/2).arccosh()
arccosh(1/2)
sage: SR(CDF(1/2)).arccosh()   # rel tol 1e-15
1.0471975511965976*I
sage: maxima('acosh(0.5)')
1.04719755119659...*%i
```

To prevent automatic evaluation use the `hold` argument:

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

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

```
sage: SR(oo).arccosh()
+Infinity
sage: SR(-oo).arccosh()
+Infinity
sage: SR(unsigned_infinity).arccosh()
+Infinity
```

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()
0.523598775598299
sage: SR(0.999).arcsin()
1.52607123962616
sage: SR(1/3).arcsin()
arcsin(1/3)
sage: SR(-1/3).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)
```

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

```
sage: SR(oo).arcsin()
Traceback (most recent call last):
...
RuntimeError: arcsin_eval(): arcsin(infinity) encountered
sage: SR(-oo).arcsin()
Traceback (most recent call last):
...
RuntimeError: arcsin_eval(): arcsin(infinity) encountered
sage: SR(unsigned_infinity).arcsin()
Infinity
```

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...
```

Sage automatically applies certain identities:

```
sage: SR(3/2).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)
```

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

```
sage: SR(oo).arcsinh()
+Infinity
sage: SR(-oo).arcsinh()
-Infinity
sage: SR(unsigned_infinity).arcsinh()
Infinity
```

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)
Graphics object consisting of 1 graphics primitive
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(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
```

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

```
sage: SR(oo).arctan()
1/2*pi
sage: SR(-oo).arctan()
-1/2*pi
sage: SR(unsigned_infinity).arctan()
Traceback (most recent call last):
...
RuntimeError: arctan_eval(): arctan(unsigned_infinity) encountered
```

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)
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
```

To prevent automatic evaluation use the `hold` argument:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

We compare a bunch of different evaluation points between Sage and Maxima:

```
sage: float(SR(0.7).arctan2(0.6))
0.8621700546672264
sage: maxima('atan2(0.7,0.6)')
0.8621700546672264
sage: float(SR(0.7).arctan2(-0.6))
2.279422598922567
sage: maxima('atan2(0.7,-0.6)')
2.279422598922567
sage: float(SR(-0.7).arctan2(0.6))
-0.8621700546672264
sage: maxima('atan2(-0.7,0.6)')
-0.8621700546672264
sage: float(SR(-0.7).arctan2(-0.6))
-2.279422598922567
sage: maxima('atan2(-0.7,-0.6)')
-2.279422598922567
sage: float(SR(0).arctan2(-0.6))
3.141592653589793
sage: maxima('atan2(0,-0.6)')
3.141592653589793
sage: float(SR(0).arctan2(0.6))
0.0
sage: maxima('atan2(0,0.6)')
0.0
sage: SR(0).arctan2(0) # see trac ticket #11423
Traceback (most recent call last):
...
RuntimeError: arctan2_eval(): arctan2(0,0) encountered
sage: SR(I).arctan2(1)
arctan2(I, 1)
sage: SR(CDF(0,1)).arctan2(1)
arctan2(1.0*I, 1)
sage: SR(1).arctan2(CDF(0,1))
arctan2(1, 1.0*I)

sage: arctan2(0,oo)
0
sage: SR(oo).arctan2(oo)
1/4*pi
sage: SR(oo).arctan2(0)
```

```
1/2*pi
sage: SR(-oo).arctan2(0)
-1/2*pi
sage: SR(-oo).arctan2(-2)
pi
sage: SR(unsigned_infinity).arctan2(2)
Traceback (most recent call last):
...
RuntimeError: arctan2_eval(): arctan2(x, unsigned_infinity) encountered
sage: SR(2).arctan2(oo)
1/2*pi
sage: SR(2).arctan2(-oo)
-1/2*pi
sage: SR(2).arctan2(SR(unsigned_infinity))
Traceback (most recent call last):
...
RuntimeError: arctan2_eval(): arctan2(unsigned_infinity, x) encountered
```

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()
arctanh(1/2)
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
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(-1/2).arctanh()
-arctanh(1/2)
sage: SR(-1/2).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)
-arctanh(1/2)
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = SR(-1/2).arctanh(hold=True); a.simplify()
-arctanh(1/2)
```

TESTS:

```
sage: SR(1).arctanh()
+Infinity
sage: SR(-1).arctanh()
```

```
-Infinity

sage: SR(oo).arctanh()
-1/2*I*pi
sage: SR(-oo).arctanh()
1/2*I*pi
sage: SR(unsigned_infinity).arctanh()
Traceback (most recent call last):
...
RuntimeError: arctanh_eval(): arctanh(unsigned_infinity) encountered
```

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,)
```

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,)
```

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()
```

Bool returns True below if the inequality is *definitely* known to be True.

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

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

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

TESTS:

```
sage: v,c = var('v,c')
sage: assume(c != 0)
sage: integral((1+v^2/c^2)^3/(1-v^2/c^2)^(3/2),v)
83/8*v/sqrt(-v^2/c^2 + 1) - 17/8*v^3/(c^2*sqrt(-v^2/c^2 + 1)) - 1/4*v^5/(c^4*sqrt(-v^2/c^2 +
sage: 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)
```

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)
```

To then evaluate again, we currently must use Maxima via `simplify()`:

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

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

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

TESTS:

Check if we handle zero correctly (trac ticket #8561):

```
sage: x.binomial(0)
1
sage: SR(0).binomial(0)
1
```

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)
```

And also handles some exponential functions:

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

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)
```

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
```

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
```

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)
```

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

```
sage: f(x = -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
```

The issue reported in trac ticket #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
```

TESTS:

This tests that trac ticket #11668 has been fixed (by trac ticket #12780):

```
sage: a,b = var('a b')
sage: A = abs((a+I*b))^2
sage: A.canonicalize_radical()
abs(a + I*b)^2
sage: imag(A)
0
sage: imag(A.canonicalize_radical())
0
```

Ensure that deprecation warnings are thrown for the old "simplify" aliases:

```
sage: x.simplify_radical()
doctest...: DeprecationWarning: simplify_radical is deprecated. Please use canonicalize_radi
See http://trac.sagemath.org/11912 for details.
x
sage: x.radical_simplify()
doctest...: DeprecationWarning: radical_simplify is deprecated. Please use canonicalize_radi
See http://trac.sagemath.org/11912 for details.
x
sage: x.simplify_exp()
doctest...: DeprecationWarning: simplify_exp is deprecated. Please use canonicalize_radical
See http://trac.sagemath.org/11912 for details.
x
sage: x.exp_simplify()
doctest...: DeprecationWarning: exp_simplify is deprecated. Please use canonicalize_radical
See http://trac.sagemath.org/11912 for details.
x
```

coeff(**args*, ***kwds*)

Deprecated: Use `coefficient()` instead. See trac ticket #17438 for details.

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

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

INPUT:

- `s` - expression
- `n` - integer, 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)
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)
sqrt(x)*sin(y) + z^z
```

TESTS:

Check if trac ticket #9505 is fixed:

```
sage: var('x,y,z')
(x, y, z)
sage: f = x*y*z^2
sage: f.coefficient(x*y)
z^2
sage: f.coefficient(x*y, 2)
Traceback (most recent call last):
...
TypeError: n != 1 only allowed for s being a variable
```

Using `coeff()` is now deprecated (trac ticket #17438):

```
sage: x.coeff(x)
doctest:...: DeprecationWarning: coeff is deprecated. Please use coefficient instead.
See http://trac.sagemath.org/17438 for details.
1
```

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]]
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]
```

TESTS:

The behaviour is undefined with noninteger or negative exponents:

```
sage: p = (17/3*a)*x^(3/2) + x*y + 1/x + x^x
sage: p.coefficients(x)
[[1, -1], [x^x, 0], [y, 1], [17/3*a, 3/2]]
sage: p.coefficients(x, sparse=False)
Traceback (most recent call last):
...
ValueError: Cannot return dense coefficient list with noninteger exponents.
```

Using `coeffs()` is now deprecated (trac ticket #17438):

```
sage: x.coeffs()
doctest:...: DeprecationWarning: coeffs is deprecated. Please use coefficients instead.
See http://trac.sagemath.org/17438 for details.
[[1, 1]]
```

Series coefficients are now handled correctly (trac ticket #17399):

```
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]
```

We can find coefficients of symbolic functions, trac ticket #12255:

```
sage: g = function('g', var('t'))
sage: f = 3*g + g**2 + t
sage: f.coefficients(g)
[[t, 0], [3, 1], [1, 2]]
```

coeffs(**args, **kwds*)

Deprecated: Use `coefficients()` instead. See trac ticket #17438 for details.

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
```

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
```

Sometimes, we do have to call expand() on the expression first to achieve the desired result:

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

TESTS:

The output should be equivalent to the input:

```
sage: polynomials = QQ['x']
sage: f = SR(polynomials.random_element())
sage: g = f.collect(x)
sage: bool(f == g)
True
```

If s is not present in the given expression, the expression should not be modified. The variable z will not be present in f below since f is a random polynomial of maximum degree 10 in x and y:

```
sage: z = var('z')
sage: polynomials = QQ['x,y']
sage: f = SR(polynomials.random_element(10))
sage: g = f.collect(z)
sage: bool(str(f) == str(g))
True
```

Check if trac ticket #9046 is fixed:

```
sage: var('a b x y z')
(a, b, x, y, z)
sage: p = -a*x^3 - a*x*y^2 + 2*b*x^2*y + 2*y^3 + x^2*z + y^2*z + x^2 + y^2 + a*x
```

```
sage: p.collect(x)
-a*x^3 + (2*b*y + z + 1)*x^2 + 2*y^3 + y^2*z - (a*y^2 - a)*x + y^2
```

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()
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
```

combine()

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

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)
```

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
```

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

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

This also works in functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

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
```

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
```

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

```
sage: a = var('a')
sage: 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)
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.newto
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_squar
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_squar
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
```

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
```

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
```

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
```

To prevent automatic evaluation use the `hold` argument:

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

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

```
sage: SR(oo).cos()
Traceback (most recent call last):
...
RuntimeError: cos_eval(): cos(infinity) encountered
sage: SR(-oo).cos()
Traceback (most recent call last):
...
RuntimeError: cos_eval(): cos(infinity) encountered
sage: SR(unsigned_infinity).cos()
Traceback (most recent call last):
...
RuntimeError: cos_eval(): cos(infinity) encountered
```

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?
```

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))
```

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

```
sage: SR(oo).cosh()
+Infinity
sage: SR(-oo).cosh()
+Infinity
sage: SR(unsigned_infinity).cosh()
```

```
Traceback (most recent call last):
...
RuntimeError: cosh_eval(): cosh(unsigned_infinity) encountered
```

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

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

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

default_variable ()

Return the default variable, which is by definition the first variable in self, or x is 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
```

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)
```

degree (*s*)

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

OUTPUT:

An integer >= 0.

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)
0
```

denominator (*normalize=True*)

Return the denominator of this symbolic expression

INPUT:

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

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
```

TESTS:

```
sage: ((x+y)^2/(x-y)^3*x^3).denominator(normalize=False)
(x - y)^3
sage: ((x+y)^2*x^3).denominator(normalize=False)
1
sage: (y/x^3).denominator(normalize=False)
x^3
sage: t = y/x^3/(x+y)^(1/2); t
y/(sqrt(x + y)*x^3)
sage: t.denominator(normalize=False)
sqrt(x + y)*x^3
sage: (1/x^3).denominator(normalize=False)
x^3
sage: (x^3).denominator(normalize=False)
1
sage: (y*x^sin(x)).denominator(normalize=False)
Traceback (most recent call last):
...
TypeError: self is not a rational expression
```

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
```

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)

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)

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

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))

sage: y = var('y')
sage: f = y^(sin(x))
sage: 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

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

TESTS:

```
sage: t.derivative()
Traceback (most recent call last):
...
ValueError: No differentiation variable specified.
```

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
sage: 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)

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)

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

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))

sage: y = var('y')
sage: f = y^(sin(x))
sage: 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

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

TESTS:

```
sage: t.derivative()
Traceback (most recent call last):
...
ValueError: No differentiation variable specified.
```

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
```

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)

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)

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

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))
```

```
sage: y = var('y')
sage: f = y^(sin(x))
sage: 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

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

TESTS:

```
sage: t.derivative()
Traceback (most recent call last):
...
ValueError: No differentiation variable specified.
```

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
```

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

sage: SR(0.5).exp().log()
```

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

To prevent automatic evaluation use the `hold` argument:

```
sage: (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
```

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

Test if trac ticket #6377 is fixed:

```
sage: SR(oo).exp()
+Infinity
sage: SR(-oo).exp()
0
sage: SR(unsigned_infinity).exp()
Traceback (most recent call last):
...
RuntimeError: exp_eval(): exp^(unsigned_infinity) encountered
```

exp_simplify(**args*, ***kwds*)

Deprecated: Use `canonicalize_radical()` instead. See trac ticket #11912 for details.

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
```

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 + y)^2) + sin((x + y)^2)^2
```

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
```

TESTS:

```
sage: var('x,y')
(x, y)
sage: ((x + (2/3)*y)^3).expand()
x^3 + 2*x^2*y + 4/3*x*y^2 + 8/27*y^3
sage: expand( (x*sin(x) - cos(y)/x)^2 )
x^2*sin(x)^2 - 2*cos(y)*sin(x) + cos(y)^2/x^2
sage: f = (x-y)*(x+y); f
(x + y)*(x - y)
sage: f.expand()
x^2 - y^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') optional, 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)
```

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

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

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

```
sage: (log(x^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)
```

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(4) + log(3)

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

TESTS:

Most of these log expansions only make sense over the reals. So, we should set the Maxima `domain` variable to 'real' before we call out to Maxima. When we return, however, we should set the `domain` back to what it was, rather than assuming that it was 'complex'. See trac ticket #12780:

```
sage: from sage.calculus.calculus import maxima
sage: maxima('domain: real;')
real
sage: x.expand_log()
x
sage: maxima('domain;')
real
sage: maxima('domain: complex;')
complex
```

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
```

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 + y)^2) + sin((x + y)^2)^2
```

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
```

TESTS:

```
sage: var('x,y')
(x, y)
sage: ((x + (2/3)*y)^3).expand()
x^3 + 2*x^2*y + 4/3*x*y^2 + 8/27*y^3
sage: expand( (x*sin(x) - cos(y)/x)^2 )
x^2*sin(x)^2 - 2*cos(y)*sin(x) + cos(y)^2/x^2
sage: f = (x-y)*(x+y); f
(x + y)*(x - y)
sage: f.expand()
x^2 - y^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)
```

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
```

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

```
sage: f(8).n()
Traceback (most recent call last):
...
TypeError: cannot evaluate symbolic expression numerically
```

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` - (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` - (default: False) If True, causes half-angles to be simplified away.
- `plus` - (default: True) Controls the sum rule; expansion of sums (e.g. 'sin(x + y)') will take place only if plus is True.
- `times` - (default: True) Controls the product rule, expansion of products (e.g. sin(2*x)) 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)
```

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(
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)
```

ALIASES:

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

factor (*dontfactor*=[ ])

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)
```

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)
```

factor_list (*dontfactor*=[ ])

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)
```

Notice that the -1 factor is separated out:

```
sage: 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)]
```

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)]
```

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)]
```

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))]
```

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)
```

To prevent automatic evaluation use the `hold` argument:

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

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

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)

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)
```

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
```

TESTS:

Check that the problem with applying $full_simplify()$ to gamma functions (trac ticket #9240) has been fixed:

```
sage: gamma(1/3)
gamma(1/3)
sage: gamma(1/3).full_simplify()
gamma(1/3)
sage: gamma(4/3)
gamma(4/3)
sage: gamma(4/3).full_simplify()
1/3*gamma(1/3)
```

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]
```

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

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)
(0.5610963381910451, 0.8603335890...)
sage: f.find_local_maximum(0,5, tol=0.1, maxfun=10)
(0.561090323458081..., 0.857926501456...)
```

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

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:

- `var` - variable (default: first variable in self)
- `a,b` - endpoints of interval on which to minimize self.
- `tol` - the convergence tolerance
- `maxfun` - maximum function evaluations

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: 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))
sage: f.find_local_minimum(1, 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=4.5e-16*, *maxiter=100*, *full_output=False*)

Numerically find a root of self on the closed interval [a,b] (or [b,a]) if possible, where self is a function in the 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 >= 0. 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 >= 0.
- `full_output` - bool (default: False), if True, also return object that contains information about convergence.

EXAMPLES:

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

```
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
```

More examples:

```
sage: (sin(x) + exp(x)).find_root(-10, 10)
-0.588532743981862...
sage: sin(x).find_root(-1,1)
```

```
0.0
sage: (1/tan(x)).find_root(3,3.5)
3.1415926535...
```

An example with a square root:

```
sage: f = 1 + x + sqrt(x+2); f.find_root(-2,10)
-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)
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*
sage: show(plot(a, 0, .002), xmin=0, xmax=.002)
```

It is easy to approximate with `find_root`:

```
sage: a.find_root(0,0.002)
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
-23040.0*(-2.0*e^(1800*t) + 25.0*e^(900*t) - 32.0)*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
```

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.
```

TESTS:

Test the special case that failed for the first attempt to fix trac ticket #3980:

```
sage: t = var('t')
sage: find_root(1/t - x,0,2)
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)
sage: assumptions()
[x > 0]
```

TESTS:

Check if trac ticket #7507 is fixed:

```
sage: forget()
sage: n = var('n')
sage: foo=sin((-1)*n*pi)
sage: foo.simplify()
-sin(pi*n)
sage: assume(n, 'odd')
sage: assumptions()
[n is odd]
sage: foo.simplify()
0
sage: forget(n, 'odd')
sage: assumptions()
[]
sage: foo.simplify()
-sin(pi*n)
```

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
```

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
```

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

TESTS:

There are two square roots of

$$(x+1)^2$$

, so this should not be simplified to

$$x+1$$

, trac ticket #12737:

```
sage: f = sqrt((x + 1)^2)
sage: f.simplify_full()
sqrt(x^2 + 2*x + 1)
```

The imaginary part of an expression should not change under simplification; trac ticket #11934:

```
sage: f = sqrt(-8*(4*sqrt(2) - 7)*x^4 + 16*(3*sqrt(2) - 5)*x^3)
sage: original = f.imag_part()
sage: simplified = f.full_simplify().imag_part()
sage: original - simplified
0
```

The invalid simplification from trac ticket #12322 should not occur after trac ticket #12737:

```
sage: t = var('t')
sage: assume(t, 'complex')
sage: assumptions()
[t is complex]
sage: f = (1/2)*log(2*t) + (1/2)*log(1/t)
sage: f.simplify_full()
1/2*log(2*t) - 1/2*log(t)
sage: forget()
```

Complex logs are not contracted, trac ticket #17556:

```
sage: x,y = SR.var('x,y')
sage: assume(y, 'complex')
sage: f = log(x*y) - (log(x) + log(y))
sage: f.simplify_full()
log(x*y) - log(x) - log(y)
sage: forget()
```

The simplifications from `simplify_rectform()` are performed, trac ticket #17556:

```
sage: f = ( e^(I*x) - e^(-I*x) ) / ( I*e^(I*x) + I*e^(-I*x) )
sage: f.simplify_full()
sin(x)/cos(x)
```

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)

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)

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
```

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)
```

This notation also works:

```
sage: h(w,a) = f
sage: 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
```

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
```

```
sage: gp('gamma(1+I)')
0.4980156681183560427136911175 - 0.1549498283018106851249551305*I # 32-bit
0.49801566811835604271369111746219809195 - 0.15494982830181068512495513048388660520*I # 64-b
```

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

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

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)
```

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

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

gcd(*b*)

Return the gcd of self and b, which must be integers or polynomials over the rational numbers.

TODO: I tried the massive gcd from http://trac.sagemath.org/sage_trac/ticket/694 on Ginac dies after about 10 seconds. Singular easily does that GCD now. Since Ginac only handles poly gcd over QQ, we should change ginac itself to use Singular.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: SR(10).gcd(SR(15))
5
sage: (x^3 - 1).gcd(x-1)
x - 1
sage: (x^3 - 1).gcd(x^2+x+1)
x^2 + x + 1
sage: (x^3 - sage.symbolic.constants.pi).gcd(x-sage.symbolic.constants.pi)
Traceback (most recent call last):
...
ValueError: gcd: arguments must be polynomials over the rationals
sage: gcd(x^3 - y^3, x-y)
-x + y
sage: gcd(x^100-y^100, x^10-y^10)
-x^10 + y^10
sage: 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*
1/7*x^5 - 17/7*y + 2/21
```

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))
```

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
```

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
```

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
```

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
```

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
```

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
```

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]
```

hypergeometric_simplify(*algorithm='maxima'*)

Simplify an expression containing 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)
....:  .simplify_hypergeometric())
laguerre(-laguerre(-e^x, x), x)
sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1)
....:  .simplify_hypergeometric(algorithm='sage'))
hypergeometric((hypergeometric((e^x,), (1,), x),), (1,), x)
```

imag(*hold=False*)

Return the imaginary part of this symbolic expression.

EXAMPLES:

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

We simplify $\ln(\exp(z))$ to z. This should only be for $-\pi < \operatorname{Im}(z) <= \pi$, 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()
```

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))
```

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

```
sage: I.imag_part()
1
sage: 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(I)
1
```

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

```
sage: x = var('x')
sage: x.imag_part()
imag_part(x)
sage: SR(2+3*I).imag_part()
3
sage: SR(CC(2,3)).imag_part()
3.00000000000000
sage: SR(CDF(2,3)).imag_part()
3.0
```

imag_part(*hold=False*)

Return the imaginary part of this symbolic expression.

EXAMPLES:

```
sage: sqrt(-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()
```

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))
```

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

```
sage: I.imag_part()
1
sage: 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(I)
1
```

To then evaluate again, we currently must use Maxima via `simplify()`:

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

TESTS:

```
sage: x = var('x')
sage: x.imag_part()
imag_part(x)
sage: SR(2+3*I).imag_part()
3
sage: SR(CC(2,3)).imag_part()
3.00000000000000
sage: SR(CDF(2,3)).imag_part()
3.0
```

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
```

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
```

TESTS:

```
sage: var('x,y')  # check that the pynac registry is not polluted
(x, y)
sage: psr = copy(sage.symbolic.ring.pynac_symbol_registry)
sage: (x^6*y^5).implicit_derivative(y, x, 3)
-792/125*y/x^3 + 12/25*(15*x^4*y^5 + 28*x^3*y^5)/(x^6*y^4) - 36/125*(20*x^5*y^4 + 43*x^4*y^4
sage: psr == sage.symbolic.ring.pynac_symbol_registry
True
```

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)
```

TESTS:

We check that trac ticket #12438 is resolved:

```
sage: f(x) = x; f
x |--> x
sage: integral(f, x)
x |--> 1/2*x^2
sage: integral(f, x, 0, 1)
1/2

sage: f(x, y) = x + y
sage: f
(x, y) |--> x + y
sage: integral(f, y, 0, 1)
x |--> x + 1/2
sage: integral(f, x, 0, 1)
y |--> y + 1/2
sage: _(3)
7/2
sage: var("z")
z
sage: integral(f, z, 0, 2)
(x, y) |--> 2*x + 2*y
sage: integral(f, z)
(x, y) |--> (x + y)*z
```

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)
```

TESTS:

We check that trac ticket #12438 is resolved:

```
sage: f(x) = x; f
x |--> x
sage: integral(f, x)
x |--> 1/2*x^2
sage: integral(f, x, 0, 1)
1/2

sage: f(x, y) = x + y
sage: f
(x, y) |--> x + y
sage: integral(f, y, 0, 1)
x |--> x + 1/2
sage: integral(f, x, 0, 1)
y |--> y + 1/2
sage: _(3)
7/2
sage: var("z")
z
sage: integral(f, z, 0, 2)
(x, y) |--> 2*x + 2*y
sage: integral(f, z)
(x, y) |--> (x + y)*z
```

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)
```

is_constant()

Return True if this symbolic expression is a constant.

This function is intended to provide an interface to query the internal representation of the expression. In this sense, the word `constant` does not reflect the mathematical properties of the expression. Expressions which have no variables may return `False`.

EXAMPLES:

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

Note that the complex I is not a constant:

```
sage: I.is_constant()
False
sage: I.is_numeric()
True
```

is_infinity()

Return True if self is an infinite expression.

EXAMPLES:

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

is_integer()

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

EXAMPLES:

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

is_negative()

Return True if this expression is known to be negative.

EXAMPLES:

```
sage: SR(-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
```

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
```

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
```

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
```

TESTS:

Check if we can handle derivatives. trac ticket #6523:

```
sage: f(x) = function('f',x)
sage: f(x).diff(x).is_zero()
False
```

Check if trac ticket #11352 is fixed:

```
sage: el = -1/2*(2*x^2 - sqrt(2*x - 1)*sqrt(2*x + 1) - 1)
sage: el.is_polynomial(x)
False
```

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
```

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
```

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: (t0*x).is_real()
False
```

The following is real, but we cannot deduce that.:

```
sage: (x*x.conjugate()).is_real()
False
```

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
```

is_series()

Return True if `self` is a series.

Series are special kinds of symbolic expressions that are constructed via the `series()` method. They usually have an `Order()` term unless the series representation is exact, see is_terminating_series().

OUTPUT:

Boolean. Whether self is a series symbolic expression. Usually, this means that it was constructed by the series() method.

Returns False if only a subexpression of the symbolic expression is a series.

EXAMPLES:

```
sage: SR(5).is_series()
False
sage: var('x')
x
sage: x.is_series()
False
sage: exp(x).is_series()
False
sage: exp(x).series(x,10).is_series()
True
```

Laurent series are series, too:

```
sage: laurent_series = (cos(x)/x).series(x, 5)
sage: laurent_series
1*x^(-1) + (-1/2)*x + 1/24*x^3 + Order(x^5)
sage: laurent_series.is_series()
True
```

Something only containing a series as a subexpression is not a series:

```
sage: sum_expr = 1 + exp(x).series(x,5); sum_expr
(1 + 1*x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + Order(x^5)) + 1
sage: sum_expr.is_series()
False
```

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

sage: ((x*y)/y).is_symbol()
True
sage: (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. See also is_series() for general series.

OUTPUT:

Boolean. Whether `self` was constructed by `series()` and has no order term.

EXAMPLES:

```
sage: (x^5+x^2+1).series(x,10)
1 + 1*x^2 + 1*x^5
sage: (x^5+x^2+1).series(x,10).is_series()
True
sage: (x^5+x^2+1).series(x,10).is_terminating_series()
True
sage: SR(5).is_terminating_series()
False
sage: var('x')
x
sage: x.is_terminating_series()
False
sage: exp(x).series(x,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
```

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: u = sin(x)^2 + cos(x)^2 - 1
sage: u.is_trivial_zero()
False
sage: u.is_zero()
True
```

is_unit()

Return True if this expression is a unit of the symbolic ring.

EXAMPLES:

```
sage: SR(1).is_unit()
True
sage: SR(-1).is_unit()
```

```
True
sage: SR(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]
```

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 op
sage: pi.iterator()
Traceback (most recent call last):
...
ValueError: expressions containing only a numeric coefficient, constant or symbol have no op
sage: SR(5).iterator()
Traceback (most recent call last):
...
ValueError: expressions containing only a numeric coefficient, constant or symbol have no op
```

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)
```

lcm(*b*)

Return the lcm of self and b, which must be integers or polynomials over the rational numbers. This 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.

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)
Traceback (most recent call last):
...
ValueError: lcm: arguments must be polynomials over the rationals
sage: lcm(x^3 - y^3, x-y)
-x^3 + y^3
sage: lcm(x^100-y^100, x^10-y^10)
-x^100 + y^100
sage: lcm(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*
 1/21*(21*x^18 - 357*x^13*y + 14*x^13 + 357*x^6 + 9*x^5*y -
         6069*x*y - 153*y^2 + 238*x + 6*y)*(21*x^7 + 357*x^6 +
                 9*x^5*y - 357*x^2*y + 14*x^2 - 6069*x*y -
                 153*y^2 + 238*x + 6*y)/(3*x^5 - 51*y + 2)
```

TESTS:

Verify that x * y = gcd(x,y) * lcm(x,y):

```
sage: x, y = var('x,y')
sage: LRs = [(SR(10), SR(15)), (x^3-1, x-1), (x^3-y^3, x-y), (x^3-1, x^2+x+1), (SR(0), x-y)]
sage: all((L.gcd(R) * L.lcm(R)) == L*R for L, R in LRs)
True
```

Make sure that the convention for what to do with the 0 is being respected:

```
sage: gcd(x, SR(0)), lcm(x, SR(0))
(x, 0)
sage: gcd(SR(0), SR(0)), lcm(SR(0), SR(0))
(0, 0)
```

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
```

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
```

```
sage: 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
```

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
sage: 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
```

limit(**args*, ***kwds*)

Return a symbolic limit. See `sage.calculus.calculus.limit`

EXAMPLES:

```
sage: (sin(x)/x).limit(x=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]
```

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)
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)
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)
```

To then evaluate again, we currently must use Maxima via simplify():

```
sage: a = I.log(hold=True); a.simplify()
1/2*I*pi
```

The hold parameter also works in functional notation:

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

TESTS:

```
sage: SR(oo).log()
+Infinity
sage: SR(-oo).log()
+Infinity
sage: SR(unsigned_infinity).log()
+Infinity
```

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') optional, 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)
```

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

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

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

```
sage: (log(x^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)
```

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(4) + log(3)

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

TESTS:

Most of these log expansions only make sense over the reals. So, we should set the Maxima `domain` variable to 'real' before we call out to Maxima. When we return, however, we should set the `domain` back to what it was, rather than assuming that it was 'complex'. See trac ticket #12780:

```
sage: from sage.calculus.calculus import maxima
sage: maxima('domain: real;')
real
sage: x.expand_log()
x
sage: maxima('domain;')
real
sage: maxima('domain: complex;')
complex
```

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: set_verbose(-1); plot(lambda x: SR(x).log_gamma(), -7,8, plot_points=1000).show()
sage: math.exp(0.5)
1.6487212707001282
sage: plot(lambda x: (SR(x).exp() - SR(-x).exp())/2 - SR(x).sinh(), -1, 1)
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)
```

To evaluate again, currently we must use numerical evaluation via `n()`:

```
sage: a = SR(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
```

INPUT:

- `self` - expression to be simplified
- `algorithm` - (default: None) optional, 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')
```

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)
```

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)
```

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)

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))
```

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)
```

`x*log(9)` is contracted only if `algorithm` is `'all'`:

```
sage: (x*log(9)).simplify_log()
x*log(9)
sage: (x*log(9)).simplify_log('all')
log(9^x)
```

TESTS:

Ensure that the option `algorithm` from one call has no influence upon future calls (a Maxima flag was set, and we have to ensure that its value has been restored):

```
sage: f = log(x)+2*log(y)+1/2*log(t)
sage: f.simplify_log('one')
1/2*log(t) + log(x) + 2*log(y)

sage: f.simplify_log('ratios')
log(sqrt(t)*x*y^2)

sage: f.simplify_log()
log(x*y^2) + 1/2*log(t)
```

This shows that the issue at trac ticket #7334 is fixed. Maxima intentionally keeps the expression inside the log factored:

```
sage: log_expr = (log(sqrt(2)-1)+log(sqrt(2)+1))
sage: log_expr.simplify_log('all')
log((sqrt(2) + 1)*(sqrt(2) - 1))
sage: _.simplify_rational()
0
```

We should use the current simplification domain rather than set it to 'real' explicitly (trac ticket #12780):

```
sage: f = sqrt(x^2)
sage: f.simplify_log()
sqrt(x^2)
sage: from sage.calculus.calculus import maxima
sage: maxima('domain: real;')
real
sage: f.simplify_log()
abs(x)
sage: maxima('domain: complex;')
complex
```

AUTHORS:

- Robert Marik (11-2009)

low_degree(*s*)

Return the exponent of the lowest nonpositive power of s in self.

OUTPUT:

An integer <= `0`.

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
```

match(*pattern*)

Check if self matches the given pattern.

INPUT:

- `pattern` – a symbolic expression, possibly containing wildcards to match for

OUTPUT:

One of

`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
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: t[w0], t[w1]
(c, b)
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], t[w1], t[w2]
(a, c, b)
sage: print ((a+b)*(a+c)).match((w0+w1)*(w1+w2))
None
sage: t = (a*(x+y)+a*z+b).match(a*w0+w1)
sage: t[w0], t[w1]
(x + y, a*z + b)
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)
{$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}
```

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()
<built-in function mul>
```

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)
<type 'sage.symbolic.expression.Expression'>
```

minpoly (**args*, ***kwds*)

Return the minimal polynomial of this symbolic expression.

EXAMPLES:

```
sage: 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
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
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = x.mul(x, hold=True); a.simplify()
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)
```

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
```

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)
```

n (*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation this symbolic expression as either a real or complex number with at least the requested number of bits or digits of precision.

EXAMPLES:

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

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

TESTS:

We test the evaluation of different infinities available in Pynac:

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

Some expressions cannot be evaluated numerically:

```
sage: n(sin(x))
Traceback (most recent call last):
...
TypeError: cannot evaluate symbolic expression numerically
sage: a = var('a')
```

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

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

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

negation()

Return the negated version of self, that 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)
```

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)
```

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)
```

nops()

Return the number of arguments of this expression.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: a.number_of_operands()
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
```

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
```

normalize()

Return this expression normalized as a fraction

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))
```

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

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

sage: ( 2^(8/9) - 2^(1/9) )(x-1)
Traceback (most recent call last):
...
ValueError: the number of arguments must be less than or equal to 0
```

number_of_operands()

Return the number of arguments of this expression.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: a.number_of_operands()
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
```

numerator(*normalize=True*)

Return the numerator of this symbolic expression

INPUT:

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

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)
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
```

TESTS:

```
sage: ((x+y)^2/(x-y)^3*x^3).numerator(normalize=False)
(x + y)^2*x^3
sage: ((x+y)^2*x^3).numerator(normalize=False)
(x + y)^2*x^3
sage: (y/x^3).numerator(normalize=False)
y
sage: t = y/x^3/(x+y)^(1/2); t
y/(sqrt(x + y)*x^3)
sage: t.numerator(normalize=False)
y
sage: (1/x^3).numerator(normalize=False)
1
sage: (x^3).numerator(normalize=False)
x^3
sage: (y*x^sin(x)).numerator(normalize=False)
Traceback (most recent call last):
...
TypeError: self is not a rational expression
```

numerator_denominator(*normalize=True*)

Return the numerator and the denominator of this symbolic expression

INPUT:

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

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()`

EXAMPLE:

```
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)
```

TESTS:

```
sage: ((x+y)^2/(x-y)^3*x^3).numerator_denominator(normalize=False)
((x + y)^2*x^3, (x - y)^3)
sage: ((x+y)^2*x^3).numerator_denominator(normalize=False)
((x + y)^2*x^3, 1)
sage: (y/x^3).numerator_denominator(normalize=False)
(y, x^3)
sage: t = y/x^3/(x+y)^(1/2); t
y/(sqrt(x + y)*x^3)
sage: t.numerator_denominator(normalize=False)
(y, sqrt(x + y)*x^3)
sage: (1/x^3).numerator_denominator(normalize=False)
(1, x^3)
sage: (x^3).numerator_denominator(normalize=False)
(x^3, 1)
sage: (y*x^sin(x)).numerator_denominator(normalize=False)
Traceback (most recent call last):
...
TypeError: self is not a rational expression
```

numerical_approx(*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation this symbolic expression as either a real or complex number with at least the requested number of bits or digits of precision.

EXAMPLES:

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

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

TESTS:

We test the evaluation of different infinities available in Pynac:

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

Some expressions cannot be evaluated numerically:

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

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

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

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]
```

operator()

Return the topmost operator in this expression.

EXAMPLES:

```
sage: x,y,z = var('x,y,z')
sage: (x+y).operator()
<built-in function add>
sage: (x^y).operator()
<built-in function pow>
sage: (x^y * z).operator()
<built-in function mul>
sage: (x < y).operator()
```

```
<built-in function lt>

sage: abs(x).operator()
abs
sage: r = gamma(x).operator(); type(r)
<class 'sage.functions.other.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
D[0](f)(x)
sage: a.operator()
D[0](f)
```

TESTS:

```
sage: (x <= y).operator()
<built-in function le>
sage: (x == y).operator()
<built-in function eq>
sage: (x != y).operator()
<built-in function ne>
sage: (x > y).operator()
<built-in function gt>
sage: (x >= y).operator()
<built-in function ge>
sage: SR._force_pyobject( (x, x + 1, x + 2) ).operator()
<type 'tuple'>
```

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.

EXAMPLES:

```
sage: f = x^2/(x+1)^3
sage: f.partial_fraction()
1/(x + 1) - 2/(x + 1)^2 + 1/(x + 1)^3
sage: 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
```

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
```

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))
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))
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))
Graphics object consisting of 1 graphics primitive
```

A very thick green plot with a frame:

```
sage: sin(x).plot((x,-4*pi, 4*pi), thickness=20, rgbcolor=(0,0.7,0)).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)
Graphics3d Object
```

A more complicated family:

```
sage: G = sum([plot(sin(n*x), (x,-2*pi, 2*pi)).plot3d(z=n) for n in [0,0.1,..1]])
sage: G.show(frame_aspect_ratio=[1,1,1/2])  # long time (5s on sage.math, 2012)
```

A plot involving the floor function:

```
sage: plot(1.0 - x * floor(1/x), (x,0.00001,1.0))
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)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '*': 'Integer Ring' and '<class 'sage.functions
```

You should evaluate the function first:

```
sage: plot(2*sin(x), -4, 4)
Graphics object consisting of 1 graphics primitive
```

TESTS:

```
sage: f(x) = x*(1 - x)
sage: plot(f,0,1)
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
-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
```

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
```

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).dict()
{0: -5, 2: 6}
```

```
sage: f = x^2*e + x + pi/e
sage: f.polynomial(RDF)
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)
2.718281828459045*x^2 + x + 1.1557273497909217
sage: 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)
(-2.71828182846)*y^2 + 31.0062766803*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
```

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)
54.598150033144236*x^4 - 20.085536923187668*I*x^3 - 7.38905609893065*x^2 + 2.718281828459045
sage: f.polynomial(CC)
54.5981500331442*x^4 - 20.0855369231877*I*x^3 - 7.38905609893065*x^2 + 2.71828182845905*I*x
```

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
```

We check to make sure constants are converted appropriately:

```
sage: (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]
```

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
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = (x^2).power(2, hold=True); a.simplify()
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
```

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)
4*x + 1
sage: var('y')
```

```
y
sage: (2*x + 4*sin(y)).primitive_part(sin(y))
x + 2*sin(y)
```

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()
-17/3
sage: a.pyobject() is b
True
```

TESTS:

```
sage: SR(oo).pyobject()
+Infinity
sage: SR(-oo).pyobject()
-Infinity
sage: SR(unsigned_infinity).pyobject()
Infinity
sage: SR(I*oo).pyobject()
Traceback (most recent call last):
...
TypeError: Python infinity cannot have complex phase.
```

radical_simplify(**args*, ***kwds*)

Deprecated: Use `canonicalize_radical()` instead. See trac ticket #11912 for details.

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
```

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 + y)^2) + sin((x + y)^2)^2
```

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
```

TESTS:

```
sage: var('x,y')
(x, y)
sage: ((x + (2/3)*y)^3).expand()
x^3 + 2*x^2*y + 4/3*x*y^2 + 8/27*y^3
sage: expand( (x*sin(x) - cos(y)/x)^2 )
x^2*sin(x)^2 - 2*cos(y)*sin(x) + cos(y)^2/x^2
sage: f = (x-y)*(x+y); f
(x + y)*(x - y)
sage: f.expand()
x^2 - y^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 commmon denominator and add)
- `map` - (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))

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)
```

With `map=True` each term in a sum is simplified separately and thus the resuls are shorter for functions which are combination of rational and nonrational funtions. 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
```

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
```

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))
```

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()
2.0
sage: SR(CC(2,3)).real_part()
2.00000000000000

sage: f = log(x)
sage: 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)
```

This also works using functional notation:

```
sage: real_part(I,hold=True)
real_part(I)
sage: real_part(I)
0
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = SR(2).real_part(hold=True); a.simplify()
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))
```

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)
```

This also works using functional notation:

```
sage: real_part(I,hold=True)
real_part(I)
sage: real_part(I)
0
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = SR(2).real_part(hold=True); a.simplify()
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)
```

And $\cos(x)$:

```
sage: f = (e^(I*x) + e^(-I*x)) / 2
sage: 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
```

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)
```

TESTS:

If the expression is already in rectangular form, it should be left alone:

```
sage: a,b = var('a,b')
sage: assume((a, 'real'), (b, 'real'))
sage: f = a + b*I
sage: f.rectform()
a + I*b
sage: forget()
```

We can check with specific real numbers:

```
sage: a = RR.random_element()
sage: b = RR.random_element()
sage: f = a + b*I
sage: bool(f.rectform() == a + b*I)
True
```

If we decompose a complex number into its real and imaginary parts, they should correspond to the real and imaginary terms of the rectangular form:

```
sage: z = CC.random_element()
sage: a = z.real_part()
sage: b = z.imag_part()
sage: bool(SR(z).rectform() == a + b*I)
True
```

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
```

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)
```

ALIASES: `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) # not tested - #15846
1
```

TESTS:

```
sage: (exp(x)/sin(x)^4).residue(x == 0)
5/6
```

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
```

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
```

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
```

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` - bool (default True); require that roots be explicit rather than implicit
- `multiplicities` - bool (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)
```

A simple example:

```
sage: ((x^2-1)^2).roots()
[(-1, 2), (1, 2)]
sage: ((x^2-1)^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
```

The default variable is a, since it is the first in alphabetical order:

```
sage: 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
```

The roots in terms of x are what we expect:

```
sage: 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)]
```

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 f.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)]
```

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):
...
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)]
```

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]
```

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...
sage: (2.5*f).roots(ring=RR)
[(-0.058811522318449..., 1)]
sage: f.roots(ring=CC, multiplicities=False)
[-0.05881152231844..., -1.331099917875... - 1.52241655183732*I, -1.331099917875... + 1.52241
sage: f.roots(ring=QQ)
[]
sage: f.roots(ring=QQbar, multiplicities=False)
[-0.05881152231844944?, -1.331099917875796? - 1.522416551837318?*I, -1.331099917875796? + 1.
```

Root finding over finite fields:

```
sage: f.roots(ring=GF(7^2, 'a'))
[(3, 1), (4*a + 6, 2), (3*a + 3, 2)]
```

TESTS:

```
sage: (sqrt(3) * f).roots(ring=QQ)
Traceback (most recent call last):
...
TypeError: unable to convert sqrt(3) to a rational
```

Check if trac ticket #9538 is fixed:

```
sage: var('f6,f5,f4,x')
(f6, f5, f4, x)
sage: e=15*f6*x^2 + 5*f5*x + f4
sage: res = e.roots(x); res
[(-1/30*(5*f5 + sqrt(25*f5^2 - 60*f4*f6))/f6, 1), (-1/30*(5*f5 - sqrt(25*f5^2 - 60*f4*f6))/f
sage: e.subs(x=res[0][0]).is_zero()
True
```

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
```

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` - an 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
sage: g.truncate()
x^3 - x^2*sin(y) - 5*x + 3
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
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
```

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)^
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)
```

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
```

TESTS:

Check if trac ticket #8943 is fixed:

```
sage: ((1+arctan(x))**(1/x)).series(x==0, 3)
(e) + (-1/2*e)*x + (1/8*e)*x^2 + Order(x^3)
```

Order may be negative:

```
sage: f = sin(x)^(-2); f.series(x, -1)
1*x^(-2) + Order(1/x)
```

Check if changing global series precision does it right:

```
sage: set_series_precision(3)
sage: (1/(1-2*x)).series(x)
1 + 2*x + 4*x^2 + Order(x^3)
sage: set_series_precision(20)
```

show()

Show this symbolic expression, i.e., typeset it nicely.

EXAMPLES:

```
sage: (x^2 + 1).show()
x^{2}  + 1
```

simplify()

Return a simplified version of this symbolic expression.

Note: Currently, this just sends the expression to Maxima and converts it back 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)
```

TESTS:

Check that trac ticket #14637 is fixed:

```
sage: assume(x > 0, x < pi/2)
sage: acos(cos(x)).simplify()
x
sage: forget()
```

simplify_exp(**args, **kwds*)

Deprecated: Use `canonicalize_radical()` instead. See trac ticket #11912 for details.

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

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)

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)
```

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
```

TESTS:

Check that the problem with applying $full_simplify()$ to gamma functions (trac ticket #9240) has been fixed:

```
sage: gamma(1/3)
gamma(1/3)
sage: gamma(1/3).full_simplify()
gamma(1/3)
sage: gamma(4/3)
gamma(4/3)
sage: gamma(4/3).full_simplify()
1/3*gamma(1/3)
```

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
```

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

TESTS:

There are two square roots of

$$(x+1)^2$$

, so this should not be simplified to

$$x+1$$

, trac ticket #12737:

```
sage: f = sqrt((x + 1)^2)
sage: f.simplify_full()
sqrt(x^2 + 2*x + 1)
```

The imaginary part of an expression should not change under simplification; trac ticket #11934:

```
sage: f = sqrt(-8*(4*sqrt(2) - 7)*x^4 + 16*(3*sqrt(2) - 5)*x^3)
sage: original = f.imag_part()
sage: simplified = f.full_simplify().imag_part()
sage: original - simplified
0
```

The invalid simplification from trac ticket #12322 should not occur after trac ticket #12737:

```
sage: t = var('t')
sage: assume(t, 'complex')
sage: assumptions()
[t is complex]
sage: f = (1/2)*log(2*t) + (1/2)*log(1/t)
sage: f.simplify_full()
1/2*log(2*t) - 1/2*log(t)
sage: forget()
```

Complex logs are not contracted, trac ticket #17556:

```
sage: x,y = SR.var('x,y')
sage: assume(y, 'complex')
sage: f = log(x*y) - (log(x) + log(y))
sage: f.simplify_full()
log(x*y) - log(x) - log(y)
sage: forget()
```

The simplifications from `simplify_rectform()` are performed, trac ticket #17556:

```
sage: f = ( e^(I*x) - e^(-I*x) ) / ( I*e^(I*x) + I*e^(-I*x) )
sage: f.simplify_full()
sin(x)/cos(x)
```

simplify_hypergeometric(*algorithm='maxima'*)

Simplify an expression containing 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)
....:  .simplify_hypergeometric())
laguerre(-laguerre(-e^x, x), x)
sage: (nest(lambda y: hypergeometric([y], [1], x), 3, 1)
....:  .simplify_hypergeometric(algorithm='sage'))
hypergeometric((hypergeometric((e^x,), (1,), x),), (1,), x)
```

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
```

INPUT:

- `self` - expression to be simplified
- `algorithm` - (default: None) optional, 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')
```

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)
```

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)
```

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)

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))
```

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)
```

`x*log(9)` is contracted only if `algorithm` is `'all'`:

```
sage: (x*log(9)).simplify_log()
x*log(9)
sage: (x*log(9)).simplify_log('all')
log(9^x)
```

TESTS:

Ensure that the option `algorithm` from one call has no influence upon future calls (a Maxima flag was set, and we have to ensure that its value has been restored):

```
sage: f = log(x)+2*log(y)+1/2*log(t)
sage: f.simplify_log('one')
1/2*log(t) + log(x) + 2*log(y)

sage: f.simplify_log('ratios')
log(sqrt(t)*x*y^2)

sage: f.simplify_log()
log(x*y^2) + 1/2*log(t)
```

This shows that the issue at trac ticket #7334 is fixed. Maxima intentionally keeps the expression inside the log factored:

```
sage: log_expr = (log(sqrt(2)-1)+log(sqrt(2)+1))
sage: log_expr.simplify_log('all')
log((sqrt(2) + 1)*(sqrt(2) - 1))
sage: _.simplify_rational()
0
```

We should use the current simplification domain rather than set it to 'real' explicitly (trac ticket #12780):

```
sage: f = sqrt(x^2)
sage: f.simplify_log()
sqrt(x^2)
sage: from sage.calculus.calculus import maxima
sage: maxima('domain: real;')
real
sage: f.simplify_log()
abs(x)
sage: maxima('domain: complex;')
complex
```

AUTHORS:

- Robert Marik (11-2009)

simplify_radical(**args*, ***kwds*)

Deprecated: Use `canonicalize_radical()` instead. See trac ticket #11912 for details.

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 commmon denominator and add)
- `map` - (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))

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)
```

With `map=True` each term in a sum is simplified separately and thus the resuls are shorter for functions which are combination of rational and nonrational funtions. 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
```

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
```

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))
```

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)

sage: y = SR.var('y')
sage: f = log(x) + 2*log(y)
sage: f.simplify_real()
log(x*y^2)
```

TESTS:

We set the Maxima `domain` variable to 'real' before we call out to Maxima. When we return, however, we should set the `domain` back to what it was, rather than assuming that it was 'complex':

```
sage: from sage.calculus.calculus import maxima
sage: maxima('domain: real;')
real
sage: x.simplify_real()
x
sage: maxima('domain;')
real
sage: maxima('domain: complex;')
complex
```

We forget the assumptions that our variables are real after simplification; make sure we don't forget an assumption that existed before we were called:

```
sage: assume(x, 'real')
sage: x.simplify_real()
x
sage: assumptions()
[x is real]
sage: forget()
```

We also want to be sure that we don't forget assumptions on other variables:

```
sage: x,y,z = SR.var('x,y,z')
sage: assume(y, 'integer')
sage: assume(z, 'antisymmetric')
sage: x.simplify_real()
x
sage: assumptions()
[y is integer, z is antisymmetric]
sage: forget()
```

No new assumptions should exist after the call:

```
sage: assumptions()
[]
sage: x.simplify_real()
x
sage: assumptions()
[]
```

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)
```

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)
```

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)
```

TESTS:

When given `None`, we should always call rectform() and return the result:

```
sage: polynomials = QQ['x']
sage: f = SR(polynomials.random_element())
sage: g = f.simplify_rectform(complexity_measure = None)
sage: bool(g == f.rectform())
True
```

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` - (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)
```

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)^3 - 3*cos(x))
sage: 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))
sin(1)
sage: sin(SR(RealField(150)(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)
```

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = SR(0).sin(hold=True); a.simplify()
0
```

TESTS:

```
sage: SR(oo).sin()
Traceback (most recent call last):
...
RuntimeError: sin_eval(): sin(infinity) encountered
sage: SR(-oo).sin()
Traceback (most recent call last):
...
RuntimeError: sin_eval(): sin(infinity) encountered
sage: SR(unsigned_infinity).sin()
Traceback (most recent call last):
...
RuntimeError: sin_eval(): sin(infinity) encountered
```

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?
```

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))
```

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)
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = arccosh(x).sinh(hold=True); a.simplify()
sqrt(x + 1)*sqrt(x - 1)
```

TESTS:

```
sage: SR(oo).sinh()
+Infinity
sage: SR(-oo).sinh()
-Infinity
sage: SR(unsigned_infinity).sinh()
Traceback (most recent call last):
...
RuntimeError: sinh_eval(): sinh(unsigned_infinity) encountered
```

solve (*x*, *multiplicities=False*, *solution_dict=False*, *explicit_solutions=False*, *to_poly_solve=False*)

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 - bool (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 - bool (default: False); if True or non-zero, return a list of dictionaries containing solutions. Not used when solving an inequality.
- explicit_solutions - bool (default: False); require that all roots be explicit rather than implicit. Not used when solving an inequality.
- to_poly_solve - bool (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 == e^(2/5*I*pi), z == e^(4/5*I*pi), z == e^(-4/5*I*pi), z == e^(-2/5*I*pi), 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])
```

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
```

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]
```

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]

sage: var('Q')
Q
sage: solve(Q*sqrt(Q^2 + 2) - 1, Q)
[Q == 1/sqrt(Q^2 + 2)]
sage: solve(Q*sqrt(Q^2 + 2) - 1, Q, to_poly_solve=True)
[Q == 1/sqrt(-sqrt(2) + 1), Q == 1/sqrt(sqrt(2) + 1)]
```

In some cases there may be infinitely many solutions indexed by a dummy variable. If it begins with `z`, it is implicitly assumed to be an integer, a real if with `r`, and so on:

```
sage: solve( sin(x)==cos(x), x, to_poly_solve=True)
[x == 1/4*pi + pi*z...]
```

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: assume(x, 'rational')
sage: solve(x^2 == 2, x)
[]
sage: solve(x^2==2-z, x)
[x == -sqrt(-z + 2)]
sage: solve((x-z)^2==2, x)
[x == z - sqrt(2), x == z + sqrt(2)]
```

There is still room for improvement:

```
sage: assume(x, 'integer')
sage: assume(z, 'integer')
sage: solve((x-z)^2==2, x)
[x == z - sqrt(2), x == z + sqrt(2)]

sage: forget()
```

In some cases it may be worthwhile to directly use `to_poly_solve` if one suspects some answers are being missed:

```
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*z77]
```

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...]
```

We can also solve for several variables:

```
sage: var('b, c')
(b, c)
sage: solve((b-1)*(c-1), [b,c])
[[b == 1, c == r4], [b == r5, c == 1]]
```

Some basic inequalities can be also solved:

```
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) # not tested - output depends on system
[[0 < y, y < x, 0 < x]]
[[y < x, 0 < y]]
```

TESTS:

trac ticket #7325 (solving inequalities):

```
sage: (x^2>1).solve(x)
[[x < -1], [x > 1]]
```

Catch error message from Maxima:

```
sage: solve(acot(x),x)
[]

sage: solve(acot(x),x,to_poly_solve=True)
[]
```

trac ticket #7491 fixed:

```
sage: y=var('y')
sage: solve(y==y,y)
[y == r1]
sage: solve(y==y,y,multiplicities=True)
([y == r1], [])

sage: from sage.symbolic.assumptions import GenericDeclaration
sage: GenericDeclaration(x, 'rational').assume()
sage: solve(x^2 == 2, x)
[]
sage: forget()
```

trac ticket #8390 fixed:

```
sage: solve(sin(x)==1/2,x)
[x == 1/6*pi]

sage: solve(sin(x)==1/2,x,to_poly_solve=True)
[x == 1/6*pi]

sage: solve(sin(x)==1/2, x, to_poly_solve='force')
[x == 1/6*pi + 2*pi*z..., x == 5/6*pi + 2*pi*z...]
```

trac ticket #11618 fixed:

```
sage: g(x)=0
sage: solve(g(x)==0,x,solution_dict=True)
[{x: r1}]
```

trac ticket #13286 fixed:

```
sage: solve([x-4], [x])
[x == 4]
```

trac ticket #13645: fixed:

```
sage: x.solve((1,2))
Traceback (most recent call last):
...
TypeError: (1, 2) are not valid variables.
```

trac ticket #17128: fixed:

```
sage: var('x,y')
(x, y)
sage: f = x+y
sage: sol = f.solve([x, y], solution_dict=True)
sage: sol[0].get(x) + sol[0].get(y)
0
```

trac ticket #16651 fixed:

```
sage: (x^7-x-1).solve(x, to_poly_solve=True)     # abs tol 1e-6
[x == 1.11277569705,
 x == (-0.363623519329 - 0.952561195261*I),
 x == (0.617093477784 - 0.900864951949*I),
 x == (-0.809857800594 - 0.262869645851*I),
 x == (-0.809857800594 + 0.262869645851*I),
 x == (0.617093477784 + 0.900864951949*I),
 x == (-0.363623519329 + 0.952561195261*I)]
```

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)
```

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

```
sage: SR(4).sqrt()
2
sage: SR(4).sqrt(hold=True)
sqrt(4)
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = SR(4).sqrt(hold=True); a.simplify()
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'
```

step (*hold=False*)

Return the value of the Heaviside step function, which is 0 for negative x, 1/2 for 0, and 1 for positive x.

EXAMPLES:

```
sage: x = var('x')
sage: SR(1.5).step()
1
sage: SR(0).step()
1/2
sage: SR(-1/2).step()
0
sage: SR(float(-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)
step(2)
```

subs (*in_dict=None*, ***kwds*)

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

# substitute with keyword arguments (works only with symbols)
sage: t.subs(a=c)
(x + y)^3 + b^2 + c^2

# 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

# substitute with a relational expression
sage: t.subs(w0^2 == w0^3)
a^3 + b^3 + (x + y)^3

sage: t.subs(w0==w0^2)
(x^2 + y^2)^18 + a^16 + b^16

# more than one keyword argument is accepted
sage: t.subs(a=b, b=c)
(x + y)^3 + b^2 + c^2

# using keyword arguments with a dictionary is allowed
sage: t.subs({a:b}, b=c)
(x + y)^3 + b^2 + c^2

# in this case keyword arguments override the dictionary
sage: t.subs({a:b}, a=c)
(x + y)^3 + b^2 + c^2
```

```
sage: t.subs({a:b, b:c})
(x + y)^3 + b^2 + c^2
```

TESTS:

```
sage: # no arguments return the same expression
sage: t.subs()
(x + y)^3 + a^2 + b^2

# similarly for an empty dictionary argument
sage: t.subs({})
(x + y)^3 + a^2 + b^2

# non keyword or dictionary argument returns error
sage: t.subs(5)
Traceback (most recent call last):
...
TypeError: subs takes either a set of keyword arguments, a dictionary, or a symbolic relatio

# substitutions with infinity
sage: (x/y).subs(y=oo)
0
sage: (x/y).subs(x=oo)
Traceback (most recent call last):
...
RuntimeError: indeterminate expression: infinity * f(x) encountered.
sage: (x*y).subs(x=oo)
Traceback (most recent call last):
...
RuntimeError: indeterminate expression: infinity * f(x) encountered.
sage: (x^y).subs(x=oo)
Traceback (most recent call last):
...
ValueError: power::eval(): pow(Infinity, f(x)) is not defined.
sage: (x^y).subs(y=oo)
Traceback (most recent call last):
...
ValueError: power::eval(): pow(f(x), infinity) is not defined.
sage: (x+y).subs(x=oo)
+Infinity
sage: (x-y).subs(y=oo)
-Infinity
sage: gamma(x).subs(x=-1)
Infinity
sage: 1/gamma(x).subs(x=-1)
0
```

Verify that this operation does not modify the passed dictionary (trac ticket #6622):

```
sage: var('v t')
(v, t)
sage: f = v*t
sage: D = {v: 2}
sage: f(D, t=3)
6
sage: D
{v: 2}
```

Check if trac ticket #9891 is fixed:

```
sage: exp(x).subs(x=log(x))
x
```

Check if trac ticket #13587 is fixed:

```
sage: t = tan(x)^2 - tan(x)
sage: t.subs(x=pi/2)
Infinity
sage: u = gamma(x) - gamma(x-1)
sage: u.subs(x=-1)
Infinity
```

subs_expr (**equations*)

Given a dictionary of key:value pairs, substitute all occurrences of key for value in self. The substitutions can also be given as a number of symbolic equalities key == value; see the examples.

> **Warning:** This is a formal pattern substitution, which may or may not have any mathematical meaning. The exact rules used at present in Sage are determined by Maxima's subst command. Sometimes patterns are not replaced even though one would think they should be - see examples below.

EXAMPLES:

```
sage: f = x^2 + 1
sage: f.subs_expr(x^2 == x)
x + 1

sage: var('x,y,z'); f = x^3 + y^2 + z
(x, y, z)
sage: f.subs_expr(x^3 == y^2, z == 1)
2*y^2 + 1
```

Or the same thing giving the substitutions as a dictionary:

```
sage: f.subs_expr({x^3:y^2, z:1})
2*y^2 + 1

sage: f = x^2 + x^4
sage: f.subs_expr(x^2 == x)
x^4 + x
sage: f = cos(x^2) + sin(x^2)
sage: f.subs_expr(x^2 == x)
cos(x) + sin(x)

sage: f(x,y,t) = cos(x) + sin(y) + x^2 + y^2 + t
sage: f.subs_expr(y^2 == t)
(x, y, t) |--> x^2 + 2*t + cos(x) + sin(y)
```

The following seems really weird, but it *is* what Maple does:

```
sage: f.subs_expr(x^2 + y^2 == t)
(x, y, t) |--> x^2 + y^2 + t + cos(x) + sin(y)
sage: maple.eval('subs(x^2 + y^2 = t, cos(x) + sin(y) + x^2 + y^2 + t)')          # optional
'cos(x)+sin(y)+x^2+y^2+t'
sage: maxima.quit()
sage: maxima.eval('cos(x) + sin(y) + x^2 + y^2 + t, x^2 + y^2 = t')
'sin(y)+y^2+cos(x)+x^2+t'
```

Actually Mathematica does something that makes more sense:

```
sage: mathematica.eval('Cos[x] + Sin[y] + x^2 + y^2 + t /. x^2 + y^2 -> t')       # optional
2 t + Cos[x] + Sin[y]
```

substitute (*in_dict=None*, ***kwds*)

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

# substitute with keyword arguments (works only with symbols)
sage: t.subs(a=c)
(x + y)^3 + b^2 + c^2

# 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

# substitute with a relational expression
sage: t.subs(w0^2 == w0^3)
a^3 + b^3 + (x + y)^3

sage: t.subs(w0==w0^2)
(x^2 + y^2)^18 + a^16 + b^16

# more than one keyword argument is accepted
sage: t.subs(a=b, b=c)
(x + y)^3 + b^2 + c^2

# using keyword arguments with a dictionary is allowed
sage: t.subs({a:b}, b=c)
(x + y)^3 + b^2 + c^2

# in this case keyword arguments override the dictionary
sage: t.subs({a:b}, a=c)
(x + y)^3 + b^2 + c^2

sage: t.subs({a:b, b:c})
(x + y)^3 + b^2 + c^2
```

TESTS:

```
sage: # no arguments return the same expression
sage: t.subs()
(x + y)^3 + a^2 + b^2

# similarly for an empty dictionary argument
sage: t.subs({})
(x + y)^3 + a^2 + b^2

# non keyword or dictionary argument returns error
sage: t.subs(5)
Traceback (most recent call last):
...
TypeError: subs takes either a set of keyword arguments, a dictionary, or a symbolic relatio
```

```
# substitutions with infinity
sage: (x/y).subs(y=oo)
0
sage: (x/y).subs(x=oo)
Traceback (most recent call last):
...
RuntimeError: indeterminate expression: infinity * f(x) encountered.
sage: (x*y).subs(x=oo)
Traceback (most recent call last):
...
RuntimeError: indeterminate expression: infinity * f(x) encountered.
sage: (x^y).subs(x=oo)
Traceback (most recent call last):
...
ValueError: power::eval(): pow(Infinity, f(x)) is not defined.
sage: (x^y).subs(y=oo)
Traceback (most recent call last):
...
ValueError: power::eval(): pow(f(x), infinity) is not defined.
sage: (x+y).subs(x=oo)
+Infinity
sage: (x-y).subs(y=oo)
-Infinity
sage: gamma(x).subs(x=-1)
Infinity
sage: 1/gamma(x).subs(x=-1)
0
```

Verify that this operation does not modify the passed dictionary (trac ticket #6622):

```
sage: var('v t')
(v, t)
sage: f = v*t
sage: D = {v: 2}
sage: f(D, t=3)
6
sage: D
{v: 2}
```

Check if trac ticket #9891 is fixed:

```
sage: exp(x).subs(x=log(x))
x
```

Check if trac ticket #13587 is fixed:

```
sage: t = tan(x)^2 - tan(x)
sage: t.subs(x=pi/2)
Infinity
sage: u = gamma(x) - gamma(x-1)
sage: u.subs(x=-1)
Infinity
```

substitute_expression(**equations*)

Given a dictionary of key:value pairs, substitute all occurrences of key for value in self. The substitutions can also be given as a number of symbolic equalities key == value; see the examples.

Warning: This is a formal pattern substitution, which may or may not have any mathematical meaning. The exact rules used at present in Sage are determined by Maxima's subst command. Sometimes patterns are not replaced even though one would think they should be - see examples below.

EXAMPLES:

```
sage: f = x^2 + 1
sage: f.subs_expr(x^2 == x)
x + 1

sage: var('x,y,z'); f = x^3 + y^2 + z
(x, y, z)
sage: f.subs_expr(x^3 == y^2, z == 1)
2*y^2 + 1
```

Or the same thing giving the substitutions as a dictionary:

```
sage: f.subs_expr({x^3:y^2, z:1})
2*y^2 + 1

sage: f = x^2 + x^4
sage: f.subs_expr(x^2 == x)
x^4 + x
sage: f = cos(x^2) + sin(x^2)
sage: f.subs_expr(x^2 == x)
cos(x) + sin(x)

sage: f(x,y,t) = cos(x) + sin(y) + x^2 + y^2 + t
sage: f.subs_expr(y^2 == t)
(x, y, t) |--> x^2 + 2*t + cos(x) + sin(y)
```

The following seems really weird, but it *is* what Maple does:

```
sage: f.subs_expr(x^2 + y^2 == t)
(x, y, t) |--> x^2 + y^2 + t + cos(x) + sin(y)
sage: maple.eval('subs(x^2 + y^2 = t, cos(x) + sin(y) + x^2 + y^2 + t)')          # optional
'cos(x)+sin(y)+x^2+y^2+t'
sage: maxima.quit()
sage: maxima.eval('cos(x) + sin(y) + x^2 + y^2 + t, x^2 + y^2 = t')
'sin(y)+y^2+cos(x)+x^2+t'
```

Actually Mathematica does something that makes more sense:

```
sage: mathematica.eval('Cos[x] + Sin[y] + x^2 + y^2 + t /. x^2 + y^2 -> t')       # optional
2 t + Cos[x] + Sin[y]
```

substitute_function (*original*, *new*)

Return this symbolic expressions all occurrences of the function *original* replaced with the function *new*.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: foo = function('foo'); bar = function('bar')
sage: f = foo(x) + 1/foo(pi*y)
sage: f.substitute_function(foo, bar)
1/bar(pi*y) + bar(x)
```

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
```

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

EXAMPLES:

```
sage: k, n = var('k,n')
sage: k.sum(k, 1, n).factor()
1/2*(n + 1)*n

sage: (1/k^4).sum(k, 1, oo)
1/90*pi^4

sage: (1/k^5).sum(k, 1, oo)
zeta(5)
```

A well known binomial identity:

```
sage: assume(n>=0)
sage: binomial(n,k).sum(k, 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
```

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
```

```
sage: (k * binomial(n, k)).sum(k, 1, n)
2^(n - 1)*n

sage: ((-1)^k*binomial(n,k)).sum(k, 0, n)
0

sage: (2^(-k)/(k*(k+1))).sum(k, 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)
```

We check a well known identity:

```
sage: bool((k^3).sum(k, 1, n) == k.sum(k, 1, n)^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)
```

The geometric series:

```
sage: assume(abs(q) < 1)
sage: (a*q^k).sum(k, 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.
```

This summation only Mathematica can perform:

```
sage: (1/(1+k^2)).sum(k, -oo, oo, algorithm = 'mathematica')     # optional - mathematica
pi*coth(pi)
```

Use Giac to perform this summation:

```
sage: (sum(1/(1+k^2), k, -oo, oo, algorithm = 'giac')).factor()       # optional - giac
(e^(2*pi) + 1)*pi/((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
```

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 convertable into a usable Sage expression.

TESTS:

Check that the sum in trac ticket #10682 is done right:

```
sage: sum(binomial(n,k)*k^2, k, 2, n)
1/4*(n^2 + n)*2^n - n
```

This sum used to give a wrong result (trac ticket #9635) but now gives correct results with all relevant assumptions:

```
sage: (n,k,j)=var('n,k,j')
sage: sum(binomial(n,k)*binomial(k-1,j)*(-1)**(k-1-j),k,j+1,n)
-sum((-1)^(-j + k)*binomial(k - 1, j)*binomial(n, k), k, j + 1, n)
sage: assume(j>-1)
sage: sum(binomial(n,k)*binomial(k-1,j)*(-1)**(k-1-j),k,j+1,n)
1
sage: forget()
sage: assume(n>=j)
sage: sum(binomial(n,k)*binomial(k-1,j)*(-1)**(k-1-j),k,j+1,n)
-sum((-1)^(-j + k)*binomial(k - 1, j)*binomial(n, k), k, j + 1, n)
sage: forget()
sage: assume(j==-1)
sage: sum(binomial(n,k)*binomial(k-1,j)*(-1)**(k-1-j),k,j+1,n)
1
sage: forget()
sage: assume(j<-1)
sage: sum(binomial(n,k)*binomial(k-1,j)*(-1)**(k-1-j),k,j+1,n)
-sum((-1)^(-j + k)*binomial(k - 1, j)*binomial(n, k), k, j + 1, n)
sage: forget()
```

Check that trac ticket #16176 is fixed:

```
sage: n = var('n')
sage: sum(log(1-1/n^2),n,2,oo)
-log(2)
```

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
```

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)
```

This also works using functional notation:

```
sage: tan(pi/12,hold=True)
tan(1/12*pi)
sage: tan(pi/12)
-sqrt(3) + 2
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = (pi/12).tan(hold=True); a.simplify()
-sqrt(3) + 2
```

TESTS:

```
sage: SR(oo).tan()
Traceback (most recent call last):
...
RuntimeError: tan_eval(): tan(infinity) encountered
sage: SR(-oo).tan()
Traceback (most recent call last):
...
RuntimeError: tan_eval(): tan(infinity) encountered
sage: SR(unsigned_infinity).tan()
Traceback (most recent call last):
...
RuntimeError: tan_eval(): tan(infinity) encountered
```

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)
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))
```

This also works using functional notation:

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

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = arcsinh(x).tanh(hold=True); a.simplify()
x/sqrt(x^2 + 1)
```

TESTS:

```
sage: SR(oo).tanh()
1
sage: SR(-oo).tanh()
-1
sage: SR(unsigned_infinity).tanh()
Traceback (most recent call last):
...
RuntimeError: tanh_eval(): tanh(unsigned_infinity) encountered
```

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)

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

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

sage: taylor (1/log (x + 1), x, 0, 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

sage: taylor ((cos(x) - sec(x))^3, x, 0, 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
```

TESTS:

Check that ticket trac ticket #7472 is fixed (Taylor polynomial in more 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 +
sage: expand(_)
x*y^3
```

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)
sage: (x < y).test_relation()
False
```

TESTS:

```
sage: all_relations = [op for name, op in sorted(operator.__dict__.items()) if len(name) ==
sage: all_relations
[<built-in function eq>, <built-in function ge>, <built-in function gt>, <built-in function
sage: [op(3, pi).test_relation() for op in all_relations]
[False, False, False, True, True, True]
sage: [op(pi, pi).test_relation() for op in all_relations]
[True, True, False, True, False, False]

sage: s = 'some_very_long_variable_name_which_will_definitely_collide_if_we_use_a_reasonable
sage: t1, t2 = var(','.join([s+'1',s+'2']))
sage: (t1 == t2).test_relation()
False
sage: (cot(pi + x) == 0).test_relation()
NotImplemented
```

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)
2*sin(x*y)
sage: f.trailing_coefficient(y)
x
sage: 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
```

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` - (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` - (default: False) If True, causes half-angles to be simplified away.
- `plus` - (default: True) Controls the sum rule; expansion of sums (e.g. 'sin(x + y)') will take place only if plus is True.
- `times` - (default: True) Controls the product rule, expansion of products (e.g. sin(2*x)) 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)
```

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(
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)
```

ALIASES:

`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
```

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)
```

ALIASES: `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` - (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)
```

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)^3 - 3*cos(x))
sage: 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)
```

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
```

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)
```

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)
```

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
0.003300223685324103 - 0.4181554491413217*I
```

```
sage: CDF(0,1).zeta()   # abs tol 1e-16
0.003300223685324103 - 0.4181554491413217*I
sage: plot(lambda x: SR(x).zeta(), -10,10).show(ymin=-3,ymax=3)
```

To prevent automatic evaluation use the `hold` argument:

```
sage: SR(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
```

To then evaluate again, we currently must use Maxima via `simplify()`:

```
sage: a = SR(2).zeta(hold=True); a.simplify()
1/6*pi^2
```

TESTS:

```
sage: t = SR(1).zeta(); t
Infinity
```

class `sage.symbolic.expression.`**`ExpressionIterator`**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

`next`()

x.next() -> the next value, or raise StopIteration

`sage.symbolic.expression.`**`is_Expression`**(*x*)

Return True if *x* is a symbolic Expression.

EXAMPLES:

```
sage: from sage.symbolic.expression import is_Expression
sage: is_Expression(x)
True
sage: is_Expression(2)
False
sage: is_Expression(SR(2))
True
```

`sage.symbolic.expression.`**`is_SymbolicEquation`**(*x*)

Return True if *x* is a symbolic equation.

EXAMPLES:

The following two examples are symbolic equations:

```
sage: from sage.symbolic.expression import is_SymbolicEquation
sage: is_SymbolicEquation(sin(x) == x)
True
sage: is_SymbolicEquation(sin(x) < x)
True
sage: is_SymbolicEquation(x)
False
```

This is not, since `2==3` evaluates to the boolean `False`:

```
sage: is_SymbolicEquation(2 == 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
```

CHAPTER

TWO

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)
```

```
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)
```

```
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
```

```
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
```

TESTS:

The arguments in the definition must be symbolic variables #10747:

```
sage: f(1)=2
Traceback (most recent call last):
...
SyntaxError: can't assign to function call
```

```
sage: f(x,1)=2
Traceback (most recent call last):
...
SyntaxError: can't assign to function call
```

```
sage: f(1,2)=3
Traceback (most recent call last):
...
SyntaxError: can't assign to function call
```

```
sage: f(1,2)=x
```

```
Traceback (most recent call last):
...
SyntaxError: can't assign to function call

sage: f(x,2)=x
Traceback (most recent call last):
...
SyntaxError: can't assign to function call
```

class sage.symbolic.callable.**CallableSymbolicExpressionFunctor**(*arguments*)

Bases: sage.categories.pushout.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)
```

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)
```

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)
```

unify_arguments(*x*)

Takes the variable list from another CallableSymbolicExpression object and compares 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)
```

AUTHORS:

•Bobby Moretti: thanks to William Stein for the rules

class sage.symbolic.callable.**CallableSymbolicExpressionRingFactory**

Bases: sage.structure.factory.UniqueFactory

INPUT:

•name – string. A name in the global namespace referring to self or a fully qualified path name to self, which is used to locate the factory on unpickling.

EXAMPLES:

```
sage: from sage.structure.factory import UniqueFactory
sage: fake_factory = UniqueFactory('ZZ')
sage: loads(dumps(fake_factory))
Integer Ring
sage: fake_factory = UniqueFactory('sage.rings.all.QQ')
sage: loads(dumps(fake_factory))
Rational Field
```

create_key(*args*, *check=True*)

EXAMPLES:

```
sage: x,y = var('x,y')
sage: CallableSymbolicExpressionRing.create_key((x,y))
(x, y)
```

create_object(*version*, *key*, ***extra_args*)

Returns 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)
```

class sage.symbolic.callable.**CallableSymbolicExpressionRing_class**(*arguments*)

Bases: sage.symbolic.ring.SymbolicRing

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
```

args()

Returns 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)
```

arguments()

Returns 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)
```

construction()

EXAMPLES:

```
sage: f(x,y) = x^2 + y
sage: f.parent().construction()
(CallableSymbolicExpressionFunctor(x, y), Symbolic Ring)
```

`sage.symbolic.callable.`**is_CallableSymbolicExpression**(*x*)

Returns `True` if `x` is a callable symbolic expression.

EXAMPLES:

```
sage: from sage.symbolic.callable import is_CallableSymbolicExpression
sage: var('a x y z')
(a, x, y, z)
sage: f(x,y) = a + 2*x + 3*y + z
sage: is_CallableSymbolicExpression(f)
True
sage: is_CallableSymbolicExpression(a+2*x)
False
sage: def foo(n): return n^2
...
sage: is_CallableSymbolicExpression(foo)
False
```

`sage.symbolic.callable.`**is_CallableSymbolicExpressionRing**(*x*)

Return `True` if `x` is a callable symbolic expression ring.

INPUT:

- x - object

OUTPUT: bool

EXAMPLES:

```
sage: from sage.symbolic.callable import is_CallableSymbolicExpressionRing
sage: is_CallableSymbolicExpressionRing(QQ)
False
sage: var('x,y,z')
(x, y, z)
sage: is_CallableSymbolicExpressionRing(CallableSymbolicExpressionRing((x,y,z)))
True
```

CHAPTER

THREE

ASSUMPTIONS

class sage.symbolic.assumptions.**GenericDeclaration**(*var*, *assumption*)

Bases: sage.structure.sage_object.SageObject

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).

INPUT:

- var – the variable about which assumptions are being made
- assumption – a 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)
sin(pi*x)
sage: sin(x*pi).simplify()
0
sage: decl.forget()
sage: sin(x*pi)
sin(pi*x)
sage: sin(x*pi).simplify()
sin(pi*x)
```

Here is the list of acceptable features:

```
sage: maxima('features')
[integer,noninteger,even,odd,rational,irrational,real,imaginary,complex,analytic,increasing,decr
```

assume()

Make this assumption.

TEST:

```
sage: from sage.symbolic.assumptions import GenericDeclaration
sage: decl = GenericDeclaration(x, 'even')
sage: decl.assume()
sage: cos(x*pi).simplify()
1
sage: decl2 = GenericDeclaration(x, 'odd')
sage: decl2.assume()
Traceback (most recent call last):
...
```

```
ValueError: Assumption is inconsistent
sage: decl.forget()
```

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)
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
```

forget()

Forget this assumption.

TEST:

```
sage: from sage.symbolic.assumptions import GenericDeclaration
sage: decl = GenericDeclaration(x, 'odd')
sage: decl.assume()
sage: cos(pi*x)
```

```
cos(pi*x)
sage: cos(pi*x).simplify()
-1
sage: decl.forget()
sage: cos(x*pi).simplify()
cos(pi*x)
```

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
```

sage.symbolic.assumptions.**assume** (**args*)

Make the given assumptions.

INPUT:

- `*args` – assumptions

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
```

This will be assumed in the current Sage session until forgotten:

```
sage: forget()
sage: bool(sqrt(x^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)
sage: assume(n+1>0)
sage: integral(x^n,x)
x^(n + 1)/(n + 1)
sage: 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()
```

An integer constraint:

```
sage: var('n, P, r, r2')
(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]
```

Simplifying certain well-known identities works as well:

```
sage: sin(n*pi)
sin(pi*n)
sage: sin(n*pi).simplify()
0
sage: forget()
sage: sin(n*pi).simplify()
sin(pi*n)
```

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):
...
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()
```

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()
```

TESTS:

Test that you can do two non-relational declarations at once (fixing trac ticket #7084):

```
sage: var('m,n')
(m, n)
sage: assume(n, 'integer'); assume(m, 'integer')
sage: sin(n*pi).simplify()
0
sage: sin(m*pi).simplify()
0
sage: forget()
sage: sin(n*pi).simplify()
sin(pi*n)
sage: sin(m*pi).simplify()
sin(pi*m)
```

`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: list(sorted(assumptions(), lambda x,y:cmp(str(x),str(y))))
[x > y, z > w]
sage: forget()
sage: 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]
```

`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: var('x,y,z')
(x, y, z)
sage: assume(x>0, y>0, z == 1, y>0)
sage: list(sorted(assumptions(), lambda x,y:cmp(str(x),str(y))))
[x > 0, y > 0, z == 1]
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()
[]
```

`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)
sage: preprocess_assumptions([x, y, 'integer', x > 4, y, 'even'])
[x is integer, y is integer, x > 4, y is even]
```

CHAPTER

FOUR

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
]
```

4.1 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>
```

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
```

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
```

4.2 Arithmetic

Add two symbolic equations:

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

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
```

Multiply two symbolic equations:

```
sage: x = var('x')
sage: m = x == 5*x + 1
sage: n = sin(x) == sin(x+2*pi)
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
```

Divide two symbolic equations:

```
sage: x = var('x')
sage: m = x == 5*x + 1
sage: n = sin(x) == sin(x+2*pi)
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)
sage: m/n
x/sin(x) != (5*x + 1)/sin(2*pi + x)
```

4.3 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)
```

4.4 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.
```

We illustrate finding multiplicities of solutions:

```
sage: f = (x-1)^5*(x^2+1)
sage: solve(f == 0, x)
[x == -I, x == I, x == 1]
sage: solve(f == 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)]]
```

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
```

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.
```

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.
```

4.5 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()
[]
```

4.6 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
```

Conversion to Maple:

```
sage: x = var('x')
sage: eq = (x == 2)
sage: 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
```

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)
```

LaTeX output:

```
sage: latex(x^(3/5) >= pi)
x^{\frac{3}{5}} \geq \pi
```

When working with the symbolic complex number I, notice that comparison do not automatically simplifies even in trivial situations:

```
sage: I^2 == -1
-1 == -1
sage: I^2 < 0
-1 < 0
sage: (I+1)^4 > 0
-4 > 0
```

Nevertheless, if you force the comparison, you get the right answer (trac ticket #7160):

```
sage: bool(I^2 == -1)
True
sage: bool(I^2 < 0)
True
sage: bool((I+1)^4 > 0)
False
```

4.7 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)]
```

```
sage: f = x^5 + a
sage: solve(f==0,x)
[x == (-a)^(1/5)*e^(2/5*I*pi), x == (-a)^(1/5)*e^(4/5*I*pi), x == (-a)^(1/5)*e^(-4/5*I*pi), x == (-a)
```

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
```

TESTS:

We test serializing symbolic equations:

```
sage: eqn = x^3 + 2/3 >= x
sage: loads(dumps(eqn))
x^3 + 2/3 >= x
sage: loads(dumps(eqn)) == eqn
True
```

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` - bool (default: False); if True or non-zero, 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` - bool (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` - bool (default: False); require that all roots be explicit rather than implicit. Not used when solving inequalities.
- `to_poly_solve` - bool (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).

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)
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 solution[x].n(digits=3), ",", solution[y].n(digits=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, due to the underlying algorithm in Maxima:

```
sage: sols = solve([x^3==y,y^2==x],[x,y]); sols[-1], sols[0]
([x == 0, y == 0], [x == (0.3090169943749475 + 0.9510565162951535*I), y == (-0.8090169943749475
sage: sols[0][0].rhs().pyobject().parent()
Complex Double Field
```

If `f` is only one equation or expression, we use the solve method for symbolic expressions, which defaults to exact answers only:

```
sage: solve([y^6==y],y)
[y == e^(2/5*I*pi), y == e^(4/5*I*pi), y == e^(-4/5*I*pi), y == e^(-2/5*I*pi), y == 1, y == 0]
sage: solve( [y^6 == y], y)==solve( y^6 == y, y)
True
```

Here we demonstrate very basic use of the optional keywords for a single expression to be solved:

```
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]
```

Note: For more details about solving a single equation, see the documentation for the single-expression `solve()`.

```
sage: from sage.symbolic.expression import Expression
sage: Expression.solve(x^2==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.
```

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
```

If there is a parameter in the answer, that will show up as a new variable. In the following example, `r1` is a real free variable (because of the `r`):

```
sage: solve([x+y == 3, 2*x+2*y == 6],x,y)
[[x == -r1 + 3, y == r1]]
```

Especially with trigonometric functions, the dummy variable may be implicitly an integer (hence the `z`):

```
sage: solve([cos(x)*sin(x) == 1/2, x+y == 0],x,y)
[[x == 1/4*pi + pi*z79, y == -1/4*pi - pi*z79]]
```

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]]
```

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]
```

Here, the first equation evaluates to `False`, so there are no solutions:

```
sage: solve([1==3, 1.00000000000000*x^3 == 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)]]
```

Inequalities can be also solved:

```
sage: solve(x^2>8,x)
[[x < -2*sqrt(2)], [x > 2*sqrt(2)]]
```

We use `use_grobner` in Maxima if no solution is obtained from Maxima's `to_poly_solve`:

```
sage: x,y=var('x y'); c1(x,y)=(x-5)^2+y^2-16; 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(11)*sqrt(5) + 123/68], [x == 9/68*sqrt(55) + 13
```

TESTS:

```
sage: solve([sin(x)==x,y^2==x],x,y)
[sin(x) == x, y^2 == x]
sage: solve(0==1,x)
Traceback (most recent call last):
...
TypeError:  The first argument must be a symbolic expression or a list of symbolic expressions.
```

Test if the empty list is returned, too, when (a list of) dictionaries (is) are requested (#8553):

```
sage: solve([SR(0)==1],x)
[]
sage: solve([SR(0)==1],x,solution_dict=True)
[]
sage: solve([x==1,x==-1],x)
[]
sage: solve([x==1,x==-1],x,solution_dict=True)
[]
sage: solve((x==1,x==-1),x,solution_dict=0)
[]
```

Relaxed form, suggested by Mike Hansen (#8553):

```
sage: solve([x^2-1],x,solution_dict=-1)
[{x: -1}, {x: 1}]
sage: solve([x^2-1],x,solution_dict=1)
[{x: -1}, {x: 1}]
sage: solve((x==1,x==-1),x,solution_dict=-1)
[]
sage: solve((x==1,x==-1),x,solution_dict=1)
[]
```

This inequality holds for any real `x` (trac #8078):

```
sage: solve(x^4+2>0,x)
[x < +Infinity]
```

Test for user friendly input handling trac ticket #13645:

```
sage: poly.<a,b> = PolynomialRing(RR)
sage: solve([a+b+a*b == 1], a)
Traceback (most recent call last):
...
TypeError: The first argument to solve() should be a symbolic expression or a list of symbolic e
sage: solve([a, b], (1, a))
Traceback (most recent call last):
...
TypeError: 1 is not a valid variable.
sage: solve([x == 1], (1, a))
Traceback (most recent call last):
...
TypeError: (1, a) are not valid variables.
```

Test that the original version of a system in the French Sage book now works (trac ticket #14306):

```
sage: var('y,z')
(y, z)
sage: solve([x^2 * y * z == 18, x * y^3 * z == 24, x * y * z^4 == 6], x, y, z)
[[x == 3, y == 2, z == 1], [x == (1.337215067... - 2.685489874...*I), y == (-1.700434271... + 1.
```

`sage.symbolic.relation.`**`solve_ineq`**(*ineq*, *vars=None*)

Solves 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 varible. 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
```

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)]]
```

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)]]
```

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]) # not tested - 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*)

Solves sytem of inequalities using Maxima and fourier elimination

Can be used for system of linear inequalities and for some types of non-linear inequalities. For examples see the section EXAMPLES 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])
[[y < min(x - 4, -x + 9)]]

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]]
```

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)]) # not tested - 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]]
```

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` - an integer
- `solution_dict` - bool (default: False); if True or non-zero, 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,)]
```

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,
```

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
```

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,)]
```

We solve a simple equation modulo 2:

```
sage: x,y = var('x,y')
sage: solve_mod([x == y], 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.

TESTS:

Make sure that we short-circuit in at least some cases:

```
sage: solve_mod([2*x==1], 2*next_prime(10^50))
[]
```

Try multi-equation cases:

```
sage: x, y, z = var("x y z")
sage: solve_mod([2*x^2 + x*y, -x*y+2*y^2+x-2*y, -2*x^2+2*x*y-y^2-x-y], 12)
[(0, 0), (4, 4), (0, 3), (4, 7)]
sage: eqs = [-y^2+z^2, -x^2+y^2-3*z^2-z-1, -y*z-z^2-x-y+2, -x^2-12*z^2-y+z]
sage: solve_mod(eqs, 11)
[(8, 5, 6)]
```

Confirm that modulus 1 now behaves as it should:

```
sage: x, y = var("x y")
sage: solve_mod([x==1], 1)
[(0,)]
sage: solve_mod([2*x^2+x*y, -x*y+2*y^2+x-2*y, -2*x^2+2*x*y-y^2-x-y], 1)
[(0, 0)]
```

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)]
```

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)]
```

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
```

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
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()
```

TESTS:

Ensure that `canonicalize_radical()` and `simplify_log` are not used inappropriately, trac ticket #17389. Either one would simplify `f` to zero below:

```
sage: x,y = SR.var('x,y')
sage: assume(y, 'complex')
sage: f = log(x*y) - (log(x) + log(y))
sage: f(x=-1, y=i)
-2*I*pi
sage: test_relation_maxima(f == 0)
False
sage: forget()
```

Ensure that the `sqrt(x^2)` -> `abs(x)` simplification is not performed when testing equality:

```
sage: assume(x, 'complex')
sage: f = sqrt(x^2) - abs(x)
sage: test_relation_maxima(f == 0)
False
sage: forget()
```

If assumptions are made, `simplify_rectform()` is used:

```
sage: assume(x, 'real')
sage: f1 = ( e^(I*x) - e^(-I*x) ) / ( I*e^(I*x) + I*e^(-I*x) )
sage: f2 = sin(x)/cos(x)
sage: test_relation_maxima(f1 - f2 == 0)
True
sage: forget()
```

But not if x itself is complex:

```
sage: assume(x, 'complex')
sage: f1 = ( e^(I*x) - e^(-I*x) ) / ( I*e^(I*x) + I*e^(-I*x) )
sage: f2 = sin(x)/cos(x)
sage: test_relation_maxima(f1 - f2 == 0)
False
sage: forget()
```

If assumptions are made, then `simplify_factorial()` is used:

```
sage: n,k = SR.var('n,k')
sage: assume(n, 'integer')
sage: assume(k, 'integer')
sage: f1 = factorial(n+1)/factorial(n)
sage: f2 = n + 1
sage: test_relation_maxima(f1 - f2 == 0)
True
sage: forget()
```

CHAPTER

FIVE

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 Trac #9217

The Sage calculus module is loosely based on the Sage Enhancement Proposal found at: http://www.sagemath.org:9001/CalculusSEP.

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
```

Sage predefines `x` to be a global indeterminate. Thus the following works:

```
sage: x^2
x^2
sage: type(x)
<type '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))
```

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)
```

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)
```

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)
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]
```

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
```

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.
```

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
```

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))]
```

And complex exponentiation works now:

```
sage: M = i*matrix([[pi]])
sage: e^M
[-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]
```

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)
```

Also we can substitute with keywords:

```
sage: f = sin(x*y - z)
sage: f(x = t, y = z)
sin(t*z - z)
```

It was formerly the case that if there was no ambiguity of variable names, we didn't have to specify them; that still works for the moment, but the behavior is deprecated:

```
sage: f = sin(x)
sage: f(y)
doctest:...: DeprecationWarning: Substitution using function-call
syntax and unnamed arguments is deprecated and will be removed
from a future release of Sage; you can use named arguments instead,
like EXPR(x=..., y=...)
See http://trac.sagemath.org/5930 for details.
sin(y)
sage: f(pi)
0
```

However if there is ambiguity, we should explicitly state what variables we're substituting for:

```
sage: f = sin(2*pi*x/y)
sage: f(x=4)
sin(8*pi/y)
```

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))
```

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)
```

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))
```

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
```

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)
1/3/sqrt(1/9*x^2 + 1)
```

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
```

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
```

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
cos(1/2*I) + sin(I)
sage: CC(f)
1.12762596520638 + 1.17520119364380*I
sage: ComplexField(200)(f)
1.1276259652063807852262251614026720125478471180986674836290 + 1.175201193643801456882381850595600815
sage: ComplexField(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
```

Note that after calling var, the variables are immediately available for use:

```
sage: (n1 + n2)^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
```

TESTS:

Substitution:

```
sage: f = x
sage: f(x=5)
5
```

Simplifying expressions involving scientific notation:

```
sage: k = var('k')
sage: a0 = 2e-06; a1 = 12
sage: c = a1 + a0*k; c
(2.00000000000000e-6)*k + 12
sage: sqrt(c)
sqrt((2.00000000000000e-6)*k + 12)
sage: sqrt(c^3)
sqrt(((2.00000000000000e-6)*k + 12)^3)
```

The symbolic calculus package uses its own copy of Maxima for simplification, etc., which is separate from the default system-wide version:

```
sage: maxima.eval('[x,y]: [1,2]')
'[1,2]'
sage: maxima.eval('expand((x+y)^3)')
'27'
```

If the copy of maxima used by the symbolic calculus package were the same as the default one, then the following would return 27, which would be very confusing indeed!

```
sage: x, y = var('x,y')
sage: expand((x+y)^3)
x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

Set x to be 5 in maxima:

```
sage: maxima('x: 5')
5
sage: maxima('x + x + %pi')
%pi+10
```

Simplifications like these are now done using Pynac:

```
sage: x + x + pi
pi + 2*x
```

But this still uses Maxima:

```
sage: (x + x + pi).simplify()
pi + 2*x
```

Note that `x` is still `x`, since the maxima used by the calculus package is different than the one in the interactive interpreter.

Check to see that the problem with the variables method mentioned in trac ticket #3779 is actually fixed:

```
sage: f = function('F',x)
sage: diff(f*SR(1),x)
D[0](F)(x)
```

Doubly ensure that trac ticket #7479 is working:

```
sage: f(x)=x
sage: integrate(f,x,0,1)
1/2
```

Check that the problem with Taylor expansions of the gamma function (trac ticket #9217) is fixed:

```
sage: taylor(gamma(1/3+x),x,0,3)
-1/432*((72*euler_gamma^3 + 36*euler_gamma^2*(sqrt(3)*pi + 9*log(3)) +
27*pi^2*log(3) + 243*log(3)^3 + 18*euler_gamma*(6*sqrt(3)*pi*log(3) + pi^2
+ 27*log(3)^2 + 12*psi(1, 1/3)) + 324*log(3)*psi(1, 1/3) + sqrt(3)*(pi^3 +
9*pi*(9*log(3)^2 + 4*psi(1, 1/3))))*gamma(1/3) - 72*psi(2,
1/3)*gamma(1/3))*x^3 + 1/24*(6*sqrt(3)*pi*log(3) + 12*euler_gamma^2 + pi^2
+ 4*euler_gamma*(sqrt(3)*pi + 9*log(3)) + 27*log(3)^2 + 12*psi(1,
1/3))*x^2*gamma(1/3) - 1/6*(6*euler_gamma + sqrt(3)*pi +
9*log(3))*x*gamma(1/3) + gamma(1/3)
sage: map(lambda f:f[0].n(), _.coefficients())  # numerical coefficients to make comparison easier; l
[2.6789385347..., -8.3905259853..., 26.662447494..., -80.683148377...]
```

Ensure that ticket #8582 is fixed:

```
sage: k = var("k")
sage: sum(1/(1+k^2), k, -oo, oo)
-1/2*I*psi(I + 1) + 1/2*I*psi(-I + 1) - 1/2*I*psi(I) + 1/2*I*psi(-I)
```

Ensure that ticket #8624 is fixed:

```
sage: integrate(abs(cos(x)) * sin(x), x, pi/2, pi)
1/2
sage: integrate(sqrt(cos(x)^2 + sin(x)^2), x, 0, 2*pi)
2*pi
```

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
```

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*D[0, 0, 0, 0](u)(x) + 1/6*h^3*D[0, 0, 0](u)(x) + 1/2*h^2*D[0, 0](u)(x) + h*D[0](u)(x) +
```

We compute a Laplace transform:

```
sage: var('s,t')
(s, t)
sage: f=function('f', t)
sage: f.diff(t,2)
D[0, 0](f)(t)
sage: f.diff(t,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 (trac ticket #12796):

```
sage: from sage.calculus.calculus import at
sage: f = function('f')
sage: at(f(x), [x == 1])
f(1)
```

TESTS:

Our one non-keyword argument must be a list:

```
sage: from sage.calculus.calculus import at
sage: f = function('f')
sage: at(f(x), x == 1)
Traceback (most recent call last):
...
TypeError: at can take at most one argument, which must be a list
```

We should convert our first argument to a symbolic expression:

```
sage: from sage.calculus.calculus import at
sage: at(int(1), x=1)
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)
```

Here the function is used implicitly:

```
sage: a = var('a')
sage: f = function('cr', a)
sage: g = f.diff(a); g
D[0](cr)(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(x) = function('f',x)
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)
```

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(s) = function('F',s)
sage: 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')
```

```
sage: f(t) = function('f',t)
sage: dummy_laplace(f(t),t,s)
laplace(f(t), t, s)
```

`sage.calculus.calculus.`**`dummy_limit`**(**args*)

This function is called to create formal wrappers of limits that Maxima can't compute:

EXAMPLES:

```
sage: a = lim(exp(x^2)*(1-erf(x)), x=infinity); a
-limit((erf(x) - 1)*e^(x^2), x, +Infinity)
sage: a = sage.calculus.calculus.dummy_limit(sin(x)/x, x, 0);a
limit(sin(x)/x, x, 0)
```

`sage.calculus.calculus.`**`inverse_laplace`**(*ex*, *t*, *s*)

Attempts to compute the inverse Laplace transform of `self` with respect to the variable t and transform parameter s. If this function cannot find a solution, a formal function is returned.

The function that is returned may be be viewed as a function of s.

DEFINITION: The inverse Laplace transform of a function $F(s)$, is the function $f(t)$ defined by

$$F(s) = \frac{1}{2\pi i} \int_{\gamma - i\infty}^{\gamma + i\infty} e^{st} F(s) dt,$$

where γ is chosen so that the contour path of integration is in the region of convergence of $F(s)$.

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)
```

No explicit inverse Laplace transform, so one is returned formally as a function `ilt`:

```
sage: inverse_laplace(cos(s), s, t)
ilt(cos(s), s, t)
```

`sage.calculus.calculus.`**`laplace`**(*ex*, *t*, *s*)

Attempts to compute and return the Laplace transform of `self` with respect to the variable t and transform parameter s. If this function cannot find a solution, a formal function is returned.

The function that is returned may be 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

$$F(s) = \int_0^\infty e^{-st} f(t) dt.$$

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
```

We do a formal calculation:

```
sage: f = function('f', x)
sage: g = f.diff(x); g
D[0](f)(x)
sage: g.laplace(x, s)
s*laplace(f(x), x, s) - f(0)
```

EXAMPLES:

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)
```

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))
sage: p2 = plot(yt,0,1/2,rgbcolor=(0,1,0))
sage: (p1+p2).save(os.path.join(SAGE_TMP, "de_plot.png"))
```

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)
t*e^(a + 2*t)*sin(t)
```

Unable to compute solution:

```
sage: laplace(1/s, s, t)
laplace(1/s, s, t)
```

`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='above')
```

INPUT:

- `dir` - (default: None); dir may have the value 'plus' (or '+' or 'right') for a limit from above, 'minus' (or '-' or 'left') 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
sage: f.limit(x = 1.2)
2.06961575467...
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
```

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?
```

With this example, Maxima is looking for a LOT of information:

```
sage: assume(a>0)
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 an integer?
sage: assume(a,'integer')
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 an even number?
sage: assume(a,'even')
sage: limit(x^a,x=0)
0
sage: 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

sage: f = log(log(x))/log(x)
sage: forget(); assume(x<-2); lim(f, x=0, taylor=True)
```

```
0
sage: forget()
```

Here ind means “indefinite but bounded”:

```
sage: lim(sin(1/x), x = 0)
ind
```

TESTS:

```
sage: lim(x^2, x=2, dir='nugget')
Traceback (most recent call last):
...
ValueError: dir must be one of None, 'plus', '+', 'right',
'minus', '-', 'left'
```

We check that trac ticket #3718 is fixed, so that Maxima gives correct limits for the floor function:

```
sage: limit(floor(x), x=0, dir='-')
-1
sage: limit(floor(x), x=0, dir='+')
0
sage: limit(floor(x), x=0)
und
```

Maxima gives the right answer here, too, showing that trac ticket #4142 is fixed:

```
sage: f = sqrt(1-x^2)
sage: g = diff(f, x); g
-x/sqrt(-x^2 + 1)
sage: limit(g, x=1, dir='-')
-Infinity

sage: limit(1/x, x=0)
Infinity
sage: limit(1/x, x=0, dir='+')
+Infinity
sage: limit(1/x, x=0, dir='-')
-Infinity
```

Check that trac ticket #8942 is fixed:

```
sage: f(x) = (cos(pi/4-x) - tan(x)) / (1 - sin(pi/4+x))
sage: limit(f(x), x = pi/4, dir='minus')
+Infinity
sage: limit(f(x), x = pi/4, dir='plus')
-Infinity
sage: limit(f(x), x = pi/4)
Infinity
```

Check that we give deprecation warnings for ‘above’ and ‘below’, trac ticket #9200:

```
sage: limit(1/x, x=0, dir='above')
doctest:...: DeprecationWarning: the keyword
'above' is deprecated. Please use 'right' or '+' instead.
See http://trac.sagemath.org/9200 for details.
+Infinity
sage: limit(1/x, x=0, dir='below')
doctest:...: DeprecationWarning: the keyword
'below' is deprecated. Please use 'left' or '-' instead.
```

```
See http://trac.sagemath.org/9200 for details.
-Infinity
```

Check that trac ticket #12708 is fixed:

```
sage: limit(tanh(x),x=0)
0
```

Check that trac ticket #15386 is fixed:

```
sage: n = var('n')
sage: assume(n>0)
sage: sequence = -(3*n^2 + 1)*(-1)^n/sqrt(n^5 + 8*n^3 + 8)
sage: limit(sequence, n=infinity)
0
```

`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='above')
```

INPUT:

- `dir` - (default: None); dir may have the value 'plus' (or '+' or 'right') for a limit from above, 'minus' (or '-' or 'left') 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
sage: f.limit(x = 1.2)
2.06961575467...
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
```

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?
```

With this example, Maxima is looking for a LOT of information:

```
sage: assume(a>0)
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 an integer?
sage: assume(a,'integer')
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 an even number?
sage: assume(a,'even')
sage: limit(x^a,x=0)
0
sage: 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

sage: f = log(log(x))/log(x)
sage: forget(); assume(x<-2); lim(f, x=0, taylor=True)
0
sage: forget()
```

Here ind means "indefinite but bounded":

```
sage: lim(sin(1/x), x = 0)
ind
```

TESTS:

```
sage: lim(x^2, x=2, dir='nugget')
Traceback (most recent call last):
...
ValueError: dir must be one of None, 'plus', '+', 'right',
'minus', '-', 'left'
```

We check that trac ticket #3718 is fixed, so that Maxima gives correct limits for the floor function:

```
sage: limit(floor(x), x=0, dir='-')
-1
sage: limit(floor(x), x=0, dir='+')
0
sage: limit(floor(x), x=0)
und
```

Maxima gives the right answer here, too, showing that trac ticket #4142 is fixed:

```
sage: f = sqrt(1-x^2)
sage: g = diff(f, x); g
-x/sqrt(-x^2 + 1)
sage: limit(g, x=1, dir='-')
-Infinity

sage: limit(1/x, x=0)
Infinity
sage: limit(1/x, x=0, dir='+')
+Infinity
sage: limit(1/x, x=0, dir='-')
-Infinity
```

Check that trac ticket #8942 is fixed:

```
sage: f(x) = (cos(pi/4-x) - tan(x)) / (1 - sin(pi/4+x))
sage: limit(f(x), x = pi/4, dir='minus')
+Infinity
sage: limit(f(x), x = pi/4, dir='plus')
-Infinity
sage: limit(f(x), x = pi/4)
Infinity
```

Check that we give deprecation warnings for 'above' and 'below', trac ticket #9200:

```
sage: limit(1/x, x=0, dir='above')
doctest:...: DeprecationWarning: the keyword
'above' is deprecated. Please use 'right' or '+' instead.
See http://trac.sagemath.org/9200 for details.
```

```
+Infinity
sage: limit(1/x, x=0, dir='below')
doctest:...: DeprecationWarning: the keyword
'below' is deprecated. Please use 'left' or '-' instead.
See http://trac.sagemath.org/9200 for details.
-Infinity
```

Check that trac ticket #12708 is fixed:

```
sage: limit(tanh(x),x=0)
0
```

Check that trac ticket #15386 is fixed:

```
sage: n = var('n')
sage: assume(n>0)
sage: sequence = -(3*n^2 + 1)*(-1)^n/sqrt(n^5 + 8*n^3 + 8)
sage: limit(sequence, n=infinity)
0
```

`sage.calculus.calculus.`**`mapped_opts`**`(`*v*`)`

Used internally when creating a string of options to pass to Maxima.

INPUT:

- `v` - an object

OUTPUT: a 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'
```

`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,foo=bar,another=false'
```

`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 bit/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 pi, 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 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 non-zero, 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
```

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 -
```

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
```

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
```

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)*s
sage: 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))^2 - 40)*(sqrt(5) + sqrt(3) + sqrt(2))^2 + 352)*(sqrt(5) + sqrt(
sage: f(a).expand()
0
```

Here we show use of the `epsilon` parameter. That this result is actually exact can be shown using the addition formula for sin, but maxima is unable to see that.

```
sage: a = sin(pi/5)
sage: a.minpoly(algorithm='numerical')
Traceback (most recent call last):
...
NotImplementedError: Could not prove minimal polynomial x^4 - 5/4*x^2 + 5/16 (epsilon 0.00000000
sage: f = a.minpoly(algorithm='numerical', epsilon=1e-100); f
x^4 - 5/4*x^2 + 5/16
sage: f(a).numerical_approx(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
```

There is a difference between algorithm='algebraic' and algorithm='numerical':

```
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 +
sage: cos(pi/33).minpoly(algorithm='numerical')
Traceback (most recent call last):
...
NotImplementedError: Could not prove minimal polynomial x^10 + 1/2*x^9 - 5/2*x^8 - 5/4*x^7 + 17/
```

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).
```

Note: Of course, failure to produce a minimal polynomial does not necessarily indicate that this number is transcendental.

`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) maxima 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)
```

EXAMPLES:

```
sage: f(x) = exp(-sqrt(x))
sage: f.nintegral(x, 0, 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)
```

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
```

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
```

Computing to higher precision we see the truth:

```
sage: f.n(200)
-7.4992740280181431112064614366622348652078895136533593355718e-13
sage: f.n(300)
-7.49927402801814311120646143662663009137292462589621789352095066181709095575681963967103004e-13
```

Now numerically integrating, we see why the answer is wrong:

```
sage: f.nintegrate(x,0,1)
(-480.0000000000001, 5.329070518200754e-12, 21, 0)
```

It is just because every floating point evaluation of return -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))')
'2.565728500561051482917356396 E-127'             # 32-bit
'2.5657285005610514829173563961304785900 E-127'   # 64-bit
sage: old_prec = gp.set_real_precision(50)
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304785900147709554020 E-127'
sage: 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) maxima 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)
```

EXAMPLES:

```
sage: f(x) = exp(-sqrt(x))
sage: f.nintegral(x, 0, 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)
```

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
```

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
```

Computing to higher precision we see the truth:

```
sage: f.n(200)
-7.4992740280181431112064614366622348652078895136533593355718e-13
sage: f.n(300)
-7.49927402801814311120646143662663009137292462589621789352095066181709095575681963967103004e-13
```

Now numerically integrating, we see why the answer is wrong:

```
sage: f.nintegrate(x,0,1)
(-480.0000000000001, 5.329070518200754e-12, 21, 0)
```

It is just because every floating point evaluation of return -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))')
'2.5657285005610514829173563961304785900 E-127'      # 32-bit
'2.5657285005610514829173563961304785900 E-127'   # 64-bit
sage: old_prec = gp.set_real_precision(50)
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304785900147709554020 E-127'
sage: 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` - a string
- `equals_sub` - (default: False) if True, replace '=' by '==' in self
- `maxima` - (default: the calculus package's Maxima) the Maxima interpreter to use.

EXAMPLES:

```
sage: from sage.calculus.calculus import symbolic_expression_from_maxima_string as sefms
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
```

TESTS:

trac ticket #8459 fixed:

```
sage: maxima('3*li[2](u)+8*li[33](exp(u))').sage()
8*polylog(33, e^u) + 3*polylog(2, u)
```

Check if trac ticket #8345 is fixed:

```
sage: assume(x,'complex')
sage: t = x.conjugate()
sage: latex(t)
\overline{x}
sage: latex(t._maxima_()._sage_())
\overline{x}
```

Check that we can understand maxima's not-equals (trac ticket #8969):

```
sage: from sage.calculus.calculus import symbolic_expression_from_maxima_string as sefms
sage: sefms("x!=3") == (factorial(x) == 3)
True
sage: sefms("x # 3") == SR(x != 3)
```

```
True
sage: solve([x != 5], x)
#0: solve_rat_ineq(ineq=_SAGE_VAR_x # 5)
[[x - 5 != 0]]
sage: solve([2*x==3, x != 5], x)
[[x == (3/2), (-7/2) != 0]]
```

Make sure that we don't accidentally pick up variables in the maxima namespace (trac #8734):

```
sage: sage.calculus.calculus.maxima('my_new_var : 2')
2
sage: var('my_new_var').full_simplify()
my_new_var
```

ODE solution constants are treated differently (trac ticket #16007):

```
sage: from sage.calculus.calculus import symbolic_expression_from_maxima_string as sefms
sage: sefms('%k1*x + %k2*y + %c')
_K1*x + _K2*y + _C
```

Check that some hypothetical variables don't end up as special constants (trac ticket #6882):

```
sage: from sage.calculus.calculus import symbolic_expression_from_maxima_string as sefms
sage: sefms('%i')^2
-1
sage: ln(sefms('%e'))
1
sage: sefms('i')^2
_i^2
sage: sefms('I')^2
_I^2
sage: sefms('ln(e)')
ln(_e)
sage: sefms('%inf')
+Infinity
```

sage.calculus.calculus.**symbolic_expression_from_string**(*s*, *syms=None*, *accept_sequence=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` - a string
- `syms` - (default: None) dictionary of strings to be regarded as symbols or functions
- `accept_sequence` - (default: False) controls whether to allow a (possibly nested) set of lists and tuples as input

EXAMPLES:

```
sage: y = var('y')
sage: sage.calculus.calculus.symbolic_expression_from_string('[sin(0)*x^2,3*spam+e^pi]',syms={'s
[0, 3*y + e^pi]
```

sage.calculus.calculus.**symbolic_sum**(*expression*, *v*, *a*, *b*, *algorithm='maxima'*)

Returns 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

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

sage: symbolic_sum(1/k^4, k, 1, oo)
1/90*pi^4

sage: symbolic_sum(1/k^5, k, 1, oo)
zeta(5)
```

A well known binomial identity:

```
sage: symbolic_sum(binomial(n,k), k, 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
```

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

sage: symbolic_sum(k * binomial(n, k), k, 1, n)
2^(n - 1)*n

sage: symbolic_sum((-1)^k*binomial(n,k), k, 0, n)
0

sage: symbolic_sum(2^(-k)/(k*(k+1)), k, 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)
```

We check a well known identity:

```
sage: bool(symbolic_sum(k^3, k, 1, n) == symbolic_sum(k, k, 1, n)^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)
```

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)
```

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
[]
```

This summation only Mathematica can perform:

```
sage: symbolic_sum(1/(1+k^2), k, -oo, oo, algorithm = 'mathematica')     # optional - mathematic
pi*coth(pi)
```

An example of this summation with Giac:

```
sage: symbolic_sum(1/(1+k^2), k, -oo, oo, algorithm = 'giac')           # optional - giac
-(pi*e^(-2*pi) - pi*e^(2*pi))/(e^(-2*pi) + e^(2*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
```

TESTS:

trac ticket #10564 is fixed:

```
sage: sum (n^3 * x^n, n, 0, infinity)
(x^3 + 4*x^2 + x)/(x^4 - 4*x^3 + 6*x^2 - 4*x + 1)
```

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 convertable into a Sage expression.

`sage.calculus.calculus.`**`var_cmp`**(*x*, *y*)

Return comparison of the two variables x and y, which is just the comparison of the underlying string representations of the variables. This is used internally by the Calculus package.

INPUT:

- `x, y` - symbolic variables

OUTPUT: Python integer; either -1, 0, or 1.

EXAMPLES:

```
sage: sage.calculus.calculus.var_cmp(x,x)
0
sage: sage.calculus.calculus.var_cmp(x,var('z'))
-1
sage: sage.calculus.calculus.var_cmp(x,var('a'))
1
```

CHAPTER

SIX

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
```

This unit acts exactly like a symbolic variable:

```
sage: s = units.length.meter
sage: s^2
meter^2
sage: s + var('x')
meter + x
```

Units have additional information in their docstring:

```
sage: # You would type: units.force.dyne?
sage: print units.force.dyne._sage_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
```

Calling the convert function with no target returns base SI units:

```
sage: 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
```

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
```

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
```

TESTS:

Check that Trac 12373 if fixed:

```
sage: b = units.amount_of_substance.mole
sage: b.convert(units.amount_of_substance.elementary_entity)
6.02214129000000e23*elementary_entity
```

AUTHORS:

- David Ackerman
- William Stein

class sage.symbolic.units.**UnitExpression**

Bases: sage.symbolic.expression.Expression

A symbolic unit.

EXAMPLES:

```
sage: acre = units.area.acre
sage: type(acre)
<class 'sage.symbolic.units.UnitExpression'>
```

TESTS:

```
sage: bool(loads(dumps(acre)) == acre)
True
sage: type(loads(dumps(acre)))
<class 'sage.symbolic.units.UnitExpression'>
```

class sage.symbolic.units.**Units**(*data*, *name*='')

A collection of units of a some type.

EXAMPLES:

```
sage: units.power
Collection of units of power: cheval_vapeur horsepower watt
```

trait_names()

Return completions of this unit objects. This is used by the Sage command line and notebook to create the list of method names.

EXAMPLES:

```
sage: units.area.trait_names()
['acre', 'are', 'barn', 'hectare', 'rood', 'section', 'square_chain', 'square_meter', 'towns
```

`sage.symbolic.units.`**`base_units`**(*unit*)

Converts unit to base SI units.

INPUT:

- `unit`

OUTPUT:

- *symbolicexpression*

EXAMPLES:

```
sage: 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
```

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
```

Returns variable if 'unit' is not a unit:

```
sage: sage.symbolic.units.base_units(var('x'))
x
```

`sage.symbolic.units.`**`convert`**(*expr*, *target*)

Converts 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:

- *symbolicexpression*

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
```

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
```

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.gali
762/25*galileo
sage: sage.symbolic.units.convert(units.mass.kilogram*units.length.meter/units.time.second^2, un
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.p
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.p
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
8.83795706233228*degree
sage: sage.symbolic.units.convert(50 * units.length.light_year / units.time.year, units.length.f
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)
```

sage.symbolic.units.**convert_temperature**(*expr*, *target*)

Function for converting between temperatures.

INPUT:

- $expr$ – a unit of temperature
- $target$ – a units of temperature

OUTPUT:

- $symbolic expression$

EXAMPLES:

```
sage: t = 32*units.temperature.fahrenheit
sage: t.convert(units.temperature.celsius)
0
sage: t.convert(units.temperature.kelvin)
273.150000000000*kelvin
```

If target is None then it defaults to kelvin:

```
sage: t.convert()
273.150000000000*kelvin
```

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
```

We directly call the convert_temperature function:

```
sage: sage.symbolic.units.convert_temperature(37*units.temperature.celsius, units.temperature.fa
493/5*fahrenheit
sage: 493/5.0
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()
```

`sage.symbolic.units.`**`is_unit`**`(`*s*`)`

Returns a boolean when asked whether the input is in the list of units.

INPUT:

- s – an object

OUTPUT:

- `bool`

EXAMPLES:

```
sage: sage.symbolic.units.is_unit(1)
False
sage: 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
```

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
```

`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:

- 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'>
```

`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:

- 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)
```

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'
```

`sage.symbolic.units.`**`unitdocs`**`(`*unit*`)`

Returns docstring for the given unit.

INPUT:

- `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
sage: 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.
```

`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']
```

CHAPTER

SEVEN

THE SYMBOLIC RING

class sage.symbolic.ring.**NumpyToSRMorphism**

Bases: sage.categories.morphism.Morphism

A Morphism which constructs Expressions from NumPy floats and complexes by converting them to elements of either RDF or CDF.

EXAMPLES:

```
sage: import numpy
sage: from sage.symbolic.ring import NumpyToSRMorphism
sage: f = NumpyToSRMorphism(numpy.float64, SR)
sage: f(numpy.float64('2.0'))
2.0
sage: _.parent()
Symbolic Ring
```

class sage.symbolic.ring.**SymbolicRing**

Bases: sage.rings.ring.CommutativeRing

Symbolic Ring, parent object for all symbolic expressions.

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)
<type 'sage.rings.integer.Integer'>
```

is_exact()

Return False, because there are approximate elements in the symbolic ring.

EXAMPLES:

```
sage: SR.is_exact()
False
```

Here is an inexact element.

```
sage: SR(1.9393)
1.93930000000000
```

is_field(*proof=True*)

Returns True, since the symbolic expression ring is (for the most part) a field.

EXAMPLES:

```
sage: SR.is_field()
True
```

pi()

EXAMPLES:

```
sage: SR.pi() is pi
True
```

symbol(*name=None, latex_name=None, domain=None*)

EXAMPLES:

```
sage: t0 = SR.symbol("t0")
sage: t0.conjugate()
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...
```

var(*name, latex_name=None, domain=None*)

Return the symbolic variable defined by x as an element of the symbolic ring.

EXAMPLES:

```
sage: zz = SR.var('zz'); zz
zz
sage: type(zz)
<type 'sage.symbolic.expression.Expression'>
sage: t = SR.var('theta2'); t
theta2
```

TESTS:

```
sage: var(' x y  z    ')
(x, y, z)
sage: var(' x   ,  y ,  z    ')
(x, y, z)
sage: var(' ')
Traceback (most recent call last):
...
ValueError: You need to specify the name of the new variable.
```

```
var(['x', 'y ', ' z '])
(x, y, z)
var(['x,y'])
Traceback (most recent call last):
...
ValueError: The name "x,y" is not a valid Python identifier.
```

Check that trac ticket #17206 is fixed:

```
sage: var1 = var('var1', latex_name=r'\sigma^2_1'); latex(var1)
{\sigma^2_1}
```

wild (*n=0*)

Return the n-th wild-card for pattern matching and substitution.

INPUT:

- n - a 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)
```

class sage.symbolic.ring.**UnderscoreSageMorphism**

Bases: sage.categories.morphism.Morphism

A Morphism which constructs Expressions from an arbitrary Python object by calling the _sage_() method on the object.

EXAMPLES:

```
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
```

sage.symbolic.ring.**is_SymbolicExpressionRing** (*R*)

Returns True if *R* is the symbolic expression ring.

EXAMPLES:

```
sage: from sage.symbolic.ring import is_SymbolicExpressionRing
sage: is_SymbolicExpressionRing(ZZ)
False
```

```
sage: is_SymbolicExpressionRing(SR)
True
```

sage.symbolic.ring.**is_SymbolicVariable**(*x*)

Returns True if x is a variable.

EXAMPLES:

```
sage: from sage.symbolic.ring import is_SymbolicVariable
sage: is_SymbolicVariable(x)
True
sage: is_SymbolicVariable(x+2)
False
```

TESTS:

```
sage: ZZ['x']
Univariate Polynomial Ring in x over Integer Ring
```

sage.symbolic.ring.**isidentifier**(*x*)

Return whether `x` is a valid identifier.

When we switch to Python 3 this function can be replaced by the official Python function of the same name.

INPUT:

- `x` – a string.

OUTPUT:

Boolean. Whether the string `x` can be used as a variable name.

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.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
```

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
```

TESTS:

These examples test that variables can only be made from valid identifiers. See Trac 7496 (and 9724) for details:

```
sage: var(' ')
Traceback (most recent call last):
...
ValueError: You need to specify the name of the new variable.
sage: var('3')
Traceback (most recent call last):
...
ValueError: The name "3" is not a valid Python identifier.
```

CHAPTER

EIGHT

CLASSES FOR SYMBOLIC FUNCTIONS

class sage.symbolic.function.**BuiltinFunction**

Bases: sage.symbolic.function.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.**DeprecatedSFunction**

Bases: sage.symbolic.function.SymbolicFunction

EXAMPLES:

```
sage: from sage.symbolic.function import DeprecatedSFunction
sage: foo = DeprecatedSFunction("foo", 2)
sage: foo
foo
sage: foo(x,2)
foo(x, 2)
sage: foo(2)
Traceback (most recent call last):
...
TypeError: Symbolic function foo takes exactly 2 arguments (1 given)
```

class sage.symbolic.function.**Function**

Bases: sage.structure.sage_object.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()

Returns a default variable.

EXAMPLES:

```
sage: sin.default_variable()
x
```

name()

Returns the name of this function.

EXAMPLES:

```
sage: foo = function("foo", nargs=2)
sage: foo.name()
'foo'
```

number_of_arguments()

Returns the number of arguments that this function takes.

EXAMPLES:

```
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)
```

variables()

Returns the variables (of which there are none) present in this SFunction.

EXAMPLES:

```
sage: sin.variables()
()
```

class sage.symbolic.function.**GinacFunction**

Bases: sage.symbolic.function.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.

sage.symbolic.function.**PrimitiveFunction**

alias of DeprecatedSFunction

sage.symbolic.function.**SFunction**

alias of DeprecatedSFunction

class sage.symbolic.function.**SymbolicFunction**

Bases: sage.symbolic.function.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.**get_sfunction_from_serial**(*serial*)

Returns an already created SFunction given the serial. These are stored in the dictionary $sage.symbolic.function.sfunction_serial_dict$.

EXAMPLES:

```
sage: from sage.symbolic.function import get_sfunction_from_serial
sage: get_sfunction_from_serial(65) #random
f
```

sage.symbolic.function.**is_inexact**(*x*)

Returns True if the argument is an inexact object.

TESTS:

```
sage: from sage.symbolic.function import is_inexact
sage: is_inexact(5)
doctest:...: DeprecationWarning: The is_inexact() function is deprecated, use the _is_numerical(
See http://trac.sagemath.org/17130 for details.
False
sage: is_inexact(5.)
True
sage: is_inexact(pi)
True
sage: is_inexact(5r)
False
sage: is_inexact(5.4r)
True
```

sage.symbolic.function.**pickle_wrapper**(*f*)

Returns a pickled version of the function f if f is not None; otherwise, it returns 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: pickle_wrapper(f)
"csage...."
sage: pickle_wrapper(None) is None
True
```

sage.symbolic.function.**unpickle_wrapper**(*p*)

Returns a unpickled version of the function defined by *p* if *p* is not None; otherwise, it returns 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
```

CHAPTER

NINE

FACTORY FOR SYMBOLIC FUNCTIONS

sage.symbolic.function_factory.**deprecated_custom_evalf_wrapper**(*func*)

This is used while pickling old symbolic functions that define a custom evalf method.

The protocol for numeric evaluation functions was changed to include a `parent` argument instead of `prec`. This function creates a wrapper around the old custom method, which extracts the precision information from the given `parent`, and passes it on to the old function.

EXAMPLES:

```
sage: from sage.symbolic.function_factory import deprecated_custom_evalf_wrapper as dcew
sage: def old_func(x, prec=0): print "x: %s, prec: %s"%(x,prec)
sage: new_func = dcew(old_func)
sage: new_func(5, parent=RR)
x: 5, prec: 53
sage: new_func(0r, parent=ComplexField(100))
x: 0, prec: 100
```

sage.symbolic.function_factory.**eval_on_operands**(*f*)

Given a method `f` return a new method which takes a single symbolic expression argument and appends operands of the given expression to the arguments of `f`.

EXAMPLES:

```
sage: def f(ex, x, y):
....:     '''
....:     Some documentation.
....:     '''
....:     return x + 2*y
....:
sage: f(None, x, 1)
x + 2
sage: from sage.symbolic.function_factory import eval_on_operands
sage: g = eval_on_operands(f)
sage: g(x + 1)
x + 2
sage: g.__doc__.strip()
'Some documentation.'
```

sage.symbolic.function_factory.**function**(*s*, **args*, ***kwds*)

Create a formal symbolic function with the name *s*.

INPUT:

- `args` - arguments to the function, if specified returns the new function evaluated at the given arguments
- `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 - a 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 - bool 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: var('a, b')
(a, b)
sage: f = function('cr', a)
sage: g = f.diff(a).integral(b)
sage: g
b*D[0](cr)(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)
```

In Sage 4.0, 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))
```

In 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.func
```

You now need to evaluate the function in order to do the arithmetic:

```
sage: 2*f(x)
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*D[0, 0](psi)(r) + 2*r*D[0](psi)(r))/r^2
sage: g.expand()
2*D[0](psi)(r)/r + D[0, 0](psi)(r)
sage: g.coefficient(psi.derivative(r,2))
1
sage: g.coefficient(psi.derivative(r,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

sage: foo = function("foo", nargs=1, conjugate_func=ev)
sage: foo(x).conjugate()
2*x

sage: def deriv(self, *args,**kwds): print 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 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 args, kwds; return sum(args[0]^i for i in range(kwd
sage: foo = function("foo", nargs=1, series_func=expand)
sage: foo(y).series(y, 5)
(y,) {'var': y, 'options': 0, 'at': 0, 'order': 5}
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)
```

Chain rule:

```
sage: def print_args(self, *args, **kwds): print "args:",args; print "kwds:",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
```

TESTS:

Make sure that trac ticket #15860 is fixed and whitespaces are removed:

```
sage: function('A, B')
(A, B)
sage: B
B
sage: C, D, E = function(' C  D E')
sage: C(D(x))
C(D(x))
sage: E
E
```

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
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
```

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
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_wr
sage: nf
g
sage: nf(2)
```

```
g(2)
sage: nf(2).n()
9.00000000000000
sage: nf(2).conjugate()
1
```

CHAPTER

TEN

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)
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
sage: laplace( e^(x+a), x, a)
e^a/(a - 1)
sage: inverse_laplace( e^a/(a-1), x, a)
ilt(e^a/(a - 1), x, a)
```

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
```

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
```

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
```

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

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

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
```

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
(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
```

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
```

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
```

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

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

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
```

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
```

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
```

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
```

TESTS:

```
sage: t1 = (sqrt(3)-3)*(sqrt(3)+1)/6;
sage: tt1 = -1/sqrt(3);
sage: t2 = sqrt(3)/6;
sage: float(t1)
-0.577350269189625...
sage: float(tt1)
-0.577350269189625...
sage: float(t2)
0.28867513459481287
sage: float(expand(t1 + t2))
-0.288675134594812...
sage: float(expand(tt1 + t2))
-0.288675134594812...
```

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
```

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)
```

```
sage: var('a,b')
(a, b)
sage: assume(b-a>0)
sage: integral( sin(x), x, a, b)
cos(a) - cos(b)
sage: 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)

sage: integral( exp(-x^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)
sage: P.show(ymin=0, ymax=0.4)
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)
```

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...
```

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
```

Sage is unable to do anything with the following integral:

```
sage: integral( exp(-x^2)*log(x), x )
integrate(e^(-x^2)*log(x), x)
```

Note, however, that:

```
sage: integral( exp(-x^2)*ln(x), x, 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
```

Sage can't do this elliptic integral (yet):

```
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
integrate(1/sqrt(2*t^4 - 3*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
```

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.
```

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)

sage: [float(h(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
```

`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
```

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)
```

```
sage: var('a,b')
(a, b)
sage: assume(b-a>0)
sage: integral( sin(x), x, a, b)
cos(a) - cos(b)
sage: 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)

sage: integral( exp(-x^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)
sage: P.show(ymin=0, ymax=0.4)
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)
```

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...
```

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
```

Sage is unable to do anything with the following integral:

```
sage: integral( exp(-x^2)*log(x), x )
integrate(e^(-x^2)*log(x), x)
```

Note, however, that:

```
sage: integral( exp(-x^2)*ln(x), x, 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
```

Sage can't do this elliptic integral (yet):

```
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
integrate(1/sqrt(2*t^4 - 3*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
```

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.
```

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)

sage: [float(h(i)) for i in range(5)] #random

[0.0,
 -1.1102230246251565e-16,
 -5.5511151231257827e-17,
 -5.5511151231257827e-17,
 -6.9388939039072284e-17]
sage: h.factor()
0
sage: bool(h == 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
```

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)
```

`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
```

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)
```

`sage.calculus.functional.`**`simplify`**(f)

Simplify the expression f.

EXAMPLES: We simplify the expression $i + x - x$.

```
sage: f = I + x - x; simplify(f)
I
```

In fact, printing f yields the same thing - i.e., the simplified form.

`sage.calculus.functional.taylor(f, *args)`

Expands 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

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 +

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 +
```

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
```

CHAPTER

ELEVEN

SYMBOLIC INTEGRATION

class sage.symbolic.integration.integral.**DefiniteIntegral**

Bases: sage.symbolic.function.BuiltinFunction

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
```

class sage.symbolic.integration.integral.**IndefiniteIntegral**

Bases: sage.symbolic.function.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
```

sage.symbolic.integration.integral.**integral**(*expression*, *v=None*, *a=None*, *b=None*, *algorithm=None*, *hold=False*)

Returns 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') one of

–'maxima' - use maxima (the default)

–'sympy' - use sympy (also in Sage)

–'mathematica_free' - use http://integrals.wolfram.com/

–'fricas' - use FriCAS (the optional fricas spkg has to be installed)

To prevent automatic evaluation use the `hold` argument.

EXAMPLES:

```
sage: x = var('x')
sage: h = sin(x)/(cos(x))^2
sage: 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)
```

```
sage: f = x*cos(x^2)
sage: f.integral(x, 0, sqrt(pi))
0
sage: f.integral(x, a=-pi, b=pi)
0
```

```
sage: f(x) = sin(x)
sage: f.integral(x, 0, pi/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)
```

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
```

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()
```

Usually the constraints are of sign, but others are possible:

```
sage: assume(n==-1)
sage: 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.
```

But Sage can calculate the convergent improper integral of this function:

```
sage: integrate(1/x^3,x,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)
```

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
```

Alternatively, just use algorithm='mathematica_free' to integrate via Mathematica over the internet (does NOT require a Mathematica license!):

```
sage: _ = var('x, y, z')
sage: f = sin(x^2) + y^z
sage: f.integrate(x, algorithm="mathematica_free")    # optional - internet
x*y^z + sqrt(1/2)*sqrt(pi)*fresnels(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')
-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")  # see Trac #14694
Traceback (most recent call last):
...
AttributeError: 'Piecewise' object has no attribute '_sage_'
```

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)
```

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^3+2*x+1)), x); A
-1/73*integrate((x^2 + 4*x + 18)/(x^3 + 2*x + 1), x) + 1/73*log(x - 4)
```

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)
```

An example of an integral that fricas can integrate, but the default integrator cannot:

```
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)) + log(sqrt(x + sqrt(x^2 + 1)) - 1)
- log(sqrt(x + sqrt(x^2 + 1)) + 1) - 2*arctan(sqrt(x + sqrt(x^2 + 1)))
```

The following definite integral is not found with the default integrator:

```
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)
```

```
integrate((x^4 - 3*x^2 + 6)/(x^6 - 5*x^4 + 5*x^2 + 4), x, 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(1/2) + arctan(2) + arctan(5) + arctan(8)
sage: integrate(f(x), x, 1, 2, algorithm="sympy")
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

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?
```

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(
```

TESTS:

The following integral was broken prior to Maxima 5.15.0 - see trac ticket #3013:

```
sage: integrate(sin(x)*cos(10*x)*log(x), x)
-1/198*(9*cos(11*x) - 11*cos(9*x))*log(x) + 1/44*Ei(11*I*x) - 1/36*Ei(9*I*x) - 1/36*Ei(-9*I*x) +
```

It is no longer possible to use certain functions without an explicit variable. Instead, evaluate the function at a variable, and then take the integral:

```
sage: integrate(sin)
Traceback (most recent call last):
...
TypeError

sage: integrate(sin(x), x)
-cos(x)
sage: integrate(sin(x), x, 0, 1)
-cos(1) + 1
```

Check if trac ticket #780 is fixed:

```
sage: _ = var('x,y')
sage: f = log(x^2+y^2)
sage: res = integral(f,x,0.0001414, 1.); res
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints; using the 'assume'
Is 50015104*y^2-50015103 positive, negative or zero?
sage: assume(y>1)
```

```
sage: res = integral(f,x,0.0001414, 1.); res
-2*y*arctan(0.0001414/y) + 2*y*arctan(1/y) + log(y^2 + 1.0) - 0.0001414*log(y^2 + 1.999395999999
sage: nres = numerical_integral(f.subs(y=2), 0.0001414, 1.); nres
(1.4638323264144..., 1.6251803529759...e-14)
sage: res.subs(y=2).n()
1.46383232641443
sage: nres = numerical_integral(f.subs(y=.5), 0.0001414, 1.); nres
(-0.669511708872807, 7.768678110854711e-15)
sage: res.subs(y=.5).n()
-0.669511708872807
```

Check if trac ticket #6816 is fixed:

```
sage: var('t,theta')
(t, theta)
sage: integrate(t*cos(-theta*t),t,0,pi)
(pi*theta*sin(pi*theta) + cos(pi*theta))/theta^2 - 1/theta^2
sage: integrate(t*cos(-theta*t),(t,0,pi))
(pi*theta*sin(pi*theta) + cos(pi*theta))/theta^2 - 1/theta^2
sage: integrate(t*cos(-theta*t),t)
(t*theta*sin(t*theta) + cos(t*theta))/theta^2
sage: integrate(x^2,(x)) # this worked before
1/3*x^3
sage: integrate(x^2,(x,)) # this didn't
1/3*x^3
sage: integrate(x^2,(x,1,2))
7/3
sage: integrate(x^2,(x,1,2,3))
Traceback (most recent call last):
...
ValueError: invalid input (x, 1, 2, 3) - please use variable, with or without two endpoints
```

Note that this used to be the test, but it is actually divergent (though Maxima currently returns the principal value):

```
sage: integrate(t*cos(-theta*t),(t,-oo,oo))
0
```

Check if trac ticket #6189 is fixed:

```
sage: n = N; n
<function numerical_approx at ...>
sage: F(x)  = 1/sqrt(2*pi*1^2)*exp(-1/(2*1^2)*(x-0)^2)
sage: G(x)  = 1/sqrt(2*pi*n(1)^2)*exp(-1/(2*n(1)^2)*(x-n(0))^2)
sage: integrate( (F(x)-F(x))^2, x, -infinity, infinity).n()
0.000000000000000
sage: integrate( ((F(x)-G(x))^2).expand(), x, -infinity, infinity).n()
-6.26376265908397e-17
sage: integrate( (F(x)-G(x))^2, x, -infinity, infinity).n()# abstol 1e-6
0
```

This was broken before Maxima 5.20:

```
sage: exp(-x*i).integral(x,0,1)
I*e^(-I) - I
```

Test deprecation warning when variable is not specified:

```
sage: x.integral()
doctest:...: DeprecationWarning:
```

```
Variable of integration should be specified explicitly.
See http://trac.sagemath.org/12438 for details.
1/2*x^2
```

Test that trac ticket #8729 is fixed:

```
sage: t = var('t')
sage: a = sqrt((sin(t))^2 + (cos(t))^2)
sage: integrate(a, t, 0, 2*pi)
2*pi
sage: a.simplify_full().simplify_trig()
1
```

Maxima uses Cauchy Principal Value calculations to integrate certain convergent integrals. Here we test that this does not raise an error message (see trac ticket #11987):

```
sage: integrate(sin(x)*sin(x/3)/x^2, x, 0, oo)
1/6*pi
```

Maxima returned a negative value for this integral prior to maxima-5.24 (trac ticket #10923). Ideally we would get an answer in terms of the gamma function; however, we get something equivalent:

```
sage: actual_result = integral(e^(-1/x^2), x, 0, 1)
sage: actual_result.canonicalize_radical()
(sqrt(pi)*(erf(1)*e - e) + 1)*e^(-1)
sage: ideal_result = 1/2*gamma(-1/2, 1)
sage: error = actual_result - ideal_result
sage: error.numerical_approx() # abs tol 1e-10
0
```

We will not get an evaluated answer here, which is better than the previous (wrong) answer of zero. See trac ticket #10914:

```
sage: f = abs(sin(x))
sage: integrate(f, x, 0, 2*pi)  # long time (4s on sage.math, 2012)
integrate(abs(sin(x)), x, 0, 2*pi)
```

Another incorrect integral fixed upstream in Maxima, from trac ticket #11233:

```
sage: a,t = var('a,t')
sage: assume(a>0)
sage: assume(x>0)
sage: f = log(1 + a/(x * t)^2)
sage: F = integrate(f, t, 1, Infinity)
sage: F(x=1, a=7).numerical_approx() # abs tol 1e-10
4.32025625668262
sage: forget()
```

Verify that MinusInfinity works with sympy (trac ticket #12345):

```
sage: integral(1/x^2, x, -infinity, -1, algorithm='sympy')
1
```

Check that trac ticket #11737 is fixed:

```
sage: N(integrate(sin(x^2)/(x^2), x, 1, infinity))
0.285736646322853
```

Check that trac ticket #14209 is fixed:

```
sage: integral(e^(-abs(x))/cosh(x),x,-infinity,infinity)
2*log(2)
sage: integral(e^(-abs(x))/cosh(x),x,-infinity,infinity)
2*log(2)
```

Check that trac ticket #12628 is fixed:

```
sage: var('z,n')
(z, n)
sage: f(z, n) = sin(n*z) / (n*z)
sage: integrate(f(z,1)*f(z,3)*f(z,5)*f(z,7),z,0,oo)
22/315*pi
sage: for k in srange(1, 16, 2):
....:     print integrate(prod(f(z, ell)
....:                     for ell in srange(1, k+1, 2)), z, 0, oo)
1/2*pi
1/6*pi
1/10*pi
22/315*pi
3677/72576*pi
48481/1247400*pi
193359161/6227020800*pi
5799919/227026800*pi
```

Check that trac ticket #12628 is fixed:

```
sage: integrate(1/(sqrt(x)*((1+sqrt(x))^2)),x,1,9)
1/2
```

`sage.symbolic.integration.integral.`**`integrate`**(*expression*, *v=None*, *a=None*, *b=None*, *algorithm=None*, *hold=False*)

Returns 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') one of
  - 'maxima' - use maxima (the default)
  - 'sympy' - use sympy (also in Sage)
  - 'mathematica_free' - use http://integrals.wolfram.com/
  - 'fricas' - use FriCAS (the optional fricas spkg has to be installed)

To prevent automatic evaluation use the `hold` argument.

EXAMPLES:

```
sage: x = var('x')
sage: h = sin(x)/(cos(x))^2
sage: 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)
```

```
sage: f = x*cos(x^2)
sage: f.integral(x, 0, sqrt(pi))
0
sage: f.integral(x, a=-pi, b=pi)
0
```

```
sage: f(x) = sin(x)
sage: f.integral(x, 0, pi/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)
```

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
```

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()
```

Usually the constraints are of sign, but others are possible:

```
sage: assume(n==-1)
sage: 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.
```

But Sage can calculate the convergent improper integral of this function:

```
sage: integrate(1/x^3,x,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)
```

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
```

Alternatively, just use algorithm='mathematica_free' to integrate via Mathematica over the internet (does NOT require a Mathematica license!):

```
sage: _ = var('x, y, z')
sage: f = sin(x^2) + y^z
sage: f.integrate(x, algorithm="mathematica_free")    # optional - internet
x*y^z + sqrt(1/2)*sqrt(pi)*fresnels(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')
-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")  # see Trac #14694
Traceback (most recent call last):
...
AttributeError: 'Piecewise' object has no attribute '_sage_'
```

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)
```

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^3+2*x+1)), x); A
-1/73*integrate((x^2 + 4*x + 18)/(x^3 + 2*x + 1), x) + 1/73*log(x - 4)
```

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)
```

An example of an integral that fricas can integrate, but the default integrator cannot:

```
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)) + log(sqrt(x + sqrt(x^2 + 1)) - 1)
- log(sqrt(x + sqrt(x^2 + 1)) + 1) - 2*arctan(sqrt(x + sqrt(x^2 + 1)))
```

The following definite integral is not found with the default integrator:

```
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)
integrate((x^4 - 3*x^2 + 6)/(x^6 - 5*x^4 + 5*x^2 + 4), x, 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(1/2) + arctan(2) + arctan(5) + arctan(8)
sage: integrate(f(x), x, 1, 2, algorithm="sympy")
-1/2*pi + arctan(8) + arctan(5) + arctan(2) + arctan(1/2)
```

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?
```

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(
```

TESTS:

The following integral was broken prior to Maxima 5.15.0 - see trac ticket #3013:

```
sage: integrate(sin(x)*cos(10*x)*log(x), x)
-1/198*(9*cos(11*x) - 11*cos(9*x))*log(x) + 1/44*Ei(11*I*x) - 1/36*Ei(9*I*x) - 1/36*Ei(-9*I*x) +
```

It is no longer possible to use certain functions without an explicit variable. Instead, evaluate the function at a variable, and then take the integral:

```
sage: integrate(sin)
Traceback (most recent call last):
...
TypeError

sage: integrate(sin(x), x)
-cos(x)
sage: integrate(sin(x), x, 0, 1)
-cos(1) + 1
```

Check if trac ticket #780 is fixed:

```
sage: _ = var('x,y')
sage: f = log(x^2+y^2)
sage: res = integral(f,x,0.0001414, 1.); res
Traceback (most recent call last):
...
ValueError: Computation failed since Maxima requested additional constraints; using the 'assume'
Is 50015104*y^2-50015103 positive, negative or zero?
sage: assume(y>1)
sage: res = integral(f,x,0.0001414, 1.); res
-2*y*arctan(0.0001414/y) + 2*y*arctan(1/y) + log(y^2 + 1.0) - 0.0001414*log(y^2 + 1.999395999999
sage: nres = numerical_integral(f.subs(y=2), 0.0001414, 1.); nres
(1.4638323264144..., 1.6251803529759...e-14)
sage: res.subs(y=2).n()
1.46383232641443
sage: nres = numerical_integral(f.subs(y=.5), 0.0001414, 1.); nres
(-0.669511708872807, 7.768678110854711e-15)
sage: res.subs(y=.5).n()
-0.669511708872807
```

Check if trac ticket #6816 is fixed:

```
sage: var('t,theta')
(t, theta)
sage: integrate(t*cos(-theta*t),t,0,pi)
(pi*theta*sin(pi*theta) + cos(pi*theta))/theta^2 - 1/theta^2
sage: integrate(t*cos(-theta*t),(t,0,pi))
(pi*theta*sin(pi*theta) + cos(pi*theta))/theta^2 - 1/theta^2
sage: integrate(t*cos(-theta*t),t)
(t*theta*sin(t*theta) + cos(t*theta))/theta^2
sage: integrate(x^2,(x)) # this worked before
1/3*x^3
sage: integrate(x^2,(x,)) # this didn't
1/3*x^3
sage: integrate(x^2,(x,1,2))
7/3
sage: integrate(x^2,(x,1,2,3))
Traceback (most recent call last):
...
ValueError: invalid input (x, 1, 2, 3) - please use variable, with or without two endpoints
```

Note that this used to be the test, but it is actually divergent (though Maxima currently returns the principal value):

```
sage: integrate(t*cos(-theta*t),(t,-oo,oo))
0
```

Check if trac ticket #6189 is fixed:

```
sage: n = N; n
<function numerical_approx at ...>
sage: F(x) = 1/sqrt(2*pi*1^2)*exp(-1/(2*1^2)*(x-0)^2)
sage: G(x) = 1/sqrt(2*pi*n(1)^2)*exp(-1/(2*n(1)^2)*(x-n(0))^2)
sage: integrate( (F(x)-F(x))^2, x, -infinity, infinity).n()
0.000000000000000
sage: integrate( ((F(x)-G(x))^2).expand(), x, -infinity, infinity).n()
-6.26376265908397e-17
sage: integrate( (F(x)-G(x))^2, x, -infinity, infinity).n()# abstol 1e-6
0
```

This was broken before Maxima 5.20:

```
sage: exp(-x*i).integral(x,0,1)
I*e^(-I) - I
```

Test deprecation warning when variable is not specified:

```
sage: x.integral()
doctest:...: DeprecationWarning:
Variable of integration should be specified explicitly.
See http://trac.sagemath.org/12438 for details.
1/2*x^2
```

Test that trac ticket #8729 is fixed:

```
sage: t = var('t')
sage: a = sqrt((sin(t))^2 + (cos(t))^2)
sage: integrate(a, t, 0, 2*pi)
2*pi
sage: a.simplify_full().simplify_trig()
1
```

Maxima uses Cauchy Principal Value calculations to integrate certain convergent integrals. Here we test that this does not raise an error message (see trac ticket #11987):

```
sage: integrate(sin(x)*sin(x/3)/x^2, x, 0, oo)
1/6*pi
```

Maxima returned a negative value for this integral prior to maxima-5.24 (trac ticket #10923). Ideally we would get an answer in terms of the gamma function; however, we get something equivalent:

```
sage: actual_result = integral(e^(-1/x^2), x, 0, 1)
sage: actual_result.canonicalize_radical()
(sqrt(pi)*(erf(1)*e - e) + 1)*e^(-1)
sage: ideal_result = 1/2*gamma(-1/2, 1)
sage: error = actual_result - ideal_result
sage: error.numerical_approx() # abs tol 1e-10
0
```

We will not get an evaluated answer here, which is better than the previous (wrong) answer of zero. See trac ticket #10914:

```
sage: f = abs(sin(x))
sage: integrate(f, x, 0, 2*pi)  # long time (4s on sage.math, 2012)
integrate(abs(sin(x)), x, 0, 2*pi)
```

Another incorrect integral fixed upstream in Maxima, from trac ticket #11233:

```
sage: a,t = var('a,t')
sage: assume(a>0)
sage: assume(x>0)
sage: f = log(1 + a/(x * t)^2)
sage: F = integrate(f, t, 1, Infinity)
sage: F(x=1, a=7).numerical_approx() # abs tol 1e-10
4.32025625668262
sage: forget()
```

Verify that MinusInfinity works with sympy (trac ticket #12345):

```
sage: integral(1/x^2, x, -infinity, -1, algorithm='sympy')
1
```

Check that trac ticket #11737 is fixed:

```
sage: N(integrate(sin(x^2)/(x^2), x, 1, infinity))
0.285736646322853
```

Check that trac ticket #14209 is fixed:

```
sage: integral(e^(-abs(x))/cosh(x),x,-infinity,infinity)
2*log(2)
sage: integral(e^(-abs(x))/cosh(x),x,-infinity,infinity)
2*log(2)
```

Check that trac ticket #12628 is fixed:

```
sage: var('z,n')
(z, n)
sage: f(z, n) = sin(n*z) / (n*z)
sage: integrate(f(z,1)*f(z,3)*f(z,5)*f(z,7),z,0,oo)
22/315*pi
sage: for k in srange(1, 16, 2):
....:     print integrate(prod(f(z, ell)
....:                     for ell in srange(1, k+1, 2)), z, 0, oo)
```

```
1/2*pi
1/6*pi
1/10*pi
22/315*pi
3677/72576*pi
48481/1247400*pi
193359161/6227020800*pi
5799919/227026800*pi
```

Check that trac ticket #12628 is fixed:

```
sage: integrate(1/(sqrt(x)*((1+sqrt(x))^2)),x,1,9)
1/2
```

CHAPTER

TWELVE

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
```

```
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
```

```
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
```

```
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)
```

```
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
```

```
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
[x*y + 1     2*x]
[    2*y x*y + 1]
sage: R.<x,y> = QQ[]
sage: A = matrix([[1,x], [y,1]])
sage: print 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^
[    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 + 
sage: var('x y')
(x, y)
```

And here are some actual tests of sympy:

```
sage: from sympy import Symbol, cos, sympify, pprint
sage: from sympy.abc import x
```

```
sage: e = sympify(1)/cos(x)**3; e
cos(x)**(-3)
sage: f = e.series(x, 0, 10); f
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 aren't working on some archictures, we disable it:

```
sage: from sympy.printing import pprint_use_unicode
sage: prev_use = pprint_use_unicode(False)
sage: pprint(e)
   1
-------
```

```
   3
cos (x)

sage: pprint(f)
       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)
False
```

And the functionality to convert from sympy format to Sage format:

```
sage: e._sage_()
cos(x)^(-3)
sage: e._sage_().taylor(x._sage_(), 0, 8)
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + 1
sage: f._sage_()
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + 1
```

Mixing SymPy with Sage:

```
sage: import sympy
sage: sympy.sympify(var("y"))+sympy.Symbol("x")
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)
<type 'sage.symbolic.expression.Expression'>
sage: t1, t2
(omega + x, omega + x)
sage: e=sympy.sin(var("y"))+sage.all.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)
<type 'sage.symbolic.expression.Expression'>
sage: e
cos(x) + sin(y)
sage: e = sage.all.cos(var("y")**3)**4+var("x")**2
sage: e = e._sympy_()
sage: e
x**2 + cos(y**3)**4

sage: a = sympy.Matrix([1, 2, 3])
sage: a[1]
2

sage: sympify(1.5)
1.50000000000000
sage: sympify(2)
2
```

```
sage: sympify(-2)
-2
```

TESTS:

This was fixed in Sympy, see trac ticket #14437:

```
sage: from sympy import Function, Symbol, rsolve
sage: u = Function('u')
sage: n = Symbol('n', integer=True)
sage: f = u(n+2) - u(n+1) + u(n)/4
sage: rsolve(f,u(n))
2**(-n)*(C0 + C1*n)
```

CHAPTER

THIRTEEN

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: print 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)
...     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
```

```
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)
```

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
```

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
```

A big product. Notice that simplifying simplifies the product further:

```
sage: A = exp(I*pi/5)
sage: b = A*A*A*A*A*A*A*A*A*A
sage: 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
```

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,)
```

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)
D[0](f)(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)
sage: integrate(log(1-x^2)/x, x)
1/2*log(x^2)*log(-x^2 + 1) + 1/2*polylog(2, -x^2 + 1)
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
```

```
sage: integrate((1-x^2)^n,x)
integrate((-x^2 + 1)^n, x)
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)
```

```
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()
```

The following are a bunch of examples of integrals that Mathematica can do, but Sage currently can't do:

```
sage: integrate(log(x)*exp(-x^2), x)        # todo -- Mathematica can do this
integrate(e^(-x^2)*log(x), x)
```

Todo - Mathematica can do this and gets $\pi^2/15$.

```
sage: integrate(log(1+sqrt(1+4*x)/2)/x, x, 0, 1)  # not tested
Traceback (most recent call last):
...
ValueError: Integral is divergent.
```

```
sage: integrate(ceil(x^2 + floor(x)), x, 0, 5)    # todo: Mathematica can do this
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)
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)
D[0](f)(x)
sage: diff(f(x,y), x, y)
D[0, 1](f)(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)
D[0, 2, 2](g)(x, y, z)
sage: integrate(sin(x), x)
```

```
-cos(x)
sage: integrate(sin(x), x, 0, pi)
2

sage: var('a b')
(a, b)
sage: 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)       # todo: maple can compute this exactly.
integrate(e^(-x^2)*log(x), x)
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)     # todo: maple can do this
integrate(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
sage: integral(integral(x*y^2, x, 0, y), y, -2, 2)
32/5
```

We verify several standard differentiation rules:

```
sage: function('f, g')
(f, g)
sage: diff(f(t)*g(t),t)
g(t)*D[0](f)(t) + f(t)*D[0](g)(t)
sage: diff(f(t)/g(t), t)
D[0](f)(t)/g(t) - f(t)*D[0](g)(t)/g(t)^2
sage: diff(f(t) + g(t), t)
D[0](f)(t) + D[0](g)(t)
sage: diff(c*f(t), t)
c*D[0](f)(t)
```

CHAPTER

FOURTEEN

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.`**`AlgebraicConverter`**(*field*)

Bases: `sage.symbolic.expression_conversions.Converter`

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import AlgebraicConverter
sage: a = AlgebraicConverter(QQbar)
sage: a.field
Algebraic Field
sage: a.reciprocal_trig_functions['cot']
tan
```

`arithmetic`(*ex*, *operator*)

Convert a symbolic expression to an algebraic number.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import AlgebraicConverter
sage: f = 2^(1/2)
sage: a = AlgebraicConverter(QQbar)
sage: a.arithmetic(f, f.operator())
1.414213562373095?
```

TESTS:

```
sage: f = pi^6
sage: a = AlgebraicConverter(QQbar)
sage: a.arithmetic(f, f.operator())
Traceback (most recent call last):
...
TypeError: unable to convert pi^6 to Algebraic Field
```

`composition`(*ex*, *operator*)

Coerce to an algebraic number.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import AlgebraicConverter
sage: a = AlgebraicConverter(QQbar)
sage: a.composition(exp(I*pi/3), exp)
0.500000000000000? + 0.866025403784439?*I
sage: a.composition(sin(pi/5), sin)
0.5877852522924731? + 0.?e-18*I
```

TESTS:

```
sage: QQbar(zeta(7))
Traceback (most recent call last):
...
TypeError: unable to convert zeta(7) to Algebraic Field
```

pyobject (*ex*, *obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import AlgebraicConverter
sage: a = AlgebraicConverter(QQbar)
sage: f = SR(2)
sage: a.pyobject(f, f.pyobject())
2
sage: _.parent()
Algebraic Field
```

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.div.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: c = Converter(use_fake_div=True)
sage: 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.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: f = x + 2
sage: Converter().arithmetic(f, f.operator())
Traceback (most recent call last):
...
NotImplementedError: arithmetic
```

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.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: f = sin(2)
sage: Converter().composition(f, f.operator())
Traceback (most recent call last):
...
NotImplementedError: composition
```

derivative (*ex*, *operator*)

The input to this method is a symbolic expression which corresponds to a relation.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: a = function('f', x).diff(x); a
D[0](f)(x)
sage: Converter().derivative(a, a.operator())
Traceback (most recent call last):
...
NotImplementedError: derivative
```

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 div>)
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>)], <
```

Check if #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 div
```

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.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: f = SR(1)
sage: Converter().pyobject(f, f.pyobject())
Traceback (most recent call last):
...
NotImplementedError: pyobject
```

relation(*ex*, *operator*)

The input to this method is a symbolic expression which corresponds to a relation.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: import operator
sage: Converter().relation(x==3, operator.eq)
Traceback (most recent call last):
...
NotImplementedError: relation
sage: Converter().relation(x==3, operator.lt)
Traceback (most recent call last):
...
NotImplementedError: 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.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: Converter().symbol(x)
Traceback (most recent call last):
...
NotImplementedError: symbol
```

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.div)
sage: f.operands()
[x, y]
```

operator()

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import FakeExpression
sage: import operator; x,y = var('x,y')
sage: f = FakeExpression([x, y], operator.div)
sage: f.operator()
<built-in function div>
```

pyobject()

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import FakeExpression
sage: import operator; x,y = var('x,y')
sage: f = FakeExpression([x, y], operator.div)
sage: f.pyobject()
Traceback (most recent call last):
...
TypeError: self must be a numeric expression
```

class sage.symbolic.expression_conversions.**FastCallableConverter**(*ex*, *etb*)

Bases: sage.symbolic.expression_conversions.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
```

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)
```

TESTS:

Check if rational functions with numerator 1 can be converted. #8056:

```
sage: (1/pi/x)._fast_callable_(etb)
div(1, mul(pi, v_0))

sage: etb = ExpressionTreeBuilder(vars=['x'], domain=RDF)
sage: (x^7)._fast_callable_(etb)
ipow(v_0, 7)
sage: f(x)=1/pi/x; plot(f,2,3)
Graphics object consisting of 1 graphics primitive
```

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)
```

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
```

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
```

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
```

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]
```

class sage.symbolic.expression_conversions.**FastFloatConverter**(*ex*, **vars*)

Bases: sage.symbolic.expression_conversions.Converter

Returns an object which provides fast floating point evaluation of the symbolic expression *ex*. This is an class used internally and is not meant to be used directly.

See sage.ext.fast_eval for more information.

EXAMPLES:

```
sage: x,y,z = var('x,y,z')
sage: f = 1 + sin(x)/x + sqrt(z^2+y^2)/cosh(x)
sage: ff = f._fast_float_('x', 'y', 'z')
sage: f(x=1.0,y=2.0,z=3.0).n()
4.1780638977...
sage: ff(1.0,2.0,3.0)
4.1780638977...
```

Using _fast_float_ without specifying the variable names is deprecated:

```
sage: f = x._fast_float_()
doctest:...: DeprecationWarning: Substitution using
function-call syntax and unnamed arguments is deprecated
and will be removed from a future release of Sage; you
```

```
can use named arguments instead, like EXPR(x=..., y=...)
See http://trac.sagemath.org/5930 for details.
sage: f(1.2)
1.2
```

Using _fast_float_ on a function which is the identity is now supported (see Trac 10246):

```
sage: f = symbolic_expression(x).function(x)
sage: f._fast_float_(x)
<sage.ext.fast_eval.FastDoubleFunc object at ...>
sage: f(22)
22
```

arithmetic(*ex*, *operator*)

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = x*x-y
sage: ff = f._fast_float_('x','y')
sage: ff(2,3)
1.0

sage: a = x + 2*y
sage: f = a._fast_float_('x', 'y')
sage: f(1,0)
1.0
sage: f(0,1)
2.0

sage: f = sqrt(x)._fast_float_('x'); f.op_list()
['load 0', 'call sqrt(1)']

sage: f = (1/2*x)._fast_float_('x'); f.op_list()
['load 0', 'push 0.5', 'mul']
```

composition(*ex*, *operator*)

EXAMPLES:

```
sage: f = sqrt(x)._fast_float_('x')
sage: f(2)
1.41421356237309...
sage: y = var('y')
sage: f = sqrt(x+y)._fast_float_('x', 'y')
sage: f(1,1)
1.41421356237309...

sage: f = sqrt(x+2*y)._fast_float_('x', 'y')
sage: f(2,0)
1.41421356237309...
sage: f(0,1)
1.41421356237309...
```

pyobject(*ex*, *obj*)

EXAMPLES:

```
sage: f = SR(2)._fast_float_()
sage: f(3)
2.0
```

relation(*ex*, *operator*)

EXAMPLES:

```
sage: ff = fast_float(x == 2, 'x')
sage: ff(2)
0.0
sage: ff(4)
2.0
sage: ff = fast_float(x < 2, 'x')
Traceback (most recent call last):
...
NotImplementedError
```

symbol (*ex*)

EXAMPLES:

```
sage: f = x._fast_float_('x', 'y')
sage: f(1,2)
1.0
sage: f = x._fast_float_('y', 'x')
sage: f(1,2)
2.0
```

class sage.symbolic.expression_conversions.**InterfaceInit** (*interface*)

Bases: sage.symbolic.expression_conversions.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)'
```

arithmetic (*ex*, *operator*)

EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.arithmetic(x+2, operator.add)
'(_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)
'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]'
```

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
D[0](f)(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)
```

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
D[0](f)(x*y)
sage: m.derivative(t, t.operator())
"at(diff('f(_SAGE_VAR_t0), _SAGE_VAR_t0, 1), [_SAGE_VAR_t0 = (_SAGE_VAR_x)*(_SAGE_VAR_y)])"
```

TESTS:

Most of these confirm that trac ticket #7401 was fixed:

```
sage: t = var('t'); f = function('f')(t)
sage: a = 2^e^t * f.subs(t=e^t) * diff(f, t).subs(t=e^t) + 2*t
sage: solve(a == 0, diff(f, t).subs(t=e^t))
[D[0](f)(e^t) == -2^(-e^t + 1)*t/f(e^t)]

sage: f = function('f', x)
sage: df = f.diff(x); df
D[0](f)(x)
sage: maxima(df)
'diff('f(_SAGE_VAR_x),_SAGE_VAR_x,1)

sage: a = df.subs(x=exp(x)); a
D[0](f)(e^x)
sage: b = maxima(a); b
%at('diff('f(_SAGE_VAR_t0),_SAGE_VAR_t0,1),[_SAGE_VAR_t0=%e^_SAGE_VAR_x])
sage: bool(b.sage() == a)
True

sage: a = df.subs(x=4); a
D[0](f)(4)
sage: b = maxima(a); b
%at('diff('f(_SAGE_VAR_t0),_SAGE_VAR_t0,1),[_SAGE_VAR_t0=4])
sage: bool(b.sage() == a)
True
```

It also works with more than one variable. Note the preferred syntax `function('f')(x, y)` to create a general symbolic function of more than one variable:

```
sage: x, y = var('x y')
sage: f = function('f')(x, y)
sage: f_x = f.diff(x); f_x
D[0](f)(x, y)
sage: maxima(f_x)
'diff('f(_SAGE_VAR_x,_SAGE_VAR_y),_SAGE_VAR_x,1)
```

```
sage: a = f_x.subs(x=4); a
D[0](f)(4, y)
sage: b = maxima(a); b
%at('diff('f(_SAGE_VAR_t0,_SAGE_VAR_t1),_SAGE_VAR_t0,1),[_SAGE_VAR_t0=4,_SAGE_VAR_t1=_SAGE_V
sage: bool(b.sage() == a)
True

sage: a = f_x.subs(x=4).subs(y=8); a
D[0](f)(4, 8)
sage: b = maxima(a); b
%at('diff('f(_SAGE_VAR_t0,_SAGE_VAR_t1),_SAGE_VAR_t0,1),[_SAGE_VAR_t0=4,_SAGE_VAR_t1=8])
sage: bool(b.sage() == a)
True
```

pyobject (*ex*, *obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: ii = InterfaceInit(gp)
sage: f = 2+I
sage: ii.pyobject(f, f.pyobject())
'I + 2'

sage: ii.pyobject(SR(2), 2)
'2'

sage: 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'
```

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'
```

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)]'
```

class sage.symbolic.expression_conversions.**PolynomialConverter**(*ex*, *base_ring=None*, *ring=None*)

Bases: sage.symbolic.expression_conversions.Converter

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
```

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
```

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
```

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
```

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
```

symbol (*ex*)

Returns 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
```

class sage.symbolic.expression_conversions.**RingConverter** (*R*, *subs_dict=None*)

Bases: `sage.symbolic.expression_conversions.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...>
```

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
```

composition(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF)
sage: R(cos(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?
```

symbol(*ex*)

All symbols appearing in the expression must appear in *subs_dict* 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
```

class sage.symbolic.expression_conversions.**SubstituteFunction**(*ex*, *original*, *new*)

Bases: `sage.symbolic.expression_conversions.Converter`

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)
```

arithmetic(*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 = x*foo(x) + pi/foo(x)
sage: s.arithmetic(f, f.operator())
x*bar(x) + pi/bar(x)
```

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)
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))
```

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())
D[0](bar)(x)
```

TESTS:

We can substitute functions under a derivative operator, trac ticket #12801:

```
sage: f = function('f')
sage: g = function('g')
sage: f(g(x)).diff(x).substitute_function(g, sin)
cos(x)*D[0](f)(sin(x))
```

pyobject(*ex*, *obj*)

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 = SR(2)
sage: s.pyobject(f, f.pyobject())
2
sage: _.parent()
Symbolic Ring
```

relation (*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: eq = foo(x) == x
sage: s.relation(eq, eq.operator())
bar(x) == x
```

symbol (*ex*)

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.symbol(x)
x
```

class sage.symbolic.expression_conversions.**SympyConverter** (*use_fake_div=False*)

Bases: sage.symbolic.expression_conversions.Converter

Converts any expression to SymPy.

EXAMPLE:

```
sage: import sympy
sage: var('x,y')
(x, y)
sage: f = exp(x^2) - arcsin(pi+x)/y
sage: f._sympy_()
exp(x**2) - asin(x + pi)/y
sage: _._sage_()
-arcsin(pi + x)/y + e^(x^2)

sage: sympy.sympify(x) # indirect doctest
x
```

TESTS:

Make sure we can convert I (trac #6424):

```
sage: bool(I._sympy_() == I)
True
sage: (x+I)._sympy_()
x + I
```

arithmetic (*ex, operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SympyConverter
sage: s = SympyConverter()
sage: f = x + 2
sage: s.arithmetic(f, f.operator())
x + 2
```

composition (*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SympyConverter
sage: s = SympyConverter()
sage: f = sin(2)
sage: s.composition(f, f.operator())
sin(2)
sage: type(_)
sin
sage: f = arcsin(2)
sage: s.composition(f, f.operator())
asin(2)
```

pyobject (*ex*, *obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SympyConverter
sage: s = SympyConverter()
sage: f = SR(2)
sage: s.pyobject(f, f.pyobject())
2
sage: type(_)
<class 'sympy.core.numbers.Integer'>
```

symbol (*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SympyConverter
sage: s = SympyConverter()
sage: s.symbol(x)
x
sage: type(_)
<class 'sympy.core.symbol.Symbol'>
```

sage.symbolic.expression_conversions.**algebraic** (*ex*, *field*)

Returns the symbolic expression *ex* as a element of the algebraic field *field*.

EXAMPLES:

```
sage: a = SR(5/6)
sage: AA(a)
5/6
sage: type(AA(a))
<class 'sage.rings.qqbar.AlgebraicReal'>
sage: QQbar(a)
5/6
sage: type(QQbar(a))
<class 'sage.rings.qqbar.AlgebraicNumber'>
sage: QQbar(i)
1*I
sage: AA(golden_ratio)
1.618033988749895?
sage: QQbar(golden_ratio)
```

```
1.618033988749895?
sage: QQbar(sin(pi/3))
0.866025403784439?

sage: QQbar(sqrt(2) + sqrt(8))
4.242640687119285?
sage: AA(sqrt(2) ^ 4) == 4
True
sage: AA(-golden_ratio)
-1.618033988749895?
sage: QQbar((2*I)^(1/2))
1 + 1*I
sage: QQbar(e^(pi*I/3))
0.500000000000000? + 0.866025403784439?*I

sage: AA(x*sin(0))
0
sage: QQbar(x*sin(0))
0
```

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)))
```

sage.symbolic.expression_conversions.**fast_float**(*ex*, **vars*)

Returns an object which provides fast floating point evaluation of the symbolic expression *ex*.

See `sage.ext.fast_eval` for more information.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import fast_float
sage: f = sqrt(x+1)
sage: ff = fast_float(f, 'x')
sage: ff(1.0)
1.4142135623730951
```

sage.symbolic.expression_conversions.**polynomial**(*ex*, *base_ring=None*, *ring=None*)

Returns a polynomial from the symbolic expression *ex*. Either a base ring *base_ring* or a polynomial ring *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*.

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()
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]
```

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
```

CHAPTER

FIFTEEN

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*`)`

Returns 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
```

CHAPTER

SIXTEEN

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))
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

sage: # (YES) Evaluate  e^(Pi*Sqrt(163)) to 50 decimal digits
sage: a = e^(pi*sqrt(163)); a
e^(sqrt(163)*pi)
sage: print RealField(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

sage: # (YES) Obtain period of decimal fraction 1/7=0.(142857).
sage: a = 1/7
sage: print a
1/7
sage: print 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]

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: print float(a-b)
0.0
sage: print 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
```

```
sage: # (NOT REALLY) 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)
False
sage: abs(float(a-b)) < 1e-10
True
sage: # 2*Infinity-3=Infinity.
sage: 2*infinity-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
```

```
sage: # (NO) Hypothesis testing with t-distribution.
sage: # (NO) Hypothesis testing with chi^2 distribution
sage: # (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[]
sage: (x^2-4)/(x^2+4*x+4) == (x-2)/(x+2)
True
sage: 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
```

```
sage: # (YES) Expand (1+x)^20, take derivative and factorize.
sage: # first do it using algebraic polys
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^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 +
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^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 +
sage: 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 -
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 +
sage: 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: 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)
```

```
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
```

```
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
```

```
sage: # (YES) Assuming  x>=y,  y>=z,  z>=x, deduce  x=z.
sage: forget()
sage: var('x,y,z')
(x, y, z)
sage: assume(x>=y, y>=z,z>=x)
sage: print 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: print list(sorted(assumptions()))
[x > y, y > 0]
sage: print bool(2*x^2 > 2*y^2)
True
sage: forget()
sage: print 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: print eqn
                                abs(x - 1) > 2
```

```
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
(x - 1)*(x - 2)*(x - 3)*(x - 4)*(x - 5) < 0
```

```
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: print 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: print 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.
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
```

```
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

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: print 13*7691
99983

sage: # (YES) (2^(1/3) + 4^(1/3))^3 - 6*(2^(1/3) + 4^(1/3))-6 = 0
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
sage: ## 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: print 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
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)

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))
sage: 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]

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, as
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)) - 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]

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)

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()
tan(real_part(z))/(tan(imag_part(z))^2*tan(real_part(z))^2 + 1)
sage: a.imag()
tanh(imag_part(z))/(tan(imag_part(z))^2*tan(real_part(z))^2 + 1)

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()

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 Trac #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()

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

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]

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]

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]]

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: print 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: print 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)

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: print 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: print 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
sage: print d.factor()
(-1) * (c - d) * (-b + c) * (b - d) * (-a + c) * (-a + b) * (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])
]

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

sage: # (OK-ish) D(x)Abs(x)
sage: #    Verify(D(x) Abs(x), Sign(x));
sage: diff(abs(x))
x/abs(x)

sage: # (YES) (Integrate(x)Abs(x))=Abs(x)*x/2
sage: 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

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
```

```
sage: # (OK-ish) (Taylor expansion of Sin(x))/(Taylor expansion of Cos(x)) = (Taylor expansion of Ta
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)
```

```
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)*
```

```
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
```

```
sage: # (NO) Compute n-th term of the Taylor series of Ln(Sin(x)/x) at x=0.
sage: # 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
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: print 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^1
```

```
sage: # (OK) Compute Legendre polynomials directly from Rodrigues's formula, P[n]=1/(2^n*n!) *(Deriv
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
```

```
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

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: # We use the trick of evaluating the algebraic poly at a symbolic variable:
sage: restore('x')
sage: p(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));

sage: # (YES) Verify that True And False=False.
sage: (True and False) == False
True

sage: # (YES) Prove x Or Not x.
sage: 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!"
```

CHAPTER SEVENTEEN

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 IVP for one first order equation, return list of points or plot.
- `desolve_system_rk4` - Solve numerically IVP for system of first order equations, return list of points.
- `desolve_odeint` - Solve numerically a system of first-order ordinary differential equations using `odeint` from scipy.integrate module.
- `desolve_system` - Solve any size system of 1st order odes 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.

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*)

Solves a 1st or 2nd order linear ODE via maxima. Including IVP and BVP.

Use `desolve?  <tab>` *if the output in truncated in notebook.*

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 for the example of the 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)

OUTPUT:

In most cases return a SymbolicEquation which defines the solution implicitly. If the result is in the form y(x)=... (happens for linear eqs.), return the right-hand side only. The possible constant solutions of separable ODE's are omitted.

EXAMPLES:

```
sage: x = var('x')
sage: y = function('y', x)
sage: desolve(diff(y,x) + y - 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)

sage: 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

sage: f = desolve(de, y, [10,2,1]); f
-x + 7*e^(x - 10) + 5*e^(-x + 10)

sage: f(x=10)
2

sage: diff(f,x)(x=10)
1

sage: de = diff(y,x,2) + y == 0
sage: desolve(de, y)
_K2*cos(x) + _K1*sin(x)
```

```
sage: desolve(de, y, [0,1,pi/2,4])
cos(x) + 4*sin(x)

sage: desolve(y*diff(y,x)+sin(x)==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], 'clairault']
```

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]]

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']
```

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

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

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']
```

Separable equations - Sage returns solution in implicit form:

```
sage: 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']

sage: desolve(diff(y,x)*sin(y) == cos(x),y,[pi/2,1])
-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)

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']

sage: desolve(diff(y,x)+(y) == cos(x),y,[0,1])
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']
```

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)
```

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
```

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
```

Some more types of ODE's:

```
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*e^x, y(x) == _C + log(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 + sq
```

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

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
```

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)

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']

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)

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']

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)

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']
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 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']

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,[0,3,1])
(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']

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)

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']
```

TESTS:

trac ticket #9961 fixed (allow assumptions on the dependent variable in desolve):

```
sage: y=function('y',x); assume(x>0); assume(y>0)
sage: sage.calculus.calculus.maxima('domain:real')  # needed since Maxima 5.26.0 to get the answ
real
sage: desolve(x*diff(y,x)-x*sqrt(y^2+x^2)-y == 0, y, contrib_ode=True)
[x - arcsinh(y(x)/x) == _C]
```

trac ticket #10682 updated Maxima to 5.26, and it started to show a different solution in the complex domain for the ODE above:

```
sage: sage.calculus.calculus.maxima('domain:complex')  # back to the default complex domain
complex
sage: desolve(x*diff(y,x)-x*sqrt(y^2+x^2)-y == 0, y, contrib_ode=True)
[1/2*(2*x^2*sqrt(x^(-2)) - 2*x*sqrt(x^(-2))*arcsinh(y(x)/sqrt(x^2)) -
    2*x*sqrt(x^(-2))*arcsinh(y(x)^2/(x*sqrt(y(x)^2))) +
    log(4*(2*x^2*sqrt((x^2*y(x)^2 + y(x)^4)/x^2)*sqrt(x^(-2)) + x^2 +
    2*y(x)^2)/x^2))/(x*sqrt(x^(-2))) == _C]
```

trac ticket #6479 fixed:

```
sage: x = var('x')
sage: y = function('y', x)
sage: desolve( diff(y,x,x) == 0, y, [0,0,1])
x

sage: desolve( diff(y,x,x) == 0, y, [0,1,1])
x + 1
```

trac ticket #9835 fixed:

```
sage: x = var('x')
sage: y = function('y', x)
sage: desolve(diff(y,x,2)+y*(1-y^2)==0,y,[0,-1,1,1])
Traceback (most recent call last):
...
NotImplementedError: Unable to use initial condition for this equation (freeofx).
```

trac ticket #8931 fixed:

```
sage: x=var('x'); f=function('f',x); k=var('k'); assume(k>0)
sage: desolve(diff(f,x,2)/f==k,f,ivar=x)
_K1*e^(sqrt(k)*x) + _K2*e^(-sqrt(k)*x)
```

trac ticket #15775 fixed:

```
sage: forget()
sage: y = function('y')(x)
sage: desolve(diff(y, x) == sqrt(abs(y)), dvar=y, ivar=x)
sqrt(-y(x))*(sgn(y(x)) - 1) + (sgn(y(x)) + 1)*sqrt(y(x)) == _C + 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 (eg, de = diff(y,x,2) == diff(y,x)+sin(x))
- `dvar` - the dependent variable (eg 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` - a list of numbers representing initial conditions, (eg, f(0)=1, f'(0)=2 is 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)
```

We can use initial conditions:

```
sage: desolve_laplace(eq,u,ics=[0,3])
-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)

sage: f=function('f', x)
sage: eq = diff(f,x) + f == 0
sage: desolve_laplace(eq,f,[0,1])
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)
```

```
sage: desolve_laplace(de,f,ics=[0,1,2])
x*e^x + e^x
```

TESTS:

Trac #4839 fixed:

```
sage: t=var('t')
sage: x=function('x', t)
sage: soln=desolve_laplace(diff(x,t)+x==1, x, ics=[0,2])
sage: soln
e^(-t) + 1

sage: soln(t=3)
e^(-3) + 1
```

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 deppendent variables.
- `ics` – a 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:

- A 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]]
```

ALGORITHM:

Uses TIDES.

REFERENCES:

- A. Abad, R. Barrio, F. Blesa, M. Rodriguez. Algorithm 924. *ACM Transactions on Mathematical Software* , *39* (1), 1-28.
- (http://www.unizar.es/acz/05Publicaciones/Monografias/MonografiasPublicadas/Monografia36/IndMonogr36.htm) 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 `odeint` from scipy.integrate module.

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. If True, the Jacobian of des is computed and used during the integration of Stiff Systems. Default value is False.

Other Parameters (taken from the documentation of odeint function from scipy.integrate module)

- `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:

Return 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])
sage: p=line(zip(sol[:,0],sol[:,1]))
sage: p.show()
```

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],rtol=1e-13,atol=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,rtol=1e-9,atol=1e-10,compute_jac=True)
sage: p=points(zip(t,sol))
sage: p.show()
```

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,h0=0.1,hmax=1,hmin=1e-4,mxstep=1000,mxords
```

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. See also `ode_solver`.

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 funciton 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 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` - (optional, default:0.1) the length of the step (positive number)
  - `output` - (optional, 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.

EXAMPLES:

```
sage: 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, 1.461590162288825]]
```

Variant 1 for input - we can pass ODE in the form used by desolve function In this example we integrate bakwards, 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, 1.909327945361535], [1, 1]]
```

Here we show how to plot simple pictures. For more advanced aplications 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
```

ALGORITHM:

4th 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)

sage: desolve_rk4_determine_bounds([0,2])
(0, 10)

sage: desolve_rk4_determine_bounds([0,2],[-2])
(-2, 0)

sage: desolve_rk4_determine_bounds([0,2],[-2,4])
(-2, 4)
```

`sage.calculus.desolvers.`**`desolve_system`**(*des*, *vars*, *ics=None*, *ivar=None*)

Solve any size system of 1st order ODE's. Initial conditions are optional.

Onedimensional 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.

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]
```

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]

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
```

TESTS:

Check that trac ticket #9823 is fixed:

```
sage: t = var('t')
sage: x = function('x', t)
sage: de1 = diff(x,t) + 1 == 0
sage: desolve_system([de1], [x])
-t + x(0)
```

Check that trac ticket #16568 is fixed:

```
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: des = [de1,de2]
sage: ics = [0,1,-1]
sage: vars = [x,y]
sage: sol = desolve_system(des, vars, ics); sol
[x(t) == 2*sin(t) + 1, y(t) == -2*cos(t) + 1]

sage: solx, soly = sol[0].rhs(), sol[1].rhs()
sage: RR(solx(t=3))
1.28224001611973

sage: P1 = plot([solx,soly], (0,1))
sage: P2 = parametric_plot((solx,soly), (0,1))
```

Now type show(P1), show(P2) to view these plots.

Check that trac ticket #9824 is fixed:

```
sage: t = var('t')
sage: epsilon = var('epsilon')
sage: x1 = function('x1', t)
sage: x2 = function('x2', t)
sage: de1 = diff(x1,t) == epsilon
sage: de2 = diff(x2,t) == -2
sage: desolve_system([de1, de2], [x1, x2], ivar=t)
[x1(t) == epsilon*t + x1(0), x2(t) == -2*t + x2(0)]
sage: desolve_system([de1, de2], [x1, x2], ics=[1,1], ivar=t)
Traceback (most recent call last):
...
ValueError: Initial conditions aren't complete: number of vars is different from number of depen
```

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 4th order Runge-Kutta method. Wrapper for Maxima command `rk`. See also `ode_solver`.

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` – (optional, default: 0.1) the length of the step

OUTPUT:

Return a list of points.

EXAMPLES:

```
sage: 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)

sage: Q=[ [j,k] for i,j,k in P]
sage: LP=list_plot(Q)
```

ALGORITHM:

4th 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 precission implemented in tides.

INPUT:

- `f` – symbolic function. Its first argument will be the independent variable. Its output should be de derivatives of the deppendent variables.
- `ics` – a 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 precission used in the computation.

OUTPUT:

- A list with the positions of the IVP.

EXAMPLES:

We integrate the Lorenz equations with Salztman values for the parameters along 10 periodic orbits with 100 digits of precission:

```
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 = -13.7636106821342005250144010543616538641008648540923684535378642921202827747268115852940239346
sage: y0 = -19.578751942451795538838041446009558866114240053427643864979133429542635474614752641
sage: z0 = 27
sage: T = 15.586522107161747275678702092126960705284805489972439358895215783190198756258880854355
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
[[0.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
-13.76361068213420052501440105436165386410086485409236845353786429212028277472681158529402393463
-19.57875194245179553883804144600955886611424005342764386497913342954263547461475264159731655067
27.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
[15.58652210716174727567870209212696070528480548997243935889521578319019875625888085435585108266
-13.76361068213420052501440105436165386410086485409236845353786429212028277472681158529402393463
-19.57875194245179553883804144600955886611424005342764386497913342954263547461475264159731655067
26.999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999963
```

ALGORITHM:

Uses TIDES.

Warning: This requires the package tides.

REFERENCES:

`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 `x0` to `x1` by `h` (so `(x1-x0)/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.

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
  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]]

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]]

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

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

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]]

sage: pts = eulers_method(5*x+y-5,0,1,1/2,1,algorithm="none")
sage: P1 = list_plot(pts)
sage: P2 = line(pts)
sage: (P1+P2).show()
```

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(t0)=x0.`

`y' = g(t, x, y), y(t0)=y0.`

The "t" column of the table increments from t_0 to t_1 by h (so $fract_1 - t_0h$ 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.

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]]

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

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
```

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: $y_1'=y_2$, $y_1(0)=1$; $y_2'=$ $fracy_1-y_21+t^2$, $y_2(0)=-1$.:

```
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                 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
```

AUTHORS:

- David Joyner

`sage.calculus.desolvers.`**`eulers_method_2x2_plot`**(*f, g, t0, x0, y0, h, t1*)

Plot solution of ODE.

This plots the soln in the rectangle `(xrange[0],xrange[1]) x (yrange[0],yrange[1])` and plots using Euler's method the numerical solution of the 1st order ODEs $x'=f(t,x,y)$, $x(a)=x_0$, $y'=g(t,x,y)$, $y(a)=y_0$.

For pedagogical purposes only.

EXAMPLES:

```
sage: from sage.calculus.desolvers import eulers_method_2x2_plot
```

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: 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)
```

CHAPTER

EIGHTEEN

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.gsl.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)
sage: a.forward_transform()
sage: a.plot().show()
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)
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)
```

This example gives a simple example of wavelet compression:

```
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)
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)
```

class `sage.gsl.dwt.`**`DiscreteWaveletTransform`**

Bases: `sage.gsl.gsl_array.GSLDoubleArray`

Discrete wavelet transform class.

`backward_transform`()

`forward_transform`()

`plot`(*xmin=None*, *xmax=None*, ***args*)

`sage.gsl.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)
sage: a.forward_transform()
sage: a.plot().show()
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)
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)
```

This example gives a simple example of wavelet compression:

```
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)
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)
```

sage.gsl.dwt.**is2pow**(*n*)

CHAPTER

NINETEEN

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.gsl.dft.IndexedSequence` (*L*, *index_object*)

Bases: `sage.structure.sage_object.SageObject`

An indexed sequence.

INPUT:

- `L` – A 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 = range(10)
sage: A = [1/10 for j in J]
sage: s = IndexedSequence(A,J)
sage: 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 = 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]
```

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 = 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]
```

AUTHOR: David Joyner (2006-09)

dct()

A discrete Cosine transform.

EXAMPLES:

```
sage: J = range(5)
sage: A = [exp(-2*pi*i*I/5) for i in J]
sage: s = IndexedSequence(A,J)
sage: s.dct()

Indexed sequence: [1/4*(sqrt(5) - 1)*e^(-2/5*I*pi) - 1/4*(sqrt(5) + 1)*e^(-4/5*I*pi) - 1/4*(
indexed by [0, 1, 2, 3, 4]
```

dft(*chi=<function <lambda> at 0x7f539d81e410>*)

A discrete Fourier transform "over $\mathbf{Q}$" using exact N-th roots of unity.

EXAMPLES:

```
sage: J = range(6)
sage: A = [ZZ(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: s.dft(lambda x:x^2)
Indexed sequence: [6, 0, 0, 6, 0, 0]
 indexed by [0, 1, 2, 3, 4, 5]
sage: s.dft()
Indexed sequence: [6, 0, 0, 0, 0, 0]
 indexed by [0, 1, 2, 3, 4, 5]
```

```
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: 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, -2.999999999999
    indexed by Multiplicative Abelian group isomorphic to C2 x C3
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, -2.999999999999
    indexed by Cyclic group of order 6 as a permutation group
sage: p = 7; J = 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]
```

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 $\mathbf{Q}$ or $\mathbf{C}$ 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 elments in the indexing set.

EXAMPLES:

```
sage: J = 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}
```

dst()

A discrete Sine transform.

EXAMPLES:

```
sage: J = 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: [1.11022302462516e-16 - 2.50000000000000*I, 1.11022302462516e-16 - 2.50000
    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 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 = 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, 0.00000000000000
    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 = 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, 0.00000000000000
    indexed by [0, 1, 2, 3, 4]
```

idft()

A discrete inverse Fourier transform. Only works over **Q**.

EXAMPLES:

```
sage: J = range(5)
sage: A = [ZZ(1) for i in J]
sage: s = IndexedSequence(A,J)
sage: fs = s.dft(); fs
Indexed sequence: [5, 0, 0, 0, 0]
```

```
    indexed by [0, 1, 2, 3, 4]
sage: it = fs.idft(); it
Indexed sequence: [1, 1, 1, 1, 1]
    indexed by [0, 1, 2, 3, 4]
sage: it == s
True
```

idwt(*other='haar'*, *wavelet_k=2*)

Implements 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 = 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, 0.00000000000000C
    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
    indexed by [0, 1, 2, 3, 4, 5, 6, 7]
sage: t.idwt() == s
True
sage: J = 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, 0.00000000000000C
    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
```

ifft()

Implements 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 = 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, 0.00000000000000
    indexed by [0, 1, 2, 3, 4]
sage: t.ifft()
Indexed sequence: [1.00000000000000, 1.00000000000000, 1.00000000000000, 1.00000000000000, 1
    indexed by [0, 1, 2, 3, 4]
sage: t.ifft() == s
1
```

index_object()

Return the indexing object.

EXAMPLES:

```
sage: J = 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]
```

list()

Return the list of `self`.

EXAMPLES:

```
sage: J = 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]
```

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 = range(3)
sage: A = [ZZ(i^2)+1 for i in I]
sage: s = IndexedSequence(A,I)
sage: P = s.plot()
sage: show(P) # Not tested
```

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 $\mathbf{R}$.

Options are `clr`, which is an RGB value, and `eps`, which is the spacing between the bars.

EXAMPLES:

```
sage: J = range(3)
sage: A = [ZZ(i^2)+1 for i in J]
sage: s = IndexedSequence(A,J)
sage: P = s.plot_histogram()
sage: show(P) # Not tested
```

CHAPTER

TWENTY

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.gsl.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)]
```

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)]
```

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)
```

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
```

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)
sage: a.forward_transform()
sage: a.plot().show()
```

`sage.gsl.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)]
```

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)]
```

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)
```

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
```

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)
sage: a.forward_transform()
sage: a.plot().show()
```

class `sage.gsl.fft.`**`FastFourierTransform_base`**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

class `sage.gsl.fft.`**`FastFourierTransform_complex`**

Bases: `sage.gsl.fft.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)
sage: abs(sum([CDF(a[i])/125-CDF(b[i]) for i in range(125)])) < 2**-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)
```

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)]
```

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())
Graphics object consisting of 250 graphics primitives
sage: abs(sum([CDF(a[i])-CDF(b[i]) for i in range(125)])) < 2**-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())
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.

EXAMPLE:

```
sage: a = FastFourierTransform(16)
sage: for i in range(16): a[i] = (random(),random())
sage: A = plot(a)
sage: B = plot(a, style='polar')
sage: type(A)
<class 'sage.plot.graphics.Graphics'>
sage: type(B)
<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
```

class `sage.gsl.fft.`**`FourierTransform_complex`**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

class `sage.gsl.fft.`**`FourierTransform_real`**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

CHAPTER

TWENTYONE

SOLVING ODE NUMERICALLY BY GSL

AUTHORS:

- Joshua Kantor (2004-2006)
- Robert Marik (2010 - fixed docstrings)

class `sage.gsl.ode.`**`PyFunctionWrapper`**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

class `sage.gsl.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 libraries ode solver routines To use it instantiate a class,:

```
sage: 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 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
```

For some algorithms the jacobian must be supplied as well, the form of this should be a function return 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
```

There are a variety of algorithms available for different types of systems. Possible algorithms are

- `rkf45` - runga-kutta-felhberg (4,5)
- `rk2` - embedded runga-kutta (2,3)
- `rk4` - 4th order classical runga-kutta

- `rk8pd` - runga-kutta prince-dormand (8,9)
- `rk2imp` - implicit 2nd order runga-kutta at gaussian points
- `rk4imp` - implicit 4th order runga-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"
```

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]
```

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
```

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 runga-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.
```

```
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: outfile = os.path.join(SAGE_TMP, 'sage.png')
sage: T.plot_solution(filename=outfile)
```

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])
```

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)
```

By default T.plot_solution() plots the y_0, to plot general y_i use:

```
sage: T.plot_solution(i=0, filename=outfile)
sage: T.plot_solution(i=1, filename=outfile)
sage: T.plot_solution(i=2, filename=outfile)
```

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()
sage: f = T.interpolate_solution(i=1)
sage: plot(f,0,12).show()
sage: f = T.interpolate_solution(i=2)
sage: plot(f,0,12).show()
sage: f = T.interpolate_solution()
sage: 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...
```

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_sysem` and overloading `c_f` and `c_j` with C functions that specify the system. The following will work in the notebook:

```
%cython
cimport sage.gsl.ode
import sage.gsl.ode
include 'gsl.pxi'

cdef class van_der_pol(sage.gsl.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: T = ode_solver()                          # not tested
sage: T.algorithm = "bsimp"                     # not tested
sage: vander = van_der_pol()                    # not tested
sage: T.function=vander                         # not tested
sage: T.ode_solve(y_0 = [1,0], t_span=[0,2000], num_points=1000)    # not tested
sage: T.plot_solution(i=0, filename=os.path.join(SAGE_TMP, 'test.png'))        # not tested
```

interpolate_solution(*i=0*)

ode_solve(*t_span=False, y_0=False, num_points=False, params=[ ]*)

plot_solution(*i=0, filename=None, interpolate=False*)

class sage.gsl.ode.**ode_system**

Bases: object

x.__init__(...) initializes x; see help(type(x)) for signature

CHAPTER

TWENTYTWO

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): Trac #12047: return 0 when the integration interval is a point; reformat documentation and add to the reference manual.

class `sage.gsl.integration.`**`PyFunctionWrapper`**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

class `sage.gsl.integration.`**`compiled_integrand`**

Bases: `object`

x.__init__(...) initializes x; see help(type(x)) for signature

`sage.gsl.integration.`**`numerical_integral`**(*func*, *a*, *b=None*, *algorithm='qag'*, *max_points=87*, *params=[ ]*, *eps_abs=1e-06*, *eps_rel=1e-06*, *rule=6*)

Returns 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
  - '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` – absolute and relative error tolerances
- `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)
```

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)
```

The input can be any callable:

```
sage: numerical_integral(lambda x: sin(x)^3 + sin(x),  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
```

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)
```

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]) for n in range(10)]  # random out
[(0.49999999999998657, 5.5511151231256336e-15),
 (0.32175055439664557, 3.5721487367706477e-15),
 (0.24030098317249229, 2.6678768435816325e-15),
 (0.19253082576711697, 2.1375215571674764e-15),
 (0.16087527719832367, 1.7860743683853337e-15),
 (0.13827545676349412, 1.5351659583939151e-15),
 (0.12129975935702741, 1.3466978571966261e-15),
 (0.10806674191683065, 1.1997818507228991e-15),
 (0.09745444625548845, 1.0819617008493815e-15),
 (0.088750683050217577, 9.8533051773561173e-16)]
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
```

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)
```

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)
```

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)
```

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)
```

We can also integrate constant expressions:

```
sage: numerical_integral(2, 1, 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 Trac ticket #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)
```

AUTHORS:

- Josh Kantor
- William Stein
- Robert Bradshaw
- Jeroen Demeyer

ALGORITHM: Uses calls to the GSL (GNU Scientific Library) C library.

TESTS:

Make sure that constant Expressions, not merely uncallable arguments, can be integrated (trac #10088), at least if we can coerce them to float:

```
sage: f, g = x, x-1
sage: numerical_integral(f-g, -2, 2)
(4.0, 0.0)
sage: numerical_integral(SR(2.5), 5, 20)
(37.5, 0.0)
sage: numerical_integral(SR(1+3j), 2, 3)
Traceback (most recent call last):
...
TypeError: unable to simplify to float approximation
```

CHAPTER TWENTYTHREE

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()`.

[BSV] provides an overview of the Riemann map and discusses the research that lead to the creation of this module.

INPUT:

- `fs` – A 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` – A 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
```

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
```

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
```

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
```

Compute rough error for this map:

```
sage: x = 0.75  # long time
sage: print "error =", 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,-
sage: pts.reverse()
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
```

ALGORITHM:

This class computes the Riemann Map via the Szego kernel using an adaptation of the method described by [KT].

REFERENCES:

compute_on_grid(*plot_range*, *x_points*)

Computes the Riemann map on a grid of points. Note that these points are complex of the form z = x + y*i.

INPUT:

- `plot_range` – a tuple of the form `[xmin, xmax, ymin, ymax]`. If the value is `[]`, the default plotting window of the map will be used.
- `x_points` – int, 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: print data[0][8,1]
(-0.0879...+0.9709...j)
```

get_szego(*boundary=-1*, *absolute_value=False*)

Returns 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)
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
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)
```

Getting the szego for a specifc boundary:

```
sage: sz0 = m.get_szego(boundary=0)
sage: sz1 = m.get_szego(boundary=1)
```

get_theta_points(*boundary=-1*)

Returns 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)
Graphics object consisting of 1 graphics primitive
```

Extending the points by a spline:

```
sage: s = spline(points)
sage: s(3*pi / 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)
```

Getting the boundary correspondence for a specifc boundary:

```
sage: tp0 = m.get_theta_points(boundary=0)
sage: tp1 = m.get_theta_points(boundary=1)
```

inverse_riemann_map(*pt*)

Returns 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: import numpy as np
sage: m.inverse_riemann_map(np.complex(-0.2, 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)
```

Default plot:

```
sage: m.plot_boundaries()
Graphics object consisting of 1 graphics primitive
```

Big blue collocation points:

```
sage: m.plot_boundaries(plotjoined=False, rgbcolor=[0,0,1], thickness=6)
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()
Graphics object consisting of 1 graphics primitive
```

Plot zoomed in on a specific spot:

```
sage: m.plot_colored(plot_range=[0,1,.25,.75])
Graphics object consisting of 1 graphics primitive
```

High resolution plot:

```
sage: m.plot_colored(plot_points=1000)  # long time (29s on sage.math, 2012)
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()
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*)

Generates a traditional "spiderweb plot" of the Riemann map. 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)
```

Default plot:

```
sage: m.plot_spiderweb()
Graphics object consisting of 21 graphics primitives
```

Simplified plot with many discrete points:

```
sage: m.plot_spiderweb(spokes=4, circles=1, pts=400, linescale=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)
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()
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,ncorners=4) # long time
sage: p = m.plot_spiderweb(withcolor=True,plot_points=500, thickness = 2.0, min_mag=0.1) # l
```

riemann_map(*pt*)

Returns 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 corrospondance, 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: import numpy as np
sage: m.riemann_map(np.complex(-3, 0.0001))
(1.405757...e-05+8.06...e-10j)
```

sage.calculus.riemann.**analytic_boundary**(*t*, *n*, *epsilon*)

Provides an exact (for n = infinity) Riemann boundary correspondence for the ellipse with axes 1 + epsilon and 1 - epsilon. The boundary is therefore given by e^(I*t)+epsilon*e^(-I*t). It is primarily useful for testing the accuracy of the numerical `Riemann_Map`.

INPUT:

- `t` – The boundary parameter, from 0 to 2*pi

- `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)

TESTS:

Checking the accuracy of this function for different n values:

```
sage: from sage.calculus.riemann import analytic_boundary
sage: t100 = analytic_boundary(pi/2, 100, .3)
sage: abs(analytic_boundary(pi/2, 10, .3) - t100) < 10^-8
True
sage: abs(analytic_boundary(pi/2, 20, .3) - t100) < 10^-15
True
```

Using this to check the accuracy of the Riemann_Map boundary:

```
sage: f(t) = e^(I*t)+.3*e^(-I*t)
sage: fp(t) = I*e^(I*t)-I*.3*e^(-I*t)
sage: m = Riemann_Map([f], [fp],0,200)
sage: s = spline(m.get_theta_points())
sage: test_pt = uniform(0,2*pi)
sage: s(test_pt) - analytic_boundary(test_pt,20, .3) < 10^-4
True
```

`sage.calculus.riemann.`**`analytic_interior`**(*z*, *n*, *epsilon*)

Provides a nearly exact compuation 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.

TESTS:

Testing the accuracy of `Riemann_Map`:

```
sage: from sage.calculus.riemann import analytic_interior
sage: f(t) = e^(I*t)+.3*e^(-I*t)
sage: fp(t) = I*e^(I*t)-I*.3*e^(-I*t)
sage: m = Riemann_Map([f],[fp],0,200)
sage: abs(m.riemann_map(.5)-analytic_interior(.5, 20, .3)) < 10^-4
True
sage: m = Riemann_Map([f],[fp],0,2000)
sage: abs(m.riemann_map(.5)-analytic_interior(.5, 20, .3)) < 10^-6
True
```

`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` – a 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

TESTS:

This is primarily tested implicitly by analytic_interior(). Here is a simple test:

```
sage: from sage.calculus.riemann import cauchy_kernel
sage: cauchy_kernel(.5,(.3, .1+.2*I, 10,'c'))
(-0.584136405997...+0.5948650858950...j)
```

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 – A 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.        ]]])
```

TESTS:

```
sage: complex_to_rgb([[0, 1, 10]])
Traceback (most recent call last):
...
TypeError: Argument 'z_values' has incorrect type (expected numpy.ndarray, got list)
```

sage.calculus.riemann.**complex_to_spiderweb**(*z_values*, *dr*, *dtheta*, *spokes*, *circles*, *rgbcolor*, *thickness*, *withcolor*, *min_mag*)

Converts 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.]],

       [[ 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.        ]]])
```

`sage.calculus.riemann.`**`get_derivatives`**(*z_values*, *xstep*, *ystep*)

Computes 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 comparisions 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...
```

CHAPTER TWENTYFOUR

REAL INTERPOLATION USING GSL

class sage.gsl.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...
```

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...
```

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
```

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...
```

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)]
```

This example is in the GSL documentation:

```
sage: v = [(i + sin(i)/2, i+cos(i^2)) for i in range(10)]
sage: s = spline(v)
sage: show(point(v) + plot(s,0,9, hue=.8))
```

We compute the area underneath the spline:

```
sage: s.definite_integral(0, 9)
41.196516041067...
```

The definite integral is additive:

```
sage: s.definite_integral(0, 4) + s.definite_integral(4, 9)
41.196516041067...
```

Switching the order of the bounds changes the sign of the integral:

```
sage: s.definite_integral(9, 0)
-41.196516041067...
```

We compute the first and second-order derivatives at a few points:

```
sage: s.derivative(5)
-0.16230085261803...
sage: s.derivative(6)
0.20997986285714...
sage: s.derivative(5, order=2)
-3.08747074561380...
sage: s.derivative(6, order=2)
2.61876848274853...
```

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.
```

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)]
```

The spline is recomputed when points are appended (trac ticket #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
```

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
```

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
```

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)]
```

This is a copy of the list, not a reference (trac ticket #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)]
```

`sage.gsl.interpolation.`**spline**

alias of `Spline`

CHAPTER TWENTYFIVE

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)
```

`derivative`(*t*)

Returns 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: cs.derivative(0) - cs.derivative(2 * pi)
0j
sage: cs.derivative(-6)
(2.52047692949...-1.89392588310...j)
```

`value`(*t*)

Returns 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: cs.value(0) - cs.value(2*pi)
0j
sage: cs.value(-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)
```

`derivative`(*t*)

Returns 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: ps.derivative(0) - ps.derivative(2*pi)
0j
sage: ps.derivative(10)
(-1.27323954473...+0j)
```

`value`(*t*)

Returns 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*pi)
0j
sage: ps.value(10)
(0.26760455264...+1j)
```

sage.calculus.interpolators.**complex_cubic_spline**(*pts*)

Creates 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` A 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: show(parametric_plot((fx, fy), (0, 2*pi)))
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())
```

Polygon approximation of a circle:

```
sage: pts = [e^(I*t / 25) for t in xrange(25)]
sage: cs = complex_cubic_spline(pts)
sage: cs.derivative(2)
(-0.0497765406583...+0.151095006434...j)
```

sage.calculus.interpolators.**polygon_spline**(*pts*)

Creates 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` – A 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)))
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())
```

Polygon approximation of an circle:

```
sage: pts = [e^(I*t / 25) for t in xrange(25)]
sage: ps = polygon_spline(pts)
sage: ps.derivative(2)
(-0.0470303661...+0.1520363883...j)
```

CHAPTER TWENTYSIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

[KT] N. Kerzman and M. R. Trummer. “Numerical Conformal Mapping via the Szego kernel”. Journal of Computational and Applied Mathematics, 14(1-2): 111–123, 1986.

[BSV] M. Bolt, S. Snoeyink, E. Van Andel. “Visual representation of the Riemann map and Ahlfors map via the Kerzman-Stein equation”. Involve 3-4 (2010), 405-420.

c

sage.calculus.calculus, 153
sage.calculus.desolvers, 269
sage.calculus.functional, 205
sage.calculus.interpolators, 329
sage.calculus.riemann, 313
sage.calculus.test_sympy, 231
sage.calculus.tests, 235
sage.calculus.wester, 259

g

sage.gsl.dft, 291
sage.gsl.dwt, 287
sage.gsl.fft, 299
sage.gsl.integration, 309
sage.gsl.interpolation, 325
sage.gsl.ode, 305

s

sage.symbolic.assumptions, 131
sage.symbolic.callable, 125
sage.symbolic.complexity_measures, 257
sage.symbolic.expression, 1
sage.symbolic.expression_conversions, 239
sage.symbolic.function, 195
sage.symbolic.function_factory, 199
sage.symbolic.integration.integral, 215
sage.symbolic.relation, 137
sage.symbolic.ring, 189
sage.symbolic.units, 181

A

abs() (sage.symbolic.expression.Expression method), 4
add() (sage.symbolic.expression.Expression method), 5
add_to_both_sides() (sage.symbolic.expression.Expression method), 5
algebraic() (in module sage.symbolic.expression_conversions), 254
AlgebraicConverter (class in sage.symbolic.expression_conversions), 239
analytic_boundary() (in module sage.calculus.riemann), 320
analytic_interior() (in module sage.calculus.riemann), 321
append() (sage.gsl.interpolation.Spline method), 326
arccos() (sage.symbolic.expression.Expression method), 5
arccosh() (sage.symbolic.expression.Expression method), 6
arcsin() (sage.symbolic.expression.Expression method), 7
arcsinh() (sage.symbolic.expression.Expression method), 8
arctan() (sage.symbolic.expression.Expression method), 8
arctan2() (sage.symbolic.expression.Expression method), 9
arctanh() (sage.symbolic.expression.Expression method), 11
args() (sage.symbolic.callable.CallableSymbolicExpressionRing_class method), 128
args() (sage.symbolic.expression.Expression method), 12
arguments() (sage.symbolic.callable.CallableSymbolicExpressionFunctor method), 126
arguments() (sage.symbolic.callable.CallableSymbolicExpressionRing_class method), 128
arguments() (sage.symbolic.expression.Expression method), 12
arithmetic() (sage.symbolic.expression_conversions.AlgebraicConverter method), 239
arithmetic() (sage.symbolic.expression_conversions.Converter method), 240
arithmetic() (sage.symbolic.expression_conversions.FastCallableConverter method), 243
arithmetic() (sage.symbolic.expression_conversions.FastFloatConverter method), 245
arithmetic() (sage.symbolic.expression_conversions.InterfaceInit method), 246
arithmetic() (sage.symbolic.expression_conversions.PolynomialConverter method), 249
arithmetic() (sage.symbolic.expression_conversions.RingConverter method), 251
arithmetic() (sage.symbolic.expression_conversions.SubstituteFunction method), 252
arithmetic() (sage.symbolic.expression_conversions.SympyConverter method), 253
assume() (in module sage.symbolic.assumptions), 133
assume() (sage.symbolic.assumptions.GenericDeclaration method), 131
assume() (sage.symbolic.expression.Expression method), 12
assumptions() (in module sage.symbolic.assumptions), 135
at() (in module sage.calculus.calculus), 159

B

backward_transform() (sage.gsl.dwt.DiscreteWaveletTransform method), 288
backward_transform() (sage.gsl.fft.FastFourierTransform_complex method), 301
base_ring() (sage.gsl.dft.IndexedSequence method), 292
base_units() (in module sage.symbolic.units), 183
binomial() (sage.symbolic.expression.Expression method), 13
BuiltinFunction (class in sage.symbolic.function), 195

C

CallableSymbolicExpressionFunctor (class in sage.symbolic.callable), 126
CallableSymbolicExpressionRing_class (class in sage.symbolic.callable), 127
CallableSymbolicExpressionRingFactory (class in sage.symbolic.callable), 127
canonicalize_radical() (sage.symbolic.expression.Expression method), 13
cauchy_kernel() (in module sage.calculus.riemann), 321
CCSpline (class in sage.calculus.interpolators), 329
characteristic() (sage.symbolic.ring.SymbolicRing method), 189
coeff() (sage.symbolic.expression.Expression method), 16
coefficient() (sage.symbolic.expression.Expression method), 16
coefficients() (sage.symbolic.expression.Expression method), 17
coeffs() (sage.symbolic.expression.Expression method), 18
collect() (sage.symbolic.expression.Expression method), 18
collect_common_factors() (sage.symbolic.expression.Expression method), 20
combine() (sage.symbolic.expression.Expression method), 20
compiled_integrand (class in sage.gsl.integration), 309
complex_cubic_spline() (in module sage.calculus.interpolators), 331
complex_to_rgb() (in module sage.calculus.riemann), 322
complex_to_spiderweb() (in module sage.calculus.riemann), 322
composition() (sage.symbolic.expression_conversions.AlgebraicConverter method), 239
composition() (sage.symbolic.expression_conversions.Converter method), 240
composition() (sage.symbolic.expression_conversions.FastCallableConverter method), 243
composition() (sage.symbolic.expression_conversions.FastFloatConverter method), 245
composition() (sage.symbolic.expression_conversions.InterfaceInit method), 246
composition() (sage.symbolic.expression_conversions.PolynomialConverter method), 250
composition() (sage.symbolic.expression_conversions.RingConverter method), 251
composition() (sage.symbolic.expression_conversions.SubstituteFunction method), 252
composition() (sage.symbolic.expression_conversions.SympyConverter method), 254
compute_on_grid() (sage.calculus.riemann.Riemann_Map method), 315
conjugate() (sage.symbolic.expression.Expression method), 20
construction() (sage.symbolic.callable.CallableSymbolicExpressionRing_class method), 128
content() (sage.symbolic.expression.Expression method), 21
contradicts() (sage.symbolic.assumptions.GenericDeclaration method), 132
contradicts() (sage.symbolic.expression.Expression method), 21
convert() (in module sage.symbolic.units), 183
convert() (sage.symbolic.expression.Expression method), 22
convert_temperature() (in module sage.symbolic.units), 184
Converter (class in sage.symbolic.expression_conversions), 240
convolution() (sage.gsl.dft.IndexedSequence method), 292
convolution_periodic() (sage.gsl.dft.IndexedSequence method), 293
cos() (sage.symbolic.expression.Expression method), 23
cosh() (sage.symbolic.expression.Expression method), 24

create_key() (sage.symbolic.callable.CallableSymbolicExpressionRingFactory method), 127
create_object() (sage.symbolic.callable.CallableSymbolicExpressionRingFactory method), 127
csgn() (sage.symbolic.expression.Expression method), 25

D

dct() (sage.gsl.dft.IndexedSequence method), 293
default_variable() (sage.symbolic.expression.Expression method), 25
default_variable() (sage.symbolic.function.Function method), 195
definite_integral() (sage.gsl.interpolation.Spline method), 326
DefiniteIntegral (class in sage.symbolic.integration.integral), 215
degree() (sage.symbolic.expression.Expression method), 26
denominator() (sage.symbolic.expression.Expression method), 26
deprecated_custom_evalf_wrapper() (in module sage.symbolic.function_factory), 199
DeprecatedSFunction (class in sage.symbolic.function), 195
derivative() (in module sage.calculus.functional), 205
derivative() (sage.calculus.interpolators.CCSpline method), 329
derivative() (sage.calculus.interpolators.PSpline method), 330
derivative() (sage.gsl.interpolation.Spline method), 327
derivative() (sage.symbolic.expression.Expression method), 27
derivative() (sage.symbolic.expression_conversions.Converter method), 240
derivative() (sage.symbolic.expression_conversions.InterfaceInit method), 246
derivative() (sage.symbolic.expression_conversions.SubstituteFunction method), 252
desolve() (in module sage.calculus.desolvers), 269
desolve_laplace() (in module sage.calculus.desolvers), 274
desolve_mintides() (in module sage.calculus.desolvers), 275
desolve_odeint() (in module sage.calculus.desolvers), 276
desolve_rk4() (in module sage.calculus.desolvers), 278
desolve_rk4_determine_bounds() (in module sage.calculus.desolvers), 279
desolve_system() (in module sage.calculus.desolvers), 279
desolve_system_rk4() (in module sage.calculus.desolvers), 280
desolve_tides_mpfr() (in module sage.calculus.desolvers), 281
dft() (sage.gsl.dft.IndexedSequence method), 293
dict() (sage.gsl.dft.IndexedSequence method), 294
diff() (in module sage.calculus.functional), 206
diff() (sage.symbolic.expression.Expression method), 28
differentiate() (sage.symbolic.expression.Expression method), 30
DiscreteWaveletTransform (class in sage.gsl.dwt), 288
divide_both_sides() (sage.symbolic.expression.Expression method), 31
dst() (sage.gsl.dft.IndexedSequence method), 294
dummy_diff() (in module sage.calculus.calculus), 160
dummy_integrate() (in module sage.calculus.calculus), 160
dummy_inverse_laplace() (in module sage.calculus.calculus), 160
dummy_laplace() (in module sage.calculus.calculus), 160
dummy_limit() (in module sage.calculus.calculus), 161
DWT() (in module sage.gsl.dwt), 287
dwt() (sage.gsl.dft.IndexedSequence method), 294

E

eulers_method() (in module sage.calculus.desolvers), 282
eulers_method_2x2() (in module sage.calculus.desolvers), 283

eulers_method_2x2_plot() (in module sage.calculus.desolvers), 284
eval_on_operands() (in module sage.symbolic.function_factory), 199
evalunitdict() (in module sage.symbolic.units), 185
exp() (sage.symbolic.expression.Expression method), 31
exp_simplify() (sage.symbolic.expression.Expression method), 32
expand() (in module sage.calculus.functional), 207
expand() (sage.symbolic.expression.Expression method), 32
expand_log() (sage.symbolic.expression.Expression method), 33
expand_rational() (sage.symbolic.expression.Expression method), 34
expand_sum() (sage.symbolic.expression.Expression method), 35
expand_trig() (sage.symbolic.expression.Expression method), 36
Expression (class in sage.symbolic.expression), 3
ExpressionIterator (class in sage.symbolic.expression), 122

F

factor() (sage.symbolic.expression.Expression method), 36
factor_list() (sage.symbolic.expression.Expression method), 37
factorial() (sage.symbolic.expression.Expression method), 38
factorial_simplify() (sage.symbolic.expression.Expression method), 39
FakeExpression (class in sage.symbolic.expression_conversions), 242
fast_callable() (in module sage.symbolic.expression_conversions), 255
fast_float() (in module sage.symbolic.expression_conversions), 255
FastCallableConverter (class in sage.symbolic.expression_conversions), 242
FastFloatConverter (class in sage.symbolic.expression_conversions), 244
FastFourierTransform() (in module sage.gsl.fft), 300
FastFourierTransform_base (class in sage.gsl.fft), 300
FastFourierTransform_complex (class in sage.gsl.fft), 300
FFT() (in module sage.gsl.fft), 299
fft() (sage.gsl.dft.IndexedSequence method), 295
find() (sage.symbolic.expression.Expression method), 39
find_local_maximum() (sage.symbolic.expression.Expression method), 40
find_local_minimum() (sage.symbolic.expression.Expression method), 40
find_root() (sage.symbolic.expression.Expression method), 41
forget() (in module sage.symbolic.assumptions), 136
forget() (sage.symbolic.assumptions.GenericDeclaration method), 132
forget() (sage.symbolic.expression.Expression method), 42
forward_transform() (sage.gsl.dwt.DiscreteWaveletTransform method), 288
forward_transform() (sage.gsl.fft.FastFourierTransform_complex method), 301
FourierTransform_complex (class in sage.gsl.fft), 303
FourierTransform_real (class in sage.gsl.fft), 303
fraction() (sage.symbolic.expression.Expression method), 43
full_simplify() (sage.symbolic.expression.Expression method), 43
Function (class in sage.symbolic.function), 195
function() (in module sage.symbolic.function_factory), 199
function() (sage.symbolic.expression.Expression method), 44
function_factory() (in module sage.symbolic.function_factory), 202

G

gamma() (sage.symbolic.expression.Expression method), 45
gcd() (sage.symbolic.expression.Expression method), 46

GenericDeclaration (class in sage.symbolic.assumptions), 131
get_derivatives() (in module sage.calculus.riemann), 323
get_fake_div() (sage.symbolic.expression_conversions.Converter method), 241
get_sfunction_from_serial() (in module sage.symbolic.function), 196
get_szego() (sage.calculus.riemann.Riemann_Map method), 315
get_theta_points() (sage.calculus.riemann.Riemann_Map method), 316
GinacFunction (class in sage.symbolic.function), 196
gradient() (sage.symbolic.expression.Expression method), 46

H

has() (sage.symbolic.assumptions.GenericDeclaration method), 133
has() (sage.symbolic.expression.Expression method), 47
has_wild() (sage.symbolic.expression.Expression method), 47
hessian() (sage.symbolic.expression.Expression method), 48
hypergeometric_simplify() (sage.symbolic.expression.Expression method), 48

I

idft() (sage.gsl.dft.IndexedSequence method), 295
idwt() (sage.gsl.dft.IndexedSequence method), 296
ifft() (sage.gsl.dft.IndexedSequence method), 296
imag() (sage.symbolic.expression.Expression method), 48
imag_part() (sage.symbolic.expression.Expression method), 49
implicit_derivative() (sage.symbolic.expression.Expression method), 50
IndefiniteIntegral (class in sage.symbolic.integration.integral), 215
index_object() (sage.gsl.dft.IndexedSequence method), 297
IndexedSequence (class in sage.gsl.dft), 292
integral() (in module sage.calculus.functional), 208
integral() (in module sage.symbolic.integration.integral), 215
integral() (sage.symbolic.expression.Expression method), 51
integrate() (in module sage.calculus.functional), 210
integrate() (in module sage.symbolic.integration.integral), 222
integrate() (sage.symbolic.expression.Expression method), 51
InterfaceInit (class in sage.symbolic.expression_conversions), 246
interpolate_solution() (sage.gsl.ode.ode_solver method), 308
inverse_laplace() (in module sage.calculus.calculus), 161
inverse_laplace() (sage.symbolic.expression.Expression method), 52
inverse_riemann_map() (sage.calculus.riemann.Riemann_Map method), 317
inverse_transform() (sage.gsl.fft.FastFourierTransform_complex method), 301
is2pow() (in module sage.gsl.dwt), 289
is_CallableSymbolicExpression() (in module sage.symbolic.callable), 128
is_CallableSymbolicExpressionRing() (in module sage.symbolic.callable), 128
is_constant() (sage.symbolic.expression.Expression method), 52
is_exact() (sage.symbolic.ring.SymbolicRing method), 189
is_Expression() (in module sage.symbolic.expression), 122
is_field() (sage.symbolic.ring.SymbolicRing method), 189
is_inexact() (in module sage.symbolic.function), 196
is_infinity() (sage.symbolic.expression.Expression method), 53
is_integer() (sage.symbolic.expression.Expression method), 53
is_negative() (sage.symbolic.expression.Expression method), 53
is_negative_infinity() (sage.symbolic.expression.Expression method), 53

is_numeric() (sage.symbolic.expression.Expression method), 53
is_polynomial() (sage.symbolic.expression.Expression method), 54
is_positive() (sage.symbolic.expression.Expression method), 54
is_positive_infinity() (sage.symbolic.expression.Expression method), 55
is_real() (sage.symbolic.expression.Expression method), 55
is_relational() (sage.symbolic.expression.Expression method), 55
is_series() (sage.symbolic.expression.Expression method), 55
is_symbol() (sage.symbolic.expression.Expression method), 56
is_SymbolicEquation() (in module sage.symbolic.expression), 122
is_SymbolicExpressionRing() (in module sage.symbolic.ring), 191
is_SymbolicVariable() (in module sage.symbolic.ring), 192
is_terminating_series() (sage.symbolic.expression.Expression method), 56
is_trivial_zero() (sage.symbolic.expression.Expression method), 57
is_unit() (in module sage.symbolic.units), 185
is_unit() (sage.symbolic.expression.Expression method), 57
isidentifier() (in module sage.symbolic.ring), 192
iterator() (sage.symbolic.expression.Expression method), 58

L

laplace() (in module sage.calculus.calculus), 161
laplace() (sage.symbolic.expression.Expression method), 58
lcm() (sage.symbolic.expression.Expression method), 58
leading_coeff() (sage.symbolic.expression.Expression method), 59
leading_coefficient() (sage.symbolic.expression.Expression method), 59
left() (sage.symbolic.expression.Expression method), 60
left_hand_side() (sage.symbolic.expression.Expression method), 60
lhs() (sage.symbolic.expression.Expression method), 60
lim() (in module sage.calculus.calculus), 163
lim() (in module sage.calculus.functional), 212
limit() (in module sage.calculus.calculus), 166
limit() (in module sage.calculus.functional), 213
limit() (sage.symbolic.expression.Expression method), 60
list() (sage.gsl.dft.IndexedSequence method), 297
list() (sage.gsl.interpolation.Spline method), 327
list() (sage.symbolic.expression.Expression method), 60
log() (sage.symbolic.expression.Expression method), 61
log_expand() (sage.symbolic.expression.Expression method), 62
log_gamma() (sage.symbolic.expression.Expression method), 63
log_simplify() (sage.symbolic.expression.Expression method), 64
low_degree() (sage.symbolic.expression.Expression method), 66

M

mapped_opts() (in module sage.calculus.calculus), 169
match() (sage.symbolic.expression.Expression method), 66
maxima_methods() (sage.symbolic.expression.Expression method), 67
maxima_options() (in module sage.calculus.calculus), 169
merge() (sage.symbolic.callable.CallableSymbolicExpressionFunctor method), 126
minpoly() (in module sage.calculus.calculus), 169
minpoly() (sage.symbolic.expression.Expression method), 67
mul() (sage.symbolic.expression.Expression method), 68

multiply_both_sides() (sage.symbolic.expression.Expression method), 68

N

N() (sage.symbolic.expression.Expression method), 3
n() (sage.symbolic.expression.Expression method), 69
name() (sage.symbolic.function.Function method), 195
negation() (sage.symbolic.expression.Expression method), 70
next() (sage.symbolic.expression.ExpressionIterator method), 122
nintegral() (in module sage.calculus.calculus), 172
nintegral() (sage.symbolic.expression.Expression method), 70
nintegrate() (in module sage.calculus.calculus), 173
nintegrate() (sage.symbolic.expression.Expression method), 70
nops() (sage.symbolic.expression.Expression method), 70
norm() (sage.symbolic.expression.Expression method), 70
normalize() (sage.symbolic.expression.Expression method), 71
number_of_arguments() (sage.symbolic.expression.Expression method), 71
number_of_arguments() (sage.symbolic.function.Function method), 196
number_of_operands() (sage.symbolic.expression.Expression method), 72
numerator() (sage.symbolic.expression.Expression method), 72
numerator_denominator() (sage.symbolic.expression.Expression method), 73
numerical_approx() (sage.symbolic.expression.Expression method), 74
numerical_integral() (in module sage.gsl.integration), 309
NumpyToSRMorphism (class in sage.symbolic.ring), 189

O

ode_solve() (sage.gsl.ode.ode_solver method), 308
ode_solver (class in sage.gsl.ode), 305
ode_system (class in sage.gsl.ode), 308
operands() (sage.symbolic.expression.Expression method), 75
operands() (sage.symbolic.expression_conversions.FakeExpression method), 242
operator() (sage.symbolic.expression.Expression method), 75
operator() (sage.symbolic.expression_conversions.FakeExpression method), 242
Order() (sage.symbolic.expression.Expression method), 4

P

partial_fraction() (sage.symbolic.expression.Expression method), 76
pi() (sage.symbolic.ring.SymbolicRing method), 190
pickle_wrapper() (in module sage.symbolic.function), 197
plot() (sage.gsl.dft.IndexedSequence method), 297
plot() (sage.gsl.dwt.DiscreteWaveletTransform method), 288
plot() (sage.gsl.fft.FastFourierTransform_complex method), 302
plot() (sage.symbolic.expression.Expression method), 77
plot_boundaries() (sage.calculus.riemann.Riemann_Map method), 317
plot_colored() (sage.calculus.riemann.Riemann_Map method), 318
plot_histogram() (sage.gsl.dft.IndexedSequence method), 297
plot_solution() (sage.gsl.ode.ode_solver method), 308
plot_spiderweb() (sage.calculus.riemann.Riemann_Map method), 318
poly() (sage.symbolic.expression.Expression method), 78
polygon_spline() (in module sage.calculus.interpolators), 331
polynomial() (in module sage.symbolic.expression_conversions), 255

polynomial() (sage.symbolic.expression.Expression method), 78
PolynomialConverter (class in sage.symbolic.expression_conversions), 249
power() (sage.symbolic.expression.Expression method), 80
power_series() (sage.symbolic.expression.Expression method), 80
preprocess_assumptions() (in module sage.symbolic.assumptions), 136
primitive_part() (sage.symbolic.expression.Expression method), 80
PrimitiveFunction (in module sage.symbolic.function), 196
PSpline (class in sage.calculus.interpolators), 330
PyFunctionWrapper (class in sage.gsl.integration), 309
PyFunctionWrapper (class in sage.gsl.ode), 305
pyobject() (sage.symbolic.expression.Expression method), 81
pyobject() (sage.symbolic.expression_conversions.AlgebraicConverter method), 240
pyobject() (sage.symbolic.expression_conversions.Converter method), 241
pyobject() (sage.symbolic.expression_conversions.FakeExpression method), 242
pyobject() (sage.symbolic.expression_conversions.FastCallableConverter method), 243
pyobject() (sage.symbolic.expression_conversions.FastFloatConverter method), 245
pyobject() (sage.symbolic.expression_conversions.InterfaceInit method), 248
pyobject() (sage.symbolic.expression_conversions.PolynomialConverter method), 250
pyobject() (sage.symbolic.expression_conversions.RingConverter method), 251
pyobject() (sage.symbolic.expression_conversions.SubstituteFunction method), 252
pyobject() (sage.symbolic.expression_conversions.SympyConverter method), 254

R

radical_simplify() (sage.symbolic.expression.Expression method), 81
rational_expand() (sage.symbolic.expression.Expression method), 81
rational_simplify() (sage.symbolic.expression.Expression method), 82
real() (sage.symbolic.expression.Expression method), 83
real_part() (sage.symbolic.expression.Expression method), 84
rectform() (sage.symbolic.expression.Expression method), 84
reduce_trig() (sage.symbolic.expression.Expression method), 85
relation() (sage.symbolic.expression_conversions.Converter method), 241
relation() (sage.symbolic.expression_conversions.FastCallableConverter method), 243
relation() (sage.symbolic.expression_conversions.FastFloatConverter method), 245
relation() (sage.symbolic.expression_conversions.InterfaceInit method), 248
relation() (sage.symbolic.expression_conversions.PolynomialConverter method), 250
relation() (sage.symbolic.expression_conversions.SubstituteFunction method), 253
residue() (sage.symbolic.expression.Expression method), 86
rhs() (sage.symbolic.expression.Expression method), 86
Riemann_Map (class in sage.calculus.riemann), 313
riemann_map() (sage.calculus.riemann.Riemann_Map method), 320
right() (sage.symbolic.expression.Expression method), 87
right_hand_side() (sage.symbolic.expression.Expression method), 87
RingConverter (class in sage.symbolic.expression_conversions), 250
roots() (sage.symbolic.expression.Expression method), 87
round() (sage.symbolic.expression.Expression method), 89

S

sage.calculus.calculus (module), 153
sage.calculus.desolvers (module), 269
sage.calculus.functional (module), 205

sage.calculus.interpolators (module), 329
sage.calculus.riemann (module), 313
sage.calculus.test_sympy (module), 231
sage.calculus.tests (module), 235
sage.calculus.wester (module), 259
sage.gsl.dft (module), 291
sage.gsl.dwt (module), 287
sage.gsl.fft (module), 299
sage.gsl.integration (module), 309
sage.gsl.interpolation (module), 325
sage.gsl.ode (module), 305
sage.symbolic.assumptions (module), 131
sage.symbolic.callable (module), 125
sage.symbolic.complexity_measures (module), 257
sage.symbolic.expression (module), 1
sage.symbolic.expression_conversions (module), 239
sage.symbolic.function (module), 195
sage.symbolic.function_factory (module), 199
sage.symbolic.integration.integral (module), 215
sage.symbolic.relation (module), 137
sage.symbolic.ring (module), 189
sage.symbolic.units (module), 181
series() (sage.symbolic.expression.Expression method), 90
SFunction (in module sage.symbolic.function), 196
show() (sage.symbolic.expression.Expression method), 91
simplify() (in module sage.calculus.functional), 213
simplify() (sage.symbolic.expression.Expression method), 91
simplify_exp() (sage.symbolic.expression.Expression method), 91
simplify_factorial() (sage.symbolic.expression.Expression method), 91
simplify_full() (sage.symbolic.expression.Expression method), 92
simplify_hypergeometric() (sage.symbolic.expression.Expression method), 93
simplify_log() (sage.symbolic.expression.Expression method), 94
simplify_radical() (sage.symbolic.expression.Expression method), 96
simplify_rational() (sage.symbolic.expression.Expression method), 96
simplify_real() (sage.symbolic.expression.Expression method), 97
simplify_rectform() (sage.symbolic.expression.Expression method), 98
simplify_trig() (sage.symbolic.expression.Expression method), 99
sin() (sage.symbolic.expression.Expression method), 100
sinh() (sage.symbolic.expression.Expression method), 100
solve() (in module sage.symbolic.relation), 142
solve() (sage.symbolic.expression.Expression method), 101
solve_ineq() (in module sage.symbolic.relation), 146
solve_ineq_fourier() (in module sage.symbolic.relation), 147
solve_ineq_univar() (in module sage.symbolic.relation), 148
solve_mod() (in module sage.symbolic.relation), 148
Spline (class in sage.gsl.interpolation), 325
spline (in module sage.gsl.interpolation), 327
sqrt() (sage.symbolic.expression.Expression method), 105
step() (sage.symbolic.expression.Expression method), 105
str_to_unit() (in module sage.symbolic.units), 185

string_length() (in module sage.symbolic.complexity_measures), 257
string_to_list_of_solutions() (in module sage.symbolic.relation), 150
subs() (sage.symbolic.expression.Expression method), 106
subs_expr() (sage.symbolic.expression.Expression method), 108
substitute() (sage.symbolic.expression.Expression method), 109
substitute_expression() (sage.symbolic.expression.Expression method), 110
substitute_function() (sage.symbolic.expression.Expression method), 111
SubstituteFunction (class in sage.symbolic.expression_conversions), 251
subtract_from_both_sides() (sage.symbolic.expression.Expression method), 111
sum() (sage.symbolic.expression.Expression method), 112
symbol() (sage.symbolic.expression_conversions.Converter method), 241
symbol() (sage.symbolic.expression_conversions.FastCallableConverter method), 244
symbol() (sage.symbolic.expression_conversions.FastFloatConverter method), 246
symbol() (sage.symbolic.expression_conversions.InterfaceInit method), 248
symbol() (sage.symbolic.expression_conversions.PolynomialConverter method), 250
symbol() (sage.symbolic.expression_conversions.RingConverter method), 251
symbol() (sage.symbolic.expression_conversions.SubstituteFunction method), 253
symbol() (sage.symbolic.expression_conversions.SympyConverter method), 254
symbol() (sage.symbolic.ring.SymbolicRing method), 190
symbolic_expression_from_maxima_string() (in module sage.calculus.calculus), 175
symbolic_expression_from_string() (in module sage.calculus.calculus), 176
symbolic_sum() (in module sage.calculus.calculus), 176
SymbolicFunction (class in sage.symbolic.function), 196
SymbolicRing (class in sage.symbolic.ring), 189
SympyConverter (class in sage.symbolic.expression_conversions), 253

T

tan() (sage.symbolic.expression.Expression method), 114
tanh() (sage.symbolic.expression.Expression method), 115
taylor() (in module sage.calculus.functional), 214
taylor() (sage.symbolic.expression.Expression method), 116
test_relation() (sage.symbolic.expression.Expression method), 116
test_relation_maxima() (in module sage.symbolic.relation), 150
the_SymbolicRing() (in module sage.symbolic.ring), 192
trailing_coeff() (sage.symbolic.expression.Expression method), 117
trailing_coefficient() (sage.symbolic.expression.Expression method), 118
trait_names() (sage.symbolic.units.Units method), 182
trig_expand() (sage.symbolic.expression.Expression method), 118
trig_reduce() (sage.symbolic.expression.Expression method), 119
trig_simplify() (sage.symbolic.expression.Expression method), 119
truncate() (sage.symbolic.expression.Expression method), 120
tuple() (sage.symbolic.expression_conversions.FastCallableConverter method), 244
tuple() (sage.symbolic.expression_conversions.InterfaceInit method), 248

U

UnderscoreSageMorphism (class in sage.symbolic.ring), 191
unify_arguments() (sage.symbolic.callable.CallableSymbolicExpressionFunctor method), 126
unit() (sage.symbolic.expression.Expression method), 120
unit_content_primitive() (sage.symbolic.expression.Expression method), 121
unit_derivations_expr() (in module sage.symbolic.units), 186

unitdocs() (in module sage.symbolic.units), 186
UnitExpression (class in sage.symbolic.units), 182
Units (class in sage.symbolic.units), 182
unpickle_function() (in module sage.symbolic.function_factory), 203
unpickle_wrapper() (in module sage.symbolic.function), 197

V

value() (sage.calculus.interpolators.CCSpline method), 329
value() (sage.calculus.interpolators.PSpline method), 330
var() (in module sage.symbolic.ring), 193
var() (sage.symbolic.ring.SymbolicRing method), 190
var_cmp() (in module sage.calculus.calculus), 178
variables() (sage.symbolic.expression.Expression method), 121
variables() (sage.symbolic.function.Function method), 196
vars_in_str() (in module sage.symbolic.units), 186

W

WaveletTransform() (in module sage.gsl.dwt), 288
wild() (sage.symbolic.ring.SymbolicRing method), 191

Z

zeta() (sage.symbolic.expression.Expression method), 121